Contents

1 Introduction 13

2 Repository API 17
   Repository Architecture 18
   Repository Items 18
   Item Descriptors 19
   MutableRepository 21
   Core Repository API Elements 21
      atg.repository.Repository 21
      atg.repository.RepositoryView 22
      atg.repository.RepositoryItem 23
      atg.repository.MutableRepository 23
      atg.repository.PropertiesChangedEvent 25
   Cloning Repository Items 26

3 Repository Queries 29
   Repository Query API 29
      atg.repository.QueryBuilder 29
      atg.repository.QueryOptions 30
   Repository Query Examples 31
   Repository Queries in the ATG Control Center 34
   Repository Query Language 34
      RQL Overview 35
      Comparison Queries 35
      Text Comparison Queries 36
      Date and Timestamp Queries 36
      Property of Property Queries 37
      Logical Operators 37
      Multi-Valued Property Queries 37
      INCLUDES ITEM 38
      IS NULL 38
      COUNT 38
      ALL 38
      Full Text Search Queries 39
      ID-based Queries 39
      ORDER BY 40
      RANGE 40
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>Parameters in Queries</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>Parameterized Field Queries</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>RQL Examples</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>RQL Grammar</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>SQL Repository Overview</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>SQL Repository Architecture</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>Repositories and Transactions</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Managing Transactions</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Repository Definition Files</td>
<td>50</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Default Values and XML File Combination</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>SQL Repository Items</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>SQL Repository Item Descriptors</td>
<td>51</td>
</tr>
<tr>
<td>6</td>
<td>SQL Repository Data Models</td>
<td>53</td>
</tr>
<tr>
<td>6.1</td>
<td>Primary and Auxiliary Tables</td>
<td>53</td>
</tr>
<tr>
<td>6.2</td>
<td>id Property</td>
<td>53</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Compound Repository IDs</td>
<td>54</td>
</tr>
<tr>
<td>6.2.2</td>
<td>IdSpaces and the id Property</td>
<td>56</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Database Sequences and Repository IDs</td>
<td>57</td>
</tr>
<tr>
<td>6.3</td>
<td>Auxiliary Tables</td>
<td>57</td>
</tr>
<tr>
<td>6.4</td>
<td>References Constraints</td>
<td>57</td>
</tr>
<tr>
<td>6.5</td>
<td>Properties and Database Columns</td>
<td>58</td>
</tr>
<tr>
<td>6.6</td>
<td>One-to-Many Relationships: Multi-Valued Properties</td>
<td>58</td>
</tr>
<tr>
<td>6.7</td>
<td>Operating on Multi-Valued Properties</td>
<td>60</td>
</tr>
<tr>
<td>6.8</td>
<td>Many-to-Many Relationships</td>
<td>61</td>
</tr>
<tr>
<td>6.9</td>
<td>Default Item Descriptor</td>
<td>62</td>
</tr>
<tr>
<td>6.10</td>
<td>Cascading Data Relationships</td>
<td>62</td>
</tr>
<tr>
<td>6.10.1</td>
<td>Cascade Insert</td>
<td>62</td>
</tr>
<tr>
<td>6.10.2</td>
<td>Cascade Update</td>
<td>63</td>
</tr>
<tr>
<td>6.10.3</td>
<td>Cascade Delete</td>
<td>63</td>
</tr>
<tr>
<td>6.10.4</td>
<td>Cascade Example</td>
<td>64</td>
</tr>
<tr>
<td>6.11</td>
<td>Item Descriptor Inheritance</td>
<td>65</td>
</tr>
<tr>
<td>6.11.1</td>
<td>Benefits of Item Descriptor Inheritance</td>
<td>67</td>
</tr>
<tr>
<td>6.11.2</td>
<td>Queries and Item Descriptor Inheritance</td>
<td>69</td>
</tr>
<tr>
<td>6.11.3</td>
<td>Item Descriptor Inheritance with the copy-from Attribute</td>
<td>69</td>
</tr>
<tr>
<td>6.11.4</td>
<td>Limitations of SQL Repository Inheritance</td>
<td>69</td>
</tr>
<tr>
<td>6.12</td>
<td>Derived Properties</td>
<td>70</td>
</tr>
<tr>
<td>6.12.1</td>
<td>Derivation Syntax</td>
<td>70</td>
</tr>
<tr>
<td>6.12.2</td>
<td>Override Properties</td>
<td>72</td>
</tr>
<tr>
<td>6.12.3</td>
<td>Properties Derived from the Same Item</td>
<td>72</td>
</tr>
<tr>
<td>6.12.4</td>
<td>Complex Derivations</td>
<td>73</td>
</tr>
<tr>
<td>6.12.5</td>
<td>Derivation Methods</td>
<td>73</td>
</tr>
<tr>
<td>6.13</td>
<td>Repository Items and Session Backup</td>
<td>77</td>
</tr>
</tbody>
</table>
7 SQL Repository Item Properties

Enumerated Properties
- enumerated
- enumerated String

Required Properties

Unique Properties

Date and Timestamp Properties
- Last-Modified Properties

Null Properties

Grouping and Sorting Properties

Property Validation with a Property Editor Class

Maintaining Item Concurrency with the Version Property

Repository Items as Properties
- Multiple Item Properties
- Adding an Item to a Multi-Item Property
- Querying Subproperties

Transient Properties

Assigning FeatureDescriptorValues with the <attribute> Tag
- Attributes Used in the ACC

Linking between Repositories

SQL Types and Repository Data Types

User-Defined Property Types
- Identifying a User-Defined Property Type
- Using the property-type Attribute
- Implementing a User-Defined Property Type
- Property Conversion Methods
- Null Values in User-Defined Property Types
- User-Defined Properties and the ACC
- User-Defined Property Type Examples

Property Fetching

Handling Large Database Columns

8 SQL Repository Queries

Repository Filtering

<filterQuery> and rqlFilterString Properties

Overriding RQL-Generated SQL

Parameterized Queries
- Parameterized Query API
- Query Types that Support Parameters
- QueryCache and Parameterized Queries
- Parameterized Query Example

Named Queries
- Named Queries in an SQL Repository Definition File
- Java Code Access to Named Queries

Text Search Queries
<item-descriptor> 183
<property> 189
<derivation> 193
<option> 194
<attribute> 195
<table> 196
<expression> 197
<rql-filter> 197
<named-query> 198
<rql-query> 198
<rql> 198
<param> 198
<sql-query> 198
<sql> 198
<input-parameter-types> 199
<returns> 199
<dependencies> 199
<transaction> 199
<rollback-transaction> 200
<add-item> 200
<update-item> 201
<remove-item> 202
<remove-all-items> 202
<query-items> 202
<print-item> 203
<set-property> 204
<import-items> 205
<export-items> 205
<load-items> 206
<dump-caches> 206
<print-ddl> 207

DTD for SQL Repository Definition Files 208
Sample SQL Repository Definition Files 215
  Simple One-to-One 216
  One-to-One with Auxiliary Table 217
  One-to-Many with an Array 218
  One-to-Many with a Set 219
  One-to-Many with a Map 220
  One-to-Many Mapping to Other Repository Items 222
  Ordered One-to-Many 223
  Many-to-Many 224
  Multi-Column Repository IDs 226
Configuring the SQL Repository Component 228
  Registering a Content Repository 228
  SQL Repository Component Properties 229
## 13 SQL Content Repositories

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting Up an SQL Content Repository</td>
<td>237</td>
</tr>
<tr>
<td>Creating an SQL Content Repository Definition</td>
<td>238</td>
</tr>
<tr>
<td>Folder and Content Item Descriptors</td>
<td>238</td>
</tr>
<tr>
<td>Path and Item ID Attributes</td>
<td>239</td>
</tr>
<tr>
<td>Defining Content Item Descriptors</td>
<td>239</td>
</tr>
<tr>
<td>Content Attributes and Properties</td>
<td>241</td>
</tr>
<tr>
<td>Storing Content on a File System</td>
<td>242</td>
</tr>
<tr>
<td>Content Repository Example</td>
<td>243</td>
</tr>
<tr>
<td>Book Item Type Properties</td>
<td>244</td>
</tr>
<tr>
<td>Locating the Content with Path and Folder Properties</td>
<td>244</td>
</tr>
<tr>
<td>Book Example Repository Definition File</td>
<td>244</td>
</tr>
<tr>
<td>Book Example SQL Table Creation Statements</td>
<td>245</td>
</tr>
<tr>
<td>Adding Content to the Content Repository</td>
<td>246</td>
</tr>
<tr>
<td>Accessing Items in the Content Repository</td>
<td>246</td>
</tr>
<tr>
<td>Configuring an SQL Content Repository</td>
<td>247</td>
</tr>
</tbody>
</table>

## 14 Repository Loader

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository Loader Architecture</td>
<td>249</td>
</tr>
<tr>
<td>Repository Loader Components</td>
<td>251</td>
</tr>
<tr>
<td>FileSystemMonitorScheduler</td>
<td>251</td>
</tr>
<tr>
<td>FileSystemMonitorService</td>
<td>252</td>
</tr>
<tr>
<td>LoaderManager</td>
<td>253</td>
</tr>
<tr>
<td>TypeMapper and TypeMappings</td>
<td>256</td>
</tr>
<tr>
<td>ContentHandlers</td>
<td>259</td>
</tr>
<tr>
<td>Repository Loader Administration</td>
<td>260</td>
</tr>
<tr>
<td>RLClient</td>
<td>260</td>
</tr>
<tr>
<td>Supplemental RLClient Parameters</td>
<td>261</td>
</tr>
<tr>
<td>Repository Loader Manifest</td>
<td>262</td>
</tr>
<tr>
<td>Manifest File Tags and Attributes</td>
<td>263</td>
</tr>
<tr>
<td>Importing Versioned Repository Data</td>
<td>263</td>
</tr>
<tr>
<td>Configuring the VersionedLoaderEventListener</td>
<td>264</td>
</tr>
<tr>
<td>Importing Targeters that Reference rules Files</td>
<td>267</td>
</tr>
<tr>
<td>Configuring TypeMapping Components for the PublishingFileRepository</td>
<td>268</td>
</tr>
<tr>
<td>Repository Loader Example</td>
<td>268</td>
</tr>
<tr>
<td>User Item Type</td>
<td>270</td>
</tr>
<tr>
<td>Item Pathnames</td>
<td>271</td>
</tr>
<tr>
<td>Type Mappings and Content Handlers</td>
<td>272</td>
</tr>
<tr>
<td>TypeMapper</td>
<td>272</td>
</tr>
<tr>
<td>xml2repository Schemas</td>
<td>273</td>
</tr>
<tr>
<td>Running the Repository Loader Example</td>
<td>273</td>
</tr>
</tbody>
</table>

## 15 Repository Web Services

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetRepositoryItem Web Service</td>
<td>275</td>
</tr>
<tr>
<td>PerformRQLQuery Web Service</td>
<td>277</td>
</tr>
<tr>
<td>PerformRQLCountQuery Web Service</td>
<td>279</td>
</tr>
</tbody>
</table>
16 Composite Repositories

Use Example 283
Primary and Contributing Item Descriptors 283
Item Inheritance and Composite Repositories 284
Transient Properties and Composite Repositories 284
Non-Serializable Items and Composite Repositories 284
Property Derivation 284
Configuring a Composite Repository 285
Property Mappings 285
Excluding Properties 286
Link Methods 286
Creating Composite and Contributing Items 287
Missing Contributing Items 287
Configuring the Composite Repository Component 288

Composite Repository Queries 288
Composite Repository Caching 289
Composite Repository Definition Tag Reference 289
<composite-repository-template> 289
<header> (composite repository) 289
<item-descriptor> composite repository 290
<primary-item-descriptor> 292
<contributing-item-descriptor> 292
<attribute> composite repository 293
<property> composite repository 294
<primary-item-descriptor-link> 295
<link-via-id> 296
<link-via-property> 296

DTD for Composite Repository Definition Files 296
Sample Composite Repository Definition File 299

17 Secured Repositories 305

Features and Architecture 305
Creating a Secured Repository 307
Modify the Underlying Repository 308
Configure the Secured Repository Adapter Component 309
Register the Secured Repository Adapter Component 310
Create the Secured Repository Definition File 310

Secured Repository Example 312
Modify the SQL for the Repository Data Store 312
Modify the XML definition file 313
Define the Secured Repository Adapter’s Definition File 314
Configure a Secured Repository Adapter Component 315
Register the Repositories 316

ACL Syntax 316
LDAP Password Encryption 351
LDAP Repository Definition Tag Reference 351
  <!DOCTYPE> LDAP repository 352
  <ldap-adapter-template> 352
  <header> LDAP repository 352
  <view> 352
  <item-descriptor> LDAP repository 353
  <id-property> 353
  <object-classes-property> 354
  <object-class> 355
  <property> LDAP repository 356
  <option> LDAP repository 358
  <attribute> LDAP repository 358
  <child-property> 359
  <new-items> 360
  <search-root> 360
Sample LDAP Repository Definition File 361
DTD for LDAP Repository Definition Files 363

Index 367
Data access is a large part of most Internet applications. ATG Data Anywhere Architecture™ provides a unified view of content and data across a business for organizations and their customers. The core of the ATG Data Anywhere Architecture is the Repository API. Through the Repository API, you can employ a single approach to accessing disparate data types, including SQL databases, LDAP directories, content management systems, and file systems.

With the ATG Data Anywhere, the application logic created by developers uses the same approach to interact with data regardless of the source of that data. One of the most powerful aspects of this architecture is that the source of the data is hidden behind the ATG Repository abstraction. It is easy to change from a relational data source to an LDAP directory as none of the application logic needs to change. After data is retrieved from a data source, it is transformed into an object-oriented representation. Manipulation of the data can be done using simple `getPropertyValue` and `setPropertyValue` methods. The Repository API ties in closely with ATG’s targeting APIs, so you can retrieve items from the repository based on a variety of targeting rules, as well as retrieving specific identified items.

The figure below provides a high-level overview of the ATG Data Anywhere Architecture.
ATG Data Anywhere Architecture offers several advantages over the standard data access methods such as Java Data Objects (JDO), Enterprise JavaBeans (EJB), and Java Database Connectivity (JDBC). Among the differences:

**Data source independence**

ATG Data Anywhere Architecture provides access to relational database management systems, LDAP directories, and file systems using the same interfaces. This insulates application developers from schema changes and also storage mechanism. Data can even move from a relational database to an LDAP directory without requiring recoding. Java Data Objects support data source independence, but it is up to vendors to provide an LDAP implementation.

**Fewer lines of Java code**

Less code leads to faster time-to-market and reduced maintenance cost. Persistent data types created with ATG Data Anywhere are described in an XML file, with no Java code required.

**Unified view of all customer interactions**

A unified view of customer data (gathered by web applications, call center applications, and ERP systems) can be provided without copying data into a central data source. This unified view of customer data leads to a coherent and consistent customer experience.

**Maximum performance**

Intelligent caching of data objects ensures excellent performance and timely, accurate results. The JDO and EJB standards rely on a vendor implementation of caching that might not be available.
**Simplified transactional control**

The key to overall system performance is minimizing the impact of transactions while maintaining the integrity of your data. In addition to full Java Transaction API (JTA) support, ATG Data Anywhere lets both page developers and software engineers control the scope of transactions with the same transactional modes—required, supports, never—used by EJB deployment engineers.

**Fine-grained access control**

You can control who has access to which data at the data type, data object, even down to the individual property with Access Control Lists (ACLs).

**Integration with ATG product suites**

ATG personalization, scenarios, commerce, portal, and content administration applications all make use of repositories for data access. A development team is free to use EJBs along side of ATG technology, but the easiest way to leverage investment in ATG technology is to follow the example set by the solution sets. The ATG solution sets satisfy all their data access needs with repositories.
The ATG Repository API (atg.repository.*) is the foundation of persistent object storage, user profiling, and content targeting in ATG products. A repository is a data access layer that defines a generic representation of a data store. Application developers use this generic representation to access data by using only interfaces such as Repository and RepositoryItem. Repositories access the underlying data storage device through a connector, which translates the request into whatever calls are needed to access that particular data store. Connectors for relational databases and LDAP directories are provided out-of-the-box. Connectors use an open, published interface, so additional custom connectors can be added if necessary.

Developers use repositories to create, query, modify, and remove repository items. A repository item is like a JavaBean, but its properties are determined dynamically at runtime. From the developer’s perspective, the available properties in a particular repository item depend on the type of item they are working with. One item might represent the user profile (name, address, phone number), while another might represent the meta-data associated with a news article (author, keywords, synopsis).

The purpose of the Repository interface system is to provide a unified perspective for data access. For example, developers can use targeting rules with the same syntax to find people or content.

Applications that use only the Repository interfaces to access data can interface to any number of back-end data stores solely through configuration. Developers do not need to write a single interface or Java class to add a new persistent data type to an application.

Each repository connects to a single data store, but multiple repositories can coexist within ATG products, where various applications and subsystems use different repositories or share the same repository. Applications that use only the Repository API to access data can interface to any number of back-end data stores solely through configuration. For example, the security system can be directed to maintain its list of usernames and passwords in an SQL database by pointing the security system at an SQL repository. Later, the security system can be changed to use an LDAP directory by reconfiguring it to point to an LDAP repository. Which repositories you use depends on the data access needs of your application, including the possible requirement to access data in a legacy data store.

The ATG platform includes the following models for repositories:

- SQL repositories use ATG’s Generic SQL Adapter (GSA) connector to map between ATG and the data in an SQL database. You can use an SQL repository to access content, user profiles, application security information, and more.
- SQL profile repository, included in the ATG Personalization module, uses the Generic SQL Adapter connector to map user data that is contained in an SQL database. See the ATG Personalization Programming Guide.
- **LDAP repositories** use the ATG LDAP connector to access user data in an LDAP directory. See the LDAP Repositories chapter.

- **Composite repositories** let you use multiple data stores as sources for a single repository.

- **Versioned repositories** extend the SQL repositories and are used in ATG Content Administration. See the ATG Content Administration Programming Guide.

When you store a document in a repository, in addition to the document meta-information, you need access to the physical content and path information that tells you where the document is stored. Content-specific repository extensions handle this. These are located in the atg.repository.content package, described later in the SQL Content Repositories chapter.

### Repository Architecture

A data store can contain many types of objects. The repository is not the data store itself; rather, it is composed of JavaBeans whose properties can be found and stored in the data store. The repository provides a mechanism to retrieve the data elements, and creates a run-time representation of the available metadata for each object. This goal is achieved through three main conceptual parts of the Repository API:

- Repository items
- Item descriptors
- Repository queries

For example, a repository might track elements of an organization. Each employee has a corresponding repository item, as does each department. An employee item descriptor specifies all properties that an employee repository item can possess; a department item descriptor specifies all the possible properties of a department. An application can build queries that return the appropriate employee or department repository items as they are needed by the application.

### Repository Items

A repository is a collection of repository items. A repository item is a JavaBean component that implements atg.repository.RepositoryItem or one of its sub-interfaces, and corresponds to the smallest uniquely identifiable entity in the underlying data store. In the SQL repository, for example, a repository item often corresponds roughly to a row in a table. In the SQL profile repository, each user profile is a repository item.

**Properties**

Each repository item is composed of named properties that store the item’s data—for example, id, firstName, and lastName. In the SQL repository, these properties generally correspond to table columns. Repository item properties are defined in its item descriptor.
Repository item properties can be single-or multi-valued. In some repository implementations such as the SQL repository, a property’s value can refer to one or more other repository items, either in the same or another repository. This enables a repository item to use properties that are complex data structures.

**Repository IDs**

Each repository item must have an identifier, which is called a repository ID. The repository ID must uniquely differentiate the repository item from all other repository items of the same type. The repository is typically configured to find the repository ID from some elements of the underlying data. In the SQL repository, for instance, each item descriptor must specify the columns that act as the repository ID, usually the same as the table’s primary key. Depending on the repository’s configuration, the repository ID might not be exposed as a property of the repository item.

The combination of item descriptors, properties, identifiers, and items allows a repository to read application data from the underlying data store and write application data back to it. If desired, a repository can be defined to expose certain properties, item descriptors, or the entire repository as read-only. Properties can also act as translators between the underlying data source and the Java application. For example, you might want your ATG application to display last names in upper case. You can define a repository property named `lastNameUpperCase` that takes the last name value from the database and returns it in upper case. The Repository API provides this and other options without requiring you to modify any application code.

**Item Descriptors**

Each repository item belongs to an item type that is defined by a Repository item descriptor. An item descriptor implements the `atg.repository.RepositoryItemDescriptor` interface and can subclass `atg.repository.ItemDescriptorImpl`. An item descriptor provides the following information:

- The item type’s name
- Item type properties
- The class of the Java object used to represent the item type—for example, `Integer` or `String`

Repository item descriptors depend on a combination of the underlying data store and the configuration of the repository. In the SQL repository, for example, an item descriptor might correspond to a single database table, or encompass multiple tables.

Item descriptors are defined in an XML repository definition file. A repository can support multiple item descriptors. For example, the figure below shows two item descriptors that are defined in a Visitor Profile repository that stores customer data: `user` and `address`. Each item descriptor is typically implemented by multiple items. The example below shows how item descriptors user and address are implemented by multiple items:
A property in one repository item can be linked to another type of repository item which allows developers to map relationships (one-to-one, one-to-many, etc.).

As shown in this figure, you can model relationships between item types. In this example, item type user has an address property. The value of the address property is a repository item of type address.

The ItemDescriptor mechanism is built upon ATG Dynamic Beans (atg.beans.*) system (described in the Nucleus: Organizing JavaBean Components chapter of the ATG Programming Guide), which lets you describe properties for Java objects without defining the getX and setX methods for each property as required by the JavaBean specification. This interface is used to describe a set of dynamic properties that occur together and have consistent behavior and semantics. An item descriptor essentially acts like a BeanInfo where one can get access to the PropertyDescriptors that compose the repository item. (For information about BeanInfos and PropertyDescriptors, see the JSDK 2 API documentation for java.beans.BeanInfo and java.beansPropertyDescriptor.)

Most repositories support simple property types such as Strings and Integers. ATG repositories can also use the Java Collections Framework to model complex relationships between items using familiar object-oriented concepts. Thus, an item property can store as its value:

- A list of multiple values as a Set, List, Map, or array.
- Another repository item or multiple repository items.

For example, a repository might have item descriptors Person and Address. The Person item descriptor might have an addresses property that exposes a list of addresses. This property might be of type RepositoryItem[], and the repository items in that array use the Address item descriptor. This allows repositories to represent one-to-one, one-to-many, or many-to-many relationships.

The information stored in ItemDescriptor components is usually not needed to develop ATG applications. This property metadata is available for applications that provide a user interface for exploring and navigating a repository. For example, the ATG Control Center uses repository ItemDescriptors to limit the selections available to its users.
MutableRepository

The base interfaces of a repository define an immutable data store. It provides a read-only version of the elements contained in the repository. Extensions of the Repository interfaces provide facilities to create, update, and remove items from a repository. See \texttt{atg.repository.MutableRepository}. The design goal for updates was to allow transactional integrity across a set of changes in a high performance manner. When an item needs to be updated, a clone of the object is returned and changes are made to the cloned object. The repository carries out those changes only when the object is submitted for an update action.

Generally, repositories use caches to improve performance. Items retrieved out of a repository either through a query process or directly by ID from the repository are cached. Typically, cache policies are based on least recently used (LRU) design patterns or time-based expiration. For more information, see the chapter SQL Repository Caching.

Core Repository API Elements

This section describes the core Repository API for users who wish to extend it. Extensions can include specific form handlers to update user information or specialized queries to search for documents. The following interfaces and classes are described:

- \texttt{atg.repository.Repository}
- \texttt{atg.repository.RepositoryView}
- \texttt{atg.repository.RepositoryItem}
- \texttt{atg.repository.MutableRepository}
- \texttt{atg.repository.PropertiesChangedEvent}

\texttt{atg.repository.Repository}

The base definition of any repository implementation, this interface provides methods to access RepositoryItems, RepositoryViews and ItemDescriptors.

Given a unique ID or set of IDs, you can retrieve repository items with the following methods:

\begin{verbatim}
RepositoryItem getItem(String pId, String pDescriptorName)
RepositoryItem[] getItems(String[] pIds, String pDescriptorName)
\end{verbatim}

Depending on the repository implementation, item IDs can take different forms. In SQL repositories, a repository item ID by default is numeric and is auto-generated through the \texttt{IdGenerator} service, which is described in the Core Dynamo Services chapter of the ATG Programming Guide. The SQL repository also supports composite repository item IDs. In that case, you can retrieve items from the repository with these methods:

\begin{verbatim}
RepositoryItem getItem(CompositeKey pId, String pDescriptorName)
RepositoryItem[] getItems(CompositeKey[] pIds, String pDescriptorName)
\end{verbatim}
In other cases, an item ID might be the path of the document, as in some of the file-system based repositories.

The Repository API includes the `RepositoryItemDescriptor` interface, a subinterface of `atg.beans.DynamicBeanInfo` (see the ATG Programming Guide chapter Nucleus: Organizing JavaBean Components). This lets you access the “dynamic bean info” of the available repository items, such as the property descriptors and property names, with this method:

```java
RepositoryItemDescriptor getItemDescriptor(String pName)
```

You can also get a list of all available ItemDescriptors from the `itemDescriptorNames` property.

**atg.repository.RepositoryView**

If you do not have an exact repository ID, you can search for items in the repository through a `RepositoryView`. Item descriptors and `RepositoryViews` often have a one-to-one relationship and often have the same name. You can find out what views are available through the `viewNames` property of the Repository component. This is useful if you need to build a system to navigate and view the entire contents of a repository. The IDs for items in different item types might not overlap. There might be no view that can return all items in the repository, but if there is, it is the default view. If you need to use the default view, you can use the view named by the `defaultViewName` property. Alternatively, you can create properties for your own services that allow you to explicitly name the view your code is interested in using. After you have a name, you can retrieve that view through the `RepositoryView getView(String pName)` method. From this returned object you can build and execute a query.

The `RepositoryView` can also implement `atg.repository.RepositoryViewContainer` if the repository needs to express a hierarchy of `RepositoryViews`. For example, a document management system might have a root view for all “documents.” From that you might have sub-document types such as white papers, promo blurbs, images, and so on. Further, the sub-view images might also have a refinement for specific image types like JPEG and GIF. You can see what attributes are available for building queries by accessing the `itemDescriptor` property of the `RepositoryView`. This describes all the property information about all the items that can be returned by this view.

Sets of repository items can be gathered by queries, which you can build with the `atg.repository.QueryBuilder` interface. This `QueryBuilder` object can be retrieved from the view’s `queryBuilder` property. After a query is built, each `RepositoryView` implementation translates the internalized data structure into its native query language, for example SQL, and return an array of repository items that match the supplied query criteria. This is the mechanism by which the targeting engine understands how to translate the rule syntax into the appropriate method calls of the `QueryBuilder`.

After a query is built from the `QueryBuilder`, it is executed via by one of several `executeQuery` methods defined in the `RepositoryView` interface. For example:

```java
RepositoryItem[] executeQuery(Query pQuery)
```

`executeQuery` methods allow range queries and sorting criteria to be added to the execution of the query. The methods return one of the following:
- An array of RepositoryItems, for elements that match the query's WHERE clauses.
- Null, if no elements can be found.

**atg.repository.RepositoryItem**

The `atg.repository.RepositoryItem` interface is the immutable interface that represents an element from a repository. Each `RepositoryItem` is uniquely identified through its `repositoryId` property. The ItemDescriptor that describes the dynamic bean info about the item is available through the `itemDescriptor` property. Given the repository item, you can also know what repository it came from with the `repository` property. To retrieve the attributes of the `RepositoryItem`, use the `getPropertyValue(String pPropertyName)` method. You can retrieve subproperties with the `DynamicBeans.getSubPropertyValue` method, which takes a hierarchy property name of the form `propertyName1.subPropertyName2.subSubPropertyName3`.

A Dynamic Bean property mapper is registered for the `RepositoryItem` interface. This lets you reference the names of these properties as though they were JavaBean properties in the ATG Servlet Bean syntax. See the Nucleus: Organizing JavaBean Components chapter of the ATG Programming Guide.

**atg.repository.MutableRepository**

Some repository services implement `MutableRepository`, a subclass of `Repository`. The SQL repository implements this interface. The `MutableRepository` interface defines these methods:

- `createItem()`
- `addItem()`
- `removeItem()`
- `updateItem()`

**createItem() method**

There are two `createItem` methods:

```java
createItem(String pDescriptorName)
createItem(String pId, String pDescriptorName)
```

Each of these requires a `DescriptorName` parameter, which is the name of the `RepositoryView` or ItemDescriptor that describes the repository item to create. Each repository has a default ItemDescriptor, which might allow your code to use the `defaultViewName` property of the repository to supply this value. One of the `createItem` methods takes a potential unique ID to use for the `MutableRepositoryItem` to create. If you do not supply an ID, one is automatically generated and guaranteed to be unique.

In the SQL profile repository, for example, the `createItem` methods return a transient instance of a `MutableRepositoryItem`. At this point, the profile does not exist persistently in a data store. The item exists only as the object reference you are returned. You can try to refetch the object (if the user’s session is not expired or the server did not restart) through the `getItem(String pId, String pDescriptorName)` method of the Repository (unless the `GSARespository.storeTransientItems` property is set to `false`). Maintaining profile RepositoryItems
in RAM rather than in the profile database allows anonymous users to be represented in the same
Repository API, but does not hamper performance for handling requests for large sites. It becomes
untenable to try to create anonymous user database records for web sites that have a large volume of
users.

addItem() method
After you create an item, you can turn it into a persistent repository item with the addItem method. This
takes in the repository item that you want to add persistently:

RepositoryItem addItem(MutableRepositoryItem pItem)

removeItem() method
Removing an item from the repository is very easy. Pass the ID and ItemDescriptor name of the item you
want to remove persistently to the removeItem method. The item's property values are deleted and are
no longer accessible from the repository:

removeItem(String pId, String pDescriptorName)

updateItem() method
The MutableRepository updates a repository item in a transactionally aware manner. It differs from a
standard JavaBean in order to ensure that the update operation in the backend data store (such as a
relational database) is efficient. Thus, updating an item requires three steps:

1. Fetch a mutable version of the repository item through the getItemForUpdate and
   getItemsForUpdate methods. These methods return instances of
   MutableRepositoryItem. This interface extends RepositoryItem and adds one
   method:

   setPropertyValue(String pPropertyName, Object pPropertyValue)

2. Use the setPropertyValue method of MutableRepositoryItem to change as
   many properties as you wish. These changes are not reflected in the repository until
   the final updateItem operation is invoked.

3. Save the changes with the updateItem method. This method extracts all the changes
   required for the item and updates the item in the data store. Depending on how you
   have configured transactional behavior, the update can be committed immediately, or
   it can happen automatically when the associated transaction commits. See
   Repositories and Transactions in the SQL Repository Architecture chapter. If there was
   any type of error, a RepositoryException is thrown.

For example:

```java
try {
    RepositoryItem user = ... // get a reference to the user you want to update
    MutableRepository mutableRepository = (MutableRepository)user.getRepository();
    MutableRepositoryItem mutableUser =
        mutableRepository.getItemForUpdate(user.getRepositoryId()),
```
This same methodology should be applied for RAM-based RepositoryItems that you have created through the `createItem` method. No database transaction is performed, but the values are updated in the repository.

The Dynamo Application Framework (DAF) includes three classes that provide useful methods for dealing with repository items:

- `atg.repository.servlet.RepositoryFormHandler`
- `atg.userprofiling.ProfileForm`
- `atg.userprofiling.ProfileFormHandler`

See the User Profile Forms chapter in the ATG Page Developer’s Guide and the source code for the `ProfileForm` and `ProfileFormHandler` classes, included in the ATG Personalization module distribution in the `<ATG10dir>/DPS/src/java/atg/userprofiling` directory.

**atg.repository.PropertiesChangedEvent**

When a repository item is modified, its item descriptor broadcasts locally a `PropertiesChangedEvent`. This event can be one of the following types:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>the item was removed in this transaction</td>
</tr>
<tr>
<td>UPDATE</td>
<td>properties of an item changed in this transaction</td>
</tr>
<tr>
<td>INSERT</td>
<td>the item was newly added to the database</td>
</tr>
<tr>
<td>CACHE_INVALIDATE</td>
<td>some application code called the <code>removeItemFromCache</code> method</td>
</tr>
</tbody>
</table>

In addition to its type, a `PropertiesChangedEvent` contains the following:
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item</td>
<td>The item that is changed. This is set to null if the item that was modified is not currently in the cache. In that case, look at the repositoryId property for the identity of the item that was changed.</td>
</tr>
<tr>
<td>repositoryId</td>
<td>The repository ID of the item that is changed.</td>
</tr>
<tr>
<td>itemDescriptorName</td>
<td>The item descriptor of the item that is changed.</td>
</tr>
<tr>
<td>properties</td>
<td>A Map where the keys are RepositoryPropertyDescriptors and the values are the new property values. If all properties have changed (or might have changed), a null value is returned for the properties map. Returned only for UPDATE events.</td>
</tr>
</tbody>
</table>

If you have a component that you want to be notified when repository item properties change, it can implement the `atg.repository.PropertiesChangedListener` interface. You can add your `PropertiesChangedListener` implementation to the `atg.repository.ItemDescriptorImpl` returned by the `repository.getItemDescriptor()` method, with the method `ItemDescriptorImpl.addPropertiesChangedListener`.

### Cloning Repository Items

The `atg.repository.RepositoryUtils` class includes a method you can use to clone a repository item. This creates a copy of a repository item in a repository without adding the item to the repository. A copy can be a deep copy or a shallow copy. Furthermore, you can specify a list of properties to exclude from the copy.

The full signature of the `cloneItem` method is:

```java
public static MutableRepositoryItem cloneItem(RepositoryItem pItem, boolean pDeepCopy, Map pPropExceptions, Map pExcludedProperties, MutableRepository pDestRepository, String pId) throws RepositoryException, DuplicateIdException
```

The `cloneItem` method's parameters are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pItem</td>
<td>Item to copy.</td>
</tr>
<tr>
<td>pDeepCopy</td>
<td>Boolean, specifies the mode of the copy:</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>true:</td>
<td>The method creates a deep copy of the item and its properties.</td>
</tr>
<tr>
<td>false:</td>
<td>The method creates a shallow copy, only getting references of child RepositoryItems.</td>
</tr>
<tr>
<td>Shallow copying only works if the source and destination repositories are the same.</td>
<td></td>
</tr>
</tbody>
</table>

| pPropExceptions (optional) | Hierarchical map of property name exceptions to the above mode. Keys are property names, while values are null or, if the property is another repository item, another Map. For example, if you clone a product item using pDeepCopy=true, you can add the key parent Category with a null value into pPropExceptions. This results in a shallow copy of the product.parent Category. |
|                          | Alternatively, you can add the key parent Category but set the value to another map of exceptions that included the key/value pair keywords=null. This results in a deep copy of product.parent Category but a shallow copy of the product.parent Category.keywords. |

| pExcludedProperties (optional) | Properties to exclude from the clone. Keys are item descriptor names and the values are collections of property names to exclude. |

| pDestRepository (optional) | Repository to copy the new item into. If the source and destination repositories are the same, properties that are items are cloned to the repository of the source item-property. |
|                          | If this parameter is omitted, the new item is copied to source repository. |

| pId (optional) | Repository ID to use in the copy of the item. If this parameter is omitted, a unique ID is automatically generated. |
# 3 Repository Queries

A repository query defines a request to find all items of a specified item type that fit a set of criteria. Those criteria are specified in terms of the item type's properties—for example:

find all Person items where age property > 30

The Repository API can express a wide variety of queries, including queries that match patterns in text, query through collections, or even query through complex values. Queries can also specify the order in which to return results, and can specify which results from a large result set to return. For example, a query can return a subset of Person items as follows:

- lastName starts with A.
- interests includes biking.
- addresses property contains an Address item with zipCode set to 02139.
- Sorts results on lastName.
- Returns only items 10-20.

Queries can be built and executed with the Repository API. In cases where the query is complex or cannot use the Repository API directly, queries can be represented in the Repository Query Language (RQL). In most cases, however, repository queries can be constructed easily with targeting UI components in the ATG Control Center.

## Repository Query API

This section describes the basic elements of queries in the Repository API:

- `atg.repository.QueryBuilder`
- `atg.repository.QueryOptions`

`atg.repository.QueryBuilder`

The `atg.repository.QueryBuilder` interface defines the available query operations that repositories support. The `QueryBuilder` interface enables you to build `Query` objects that can be passed to the repository. A `Query` is constructed from `QueryExpressions`. Each `Query` relates one or more `QueryExpressions` and a query operation. Queries can use standard logical query operations such as AND, OR, NOT, EQUALS, GREATER THAN, LESS THAN OR EQUALS, and more complicated query operations such as collection inclusion, and pattern matching.
The **QueryBuilder** implementation is not required to support all query operations—it depends on what query features the data store supports. For unsupported query operations, the method should throw a **RepositoryException**. You can use the `atg.repository.QueryOptions` class to limit the size of a query’s result set or otherwise modify the query.

**Query Creation Example**

The following example creates a query that returns all repository items whose gender property is female:

1. Given a **RepositoryView**, initialize a **QueryBuilder** for it:

   ```java
   QueryBuilder b = view.getQueryBuilder();
   ```

2. Create a **QueryExpression** for the gender property and a **QueryExpression** for the constant female:

   ```java
   QueryExpression gender = b.createPropertyQueryExpression("gender");
   QueryExpression female = b.createConstantQueryExpression("female");
   ```

3. Create a **ComparisonQuery** that incorporates the gender and female **QueryExpressions**:

   ```java
   Query femaleQuery = b.createComparisonQuery(gender, female,
   QueryBuilder.EQUALS);
   ```

4. Pass the resulting **Query** to the **RepositoryView** for execution:

   ```java
   items = view.executeQuery(femaleQuery);
   ```

**atg.repository.QueryOptions**

You can use the `atg.repository.QueryOptions` class to specify ways that a query can be modified. You can set the **QueryOptions** properties, and pass the **QueryOptions** bean to the following `executeQuery` method:

```java
RepositoryItem[] executeQuery(Query pQuery, QueryOptions pQueryOptions);
```

The **QueryOptions** properties let you limit the size of the result set, direct how the result set should be sorted, and precache specified properties:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startIndex</code></td>
<td>The index of the first element of a query result set that should actually be returned. By setting <code>startIndex</code> and <code>endingIndex</code>, you can limit the size of the query’s result set.</td>
</tr>
<tr>
<td><code>endingIndex</code></td>
<td>The items beginning with the <code>endingIndex</code> element of the query result set are not returned. In other words, the total number of items returned is <code>endingIndex - startIndex</code>. A value of -1 indicates that there is no limit to the number of items returned.</td>
</tr>
</tbody>
</table>
### Repository Query Examples

The examples in this section demonstrate how to perform some simple repository queries. In the Repository API, all queries are performed with `Query` or `QueryExpression` objects. A `QueryExpression`
μ is a building block you can use to create simple or complex queries. A Query is a repository query that can be executed. A Query can also be used as a building block to create queries that are more complicated.

The following example assumes an item descriptor named user with an integer property named userType. The query finds users whose userType property is set to 2:

```java
import atg.repository.*;

MutableRepository pRepository =
  (MutableRepository)ServletUtil.getCurrentRequest().resolveName
  ("/atg/userprofiling/ProfileAdapterRepository");

// Queries are created using QueryBuilders and executed by
// RepositoryViews. A Query is defined in the context of a
// specific item descriptor and thus must be built and executed with
// the right QueryBuilder and RepositoryView.
RepositoryItemDescriptor userDesc = pRepository.getItemDescriptor("user");
RepositoryView userView = userDesc.getRepositoryView();
QueryBuilder userBuilder = userView.getQueryBuilder();

// create a QueryExpression that represents the property userType
QueryExpression userType =
  userBuilder.createPropertyQueryExpression("userType");

// create a QueryExpression that represents the constant 2
QueryExpression two =
  userBuilder.createConstantQueryExpression(new Integer(2));

// now we build our query: userType = 2
Query userTypeIsTwo =
  userBuilder.createComparisonQuery(userType, two, QueryBuilder.EQUALS);

// finally, execute the query and get the results
RepositoryItem[] answer = userView.executeQuery(userTypeIsTwo);

System.out.println("running query: userType = 2");
if (answer == null)
{
  System.out.println("no items were found");
}
else
{
  for (int i=0; i<answer.length; i++)
    System.out.println("id: " + answer[i].getRepositoryId());
}
```

The preceding example can be expanded to include multiple selection criteria:
userType < 2 AND login STARTS WITH "j"

import atg.repository.*;

MutableRepository pRepository =
    (MutableRepository)ServletUtil.getCurrentRequest().resolveName
    ("/atg/userprofiling/ProfileAdapterRepository");

// reuse the building blocks we have to create
// the "userType < 2" query
Query userTypeLTTwo =
    userBuilder.createComparisonQuery(userType, two, QueryBuilder.LESS_THAN);

// create the "login STARTS WITH j" query
QueryExpression login =
    userBuilder.createPropertyQueryExpression("login");

QueryExpression j =
    userBuilder.createConstantQueryExpression("j");

// Note that we could make this query case-insensitive by adding another
// parameter to the createPatternMatchQuery, with a value of true
Query startsWithJ =
    userBuilder.createPatternMatchQuery(login, j, QueryBuilder.STARTS_WITH);

// now AND the two pieces together. You can AND together as many
// Query pieces as you like: we only have two in our example
Query[] pieces = { userTypeLTTwo, startsWithJ }
Query andQuery = userBuilder.createAndQuery(pieces);

// execute the query and get the results
answer = userView.executeQuery(andQuery);

System.out.println("running query: userType < 2 AND login STARTS WITH j");
if (answer == null) {
    System.out.println("no items were found");
} else {
    for (int i = 0; i < answer.length; i++)
    {
        RepositoryItem item = answer[i];
        String id = item.getRepositoryId();
        String l = (String)item.getPropertyValue("login");
        Integer a = (Integer)item.getPropertyValue("userType");
        System.out.println("item: " + id + ", login=" + l + ", userType=" + a);
    }
}
Repository Queries in the ATG Control Center

The ATG Control Center includes UI components for creating repositories queries. For example, the People and Organizations > Profile Repository window displays an expression editor for composing queries on the Profile repository like this:

List items of type User whose Gender is female

Note: ATG Control Center limits the number of items that can be returned by such queries. This limit is configurable and is set in the maxQueryCount property of /atg/devtools/RepositoryAgent. The default value is 1000.

Repository Query Language

ATG's Repository Query Language, or RQL, is a generic language for formulating queries that map to any repository implementation, such as SQL or LDAP. The repository connectors translate those queries into a syntax that the underlying data store understands.

You can use RQL in several different ways:

- Use RQL servlet beans in a JSP to perform repository queries: RQLQueryForEach and RQLQueryRange. For detailed information about these servlet beans, see the ATG Page Developer's Guide.
- Define an RQL filter that is implicitly applied to all queries performed by the repository. See Repository Filtering in the SQL Repository Queries chapter.
- Include RQL queries in <query-items> tags in the XML repository definition file. This is mainly useful for unit testing queries; it can also be used to preload repository caches. See Querying Items in the Developing and Testing an SQL Repository chapter and Preloading Caches in the SQL Repository Caching chapter.
- Use RQL directly by creating an atg.repository.rql.RqlStatement object. You can get a RqlQuery object from the RqlStatement object, then get an atg.repository.Query object from the RqlQuery object. This approach can be simpler than using a QueryBuilder implementation to create a Query object.

This section describes the details of RQL syntax and structure.
RQL Overview

RQL is a textual query syntax that is similar to SQL. It describes the set of conditions that must be matched by items of a particular item descriptor. The following is a simple RQL query that matches all items whose age property is greater than 30.

age > 30

Note: This RQL query omits the name of the item descriptor, which is usually implied by the context of the query’s use.

RQL supports all standard comparison operators and logical operators such as AND, OR, and NOT. For example:

age > 30 AND (lastName = 'jones' OR paymentOverdue = true)

RQL keywords are case-insensitive—for example, keywords NOT and not are equivalent.

Constants such as 30, true, or jones can represent numbers, boolean, or String values. String values are represented with Java syntax. They must be enclosed by quotes, and escape sequences for special characters or UNICODE characters must use Java escape syntax.

Properties such as age or lastName must be property names as they are defined in the repository. For example, a database might have a column named phone that is mapped to the repository property primaryPhoneNumber. An RQL query must use the repository’s property name primaryPhoneNumber.

age > 30 and (lastName = 'jones' or paymentOverdue = true)

RQL statements specify the conditions that an item must meet in order to be included in the result set. An RQL statement can also specify other directives to apply to the result set, such as ordering results and returning a portion of the result set.

Comparison Queries

Comparison queries are the simplest RQL queries, where a property’s value is compared against another property value, or against a constant. For example:

age > 30

All standard comparison operators can be used:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
</tr>
</tbody>
</table>
These operators can be applied to String properties and arguments, where case ordering is determined by lexical order of the Strings.

In general, these operators can only be used on properties that are scalar values. They should not be used on multi-valued properties.

**Text Comparison Queries**

Text comparison queries can be applied to String properties to determine if a portion or all of a property’s value matches a given comparison value. For example:

```rql
firstName STARTS WITH "h"
lastName ENDS WITH "son"
phoneNumber CONTAINS '33'
state EQUALS "Utah"
```

Be sure to enclose the comparison value in double quotes; otherwise, the RQL parser assumes the operand refers to a property name rather than a value.

By default, text comparison queries are case-sensitive. To perform a case-insensitive comparison, use the IGNORECASE directive. For example:

```rql
sports CONTAINS IGNORECASE "ball"
```

You can also make a negative text comparison query:

```rql
NOT firstName STARTS WITH IGNORECASE 'j'
```

**Note:** Negated pattern match queries can cause performance problems. Consider the queries you want to use and plan your database indexes accordingly to avoid table scans. STARTS WITH and EQUALS queries can be optimized easily with database indexes, while other pattern match queries generally cannot be. Case-insensitive pattern matching can also affect the ability of the database to optimize the query.

**Date and Timestamp Queries**

You can query on date and timestamp properties by using RQL’s `date` and `timestamp` functions. These functions create date literals, which let you create RQL queries that compare a date or timestamp string to date or timestamp property values. These functions use the following formats:

```rql
date("yyyy-MM-dd")
datetime("yyyy-MM-dd HH:mm:ss zzz")
```

For example:

```rql
submittedDate > date("2009-10-22")
submittedDate > datetime("2009-10-22 16:08:45 EDT")
```
Property of Property Queries

The queries shown earlier, as well as those described in the Full Text Search Queries section, can be applied to scalar properties. Some repositories support the use of properties that are themselves an item from another (or the same) item descriptor. For example, the address property might point to another item descriptor which itself has properties like city, state, and zip.

Queries can drill down through these properties with a dot notation. For example:

```
address.zip = "48322"
```

RQL allows for multiple levels of “property-of-property” expressions. For example:

```
department.manager.address.state.
```

Logical Operators

Query expressions can be combined with AND, OR, and NOT operators. Parentheses can be used to affect grouping. NOT has the highest precedence, AND the next highest precedence, and OR has the lowest precedence. For example, this expression:

```
name = "joe" OR NOT phone ENDS WITH "7" AND age > 30
```

is grouped as follows:

```
(name = "joe" OR ((NOT phone ENDS WITH "7") AND age > 30))
```

Multi-Valued Property Queries

Logical operators and the MATCH and MATCHES operators (described in the later section Full Text Search Queries) should only be applied to scalar properties. Another set of queries can be applied to arrays or collections of scalar values—for example, properties of type int[], or Set of Strings. You can query multi-valued properties with the following operators:

- INCLUDES
- INCLUDES ANY
- INCLUDES ALL (not valid for GSARespository queries)

The INCLUDES query matches items where the specified property includes the specified value. For example:

```
interests INCLUDES "biking"
```

The INCLUDES query can also match one of a set of items by including the ANY or ALL keyword, followed by a comma-separated set of items that are enclosed in braces. For example:

```
interests INCLUDES ANY { "biking", "swimming" }
```

This is equivalent to:
(interests INCLUDES "biking") OR (interests INCLUDES "swimming")

While this:

interests INCLUDES ALL { "biking", "swimming" }

is equivalent to:

(interests INCLUDES "biking") AND (interests INCLUDES "swimming")

**INCLUDES ITEM**

Some repositories support properties that point to multiple items of another (or the same) item descriptor. For example, the *addresses* property might point to an array of items, which themselves have address-related properties.

In this case, RQL allows for a subquery to be defined on these properties. For example:

`addresses INCLUDES ITEM (zip = "48322" AND state = "MI")`

This query means "find all people whose list of addresses includes at least one address whose zip code is 48322 and whose state is MI".

**IS NULL**

An IS NULL query can determine whether an expression evaluates to null. For example:

`phoneNumber IS NULL`

This expression evaluates to true if the `phoneNumber` is null.

**COUNT**

The COUNT operator can be used to query on the size of a multi-valued property. For example:

`COUNT (addresses) > 3`

This finds all people whose addresses property contains 4 or more elements.

**ALL**

An RQL query of ALL returns all items in a particular item descriptor. Use this query with care, because the result set can be very large. Usually this is combined with an ORDER BY or RANGE directive (described below). The RQL query is simply:

`ALL`
Full Text Search Queries

Some content repositories support the ability to perform full text searches. The formats of the text strings and other search directives vary from repository to repository. However, the basic query looks like this:

MATCH "mars"

This returns those items whose content matches "mars" in a full text search. (Content repositories allow parts of the item's data to be designated as "content" for the purposes of display and searching).

Another form of the query allows the full text search to proceed over a particular property:

firstName MATCHES 'abr'

Note that MATCH and MATCHES queries apply only to scalar properties.

Both forms of the query allow a special USING directive to pass special instructions to the underlying search engine. The format of this directive depends on the repository and whatever search engine it uses.

For example, to use the Oracle ConText full text search engine, the query looks like this:

firstName MATCHES 'abr' USING 'ORACLE_CONTEXT'

ID-based Queries

RQL can query items based on their repository IDs. This ability should be used with care, because repository IDs are not portable across repository implementations.

The first query searches for items that match a set of IDs. For example:

ID IN { "0002421", "0002219", "0003244" }

The next ID-based query applies only to content repositories, where items are organized into folders. This query restricts the search to only those items in the specified folders. The folders must be specified by ID:

IN FOLDERS { "10224", "10923", "12332" }

Note that passing in an empty or null set of IDs results in an exception.

Composite IDs can be specified in RQL queries with the following format for integers:

[value1, value2 ... valueN]

String IDs use the format:

["value1", "value2" ... "valueN"]

So a simple comparison query of a composite ID property might look like:

id = ["dept2", "emp345"]
Such a query returns an item with a composite repository ID of `dept2:emp345`. A query like this returns items with any of the IDs `dept2:emp345`, `dept2:emp346`, or `dept2:emp347`:

```
ID IN { ["dept2", "emp345"], ["dept2", "emp346"], ["dept2", "emp347"] }
```

**ORDER BY**

After a query is defined with the aforementioned query elements, the result is a set of items. The ORDER BY directive orders the results by item properties. For example:

```
age > 30 ORDER BY firstName
```

This query returns a result set where items are ordered by the `firstName` property in ascending order—the default. Results can also be ordered in descending order by adding `SORT DESC` to the end of the directive:

```
age > 30 ORDER BY firstName SORT DESC
```

Results can be ordered by multiple properties. For example:

```
age > 30 ORDER BY lastName, firstName SORT DESC
```

This orders results by `lastName`. If multiple results have the same `lastName`, within their group they are ordered by `firstName` in descending order.

A further directive, `CASE IGNORECASE`, can specify case-insensitive sorting:

```
age > 30 ORDER BY firstName SORT DESC CASE IGNORECASE
```

Note that you can omit the tokens `SORT` and `CASE`, unless you use parameters for the ASC/DESC or USECASE/IGNORECASE tokens.

**RANGE**

Many queries have the potential for returning large result sets. Most applications do not want to display the entire result set—they might want to display just the first 10 results. Or they might want to page through the results, showing results 0-9, then results 10-19, and so on.

The RANGE directive is used to specify this in the RQL query. The RANGE directive must come after the ORDER BY directive (if any). It has three forms. The first is the most common:

```
age > 30 RANGE +10
```

This causes only the first 10 results to be returned. If the result set is already less than 10, all results are returned.

The next form of the RANGE directive allows the results to start at a specified index:

```
age > 30 RANGE 10+
```
This causes the first 10 results to be skipped, and the remaining results to be returned.

The final form of the RANGE directive combines the above two forms, and is often used for paging:

\[ \text{age} > 30 \text{ RANGE } 40+10 \]

This skips the first 40 results, then returns up to the next 10 results.

**Parameters in Queries**

In all of the previous examples, the queries contain hard-coded constants, such as 30 or joe. Most typically, the actual values used in the query are unknown when the RQL statement is written. In this case, the values can be substituted with parameter expressions. For example:

\[ \text{age} > \text{?0} \text{ AND firstName CONTAINS } \text{?1} \text{ RANGE } \text{?2}+10 \]

Every \(?\{\text{number}\}\) represents a parameterized value that is supplied on query execution. How those values are supplied depends on the application performing the query. In the case of entity EJBs, where RQL queries are used to represent finder methods, the parameters are filled in from the arguments of the finder methods. For example, \(?0\) is substituted with the value of the first argument, \(?1\) with the second, and so on.

Parameter expressions can generally be used wherever constant values are used, including in RANGE expressions. However, parameter expressions cannot be used in array expressions, such as ID IN or IN FOLDERS queries. Also, parameter expressions cannot be used as substitutes for property names; all property names must be hard-coded into the RQL query when it is written.

**Parameterized Field Queries**

When a parameterized query is used, each numbered placeholder is substituted with the value of an entire object at runtime. However, it is sometimes more useful to substitute in the value of one of the object's fields, rather than the entire value of the object. A parameterized field query specifies this with the syntax \(?\{\text{number}\}.\{\text{fieldName}\}\). For example:

\[ \text{name} = \text{?0.name} \text{ AND age} = \text{?0.age} \]

In this example, only one object is passed into the query at runtime. However, this object is expected to have two public member variables called name and age. The query extracts the values of these member variables from the object and substitute those values for the \(?0.\text{name}\) and \(?0.\text{age}\) parameters. Note that the fields must be public member variables of the object that is passed in, not JavaBean properties. For example, the following object can be passed in to the query:

```java
public class QuerySpecifier {
    public String name;
    public int age;
}
```

Parameterized Field Queries are used most often for entity EJBs, which allow primary key classes to contain multiple fields. In this case, only one object is passed to the query (the primary key), but if the
primary key spans multiple database fields, the primary key object contains the values of those fields in its public member variables.

**RQL Examples**

The following example shows how you might use a parameter expression in Java code. It creates an RQL statement and uses it in executing a query to find person repository items where the value of the age property is greater than 23.

```java
RepositoryView view = repository.getView("person");
RqlStatement statement = RqlStatement.parseRqlStatement("age > ?0");
Object params[] = new Object[1];
params[0] = new Integer(23);
RepositoryItem[] items = statement.executeQuery(view, params);
```

Here is another example that demonstrates a text comparison query:

```java
RqlStatement statement = RqlStatement.parseRqlStatement("lastName STARTS WITH ?0");
Object params[] = {new String("m")};
items = statement.executeQuery(view, params);
```

Note how in the text comparison queries the comparison value "m" is enclosed in double quotes; otherwise, the RQL parser assumes the comparison term refers to a property name rather than a property value.

**RQL Grammar**

The following is a formal definition of the RQL grammar:

```
RQLStatement:: Query OrderByClause RangeClause
Query:: OR | AND | NOT | Comparison | ID IN | IN FOLDERS | ALL | TextSearch | PropertyTextSearch | INCLUDES ITEM | IS NULL | (Query)
```

The precedence order of the queries from highest to lowest is as follows:

- (Query)
- Comparison, ID IN, IN FOLDERS, ALL, TextSearch, PropertyTextSearch, INCLUDES ITEM, IS NULL
- NOT
- AND
- OR
- OR:: Query OR Query ...
- AND:: Query AND Query ...
- NOT:: NOT Query
- Comparison:: Expression ComparisonOperator Expression
- ComparisonOperator:: = | != | < | <= | > | >= | INCLUDES ANY | INCLUDES ALL |
- INCLUES | STARTS WITH [IGNORECASE] | ENDS WITH [IGNORECASE] | CONTAINS [IGNORECASE]
- IdIn:: ID IN StringArray
- InFolders:: IN FOLDERS StringArray
- ALL:: ALL
- TextSearch:: MATCH StringLiteral [USING StringLiteral]
- PropertyTextSearch:: ObjectExpression MATCHES StringLiteral [USING StringLiteral]
- IncludesItem:: Expression INCLUDES ITEM ( Query )
- Expression:: CountExpression | ObjectExpression | ParameterExpression | ConstantExpression
- CountExpression:: COUNT ( ObjectExpression | ParameterExpression | ConstantExpression )
- ObjectExpression:: PropertyName | ObjectExpression.PropertyName | ObjectExpression[Expression]
- PropertyName:: <Java identifier>
- ParameterExpression:: ?<Parameter number>[.<Field name>]
- ConstantExpression:: StringLiteral | IntegerLiteral | DoubleLiteral | BooleanLiteral | ArrayLiteral
- StringLiteral:: "<Java string literal>"
  The string literal uses the Java format, including escape characters (including octal and Unicode), and must be enclosed in double quotes.
  - IntegerLiteral:: <Java integer literal>
  - DoubleLiteral:: <Java double literal>
  - BooleanLiteral:: true | false
  - ArrayLiteral:: [ ConstantExpression, ... ]
  - StringArray:: [ StringLiteral, ... ]
  - OrderByClause:: ORDER BY PropertyName [[SORT] [ ASC | DESC ]] [CASE [ IGNORECASE | USECASE ]]  The SORT ASC/DESC directives are optional and default to SORT ASC. The CASE IGNORECASE/USECASE directives are optional and default to CASE USECASE.
  - RangeClause:: RANGE <Starting Index> + <Count>
4 SQL Repository Overview

The ATG SQL repository can be used to connect ATG applications to an SQL database. An SQL database provides fast, scalable storage and retrieval of persistent information. The SQL repository works with an SQL database to store objects and make those objects visible inside an ATG application as Dynamic Beans. The uses of an SQL repository can be as varied as the uses of a relational database.

The ATG platform includes SQL repositories that store:

- User profiles (the Personalization module’s SQL Profile Repository). See the SQL Profile Repositories chapter in the ATG Personalization Programming Guide.
- Web site content (the SQL content repository). See this chapter and the SQL Content Repositories chapter.
- Security profiles used by the Administrative Security system. See the Managing Access Control chapter of the ATG Programming Guide.

In addition, an ATG Commerce site uses repositories that store:

- Store catalog
- In-process orders
- Inventory
- Gift lists and wish lists
- Pricing and promotions

Refer to the ATG Commerce Programming Guide for information about these repositories.

The ATG platform includes a component at /atg/registry/ContentRepositories that instantiates the class atg.repository.nucleus.RepositoryRegistryService, which maintains a list of all registered SQL content repositories.

Repository setup steps

You set up an SQL repository on the ATG platform in the following steps:

1. Create the repository definition file to be used by the SQL repository.

   This template is an XML file that defines repository item descriptors and their attributes, and describes the relationship of your SQL repository to the SQL database. While the SQL repository can represent a variety of data models, it cannot easily represent any arbitrary data model. Thus, it is usually a good idea to design the SQL repository schema before you design your SQL database schema.
The SQL Repository Data Models and SQL Repository Item Properties chapters describe how to design item descriptors and other SQL repository elements. See also the SQL Repository Definition Tag Reference for details about the XML tags used to create the SQL repository definition file.

2. Configure an SQL Repository component.
   This component’s definitionFiles property points to the repository definition file. For detailed information, see Configuring the SQL Repository Component.

3. Create the SQL database schema on your SQL database server.
   You can use the startSQLRepository script with the -outputSQL option to generate a preliminary form of the SQL needed to create the database schema, then edit the output to optimize the database schema.
5 SQL Repository Architecture

The SQL repository is a generalized and flexible implementation of the ATG Repository API that an application can use to access data stored in an SQL database. See the Repository API chapter for more information. The SQL repository is implemented through the \texttt{atg.adapter.gsa} package (GSA stands for Generic SQL Adapter).

The main ATG component in the SQL repository is an instance of the \texttt{atg.adapter.gsa.GSARepository} class, which extends the class \texttt{atg.repository.RepositoryImpl} and implements two interfaces:

- \texttt{atg.repository.MutableRepository}
- \texttt{atg.repository.content.ContentRepository}

You create an SQL repository instance by instantiating the \texttt{atg.adapter.gsa.GSARepository} class. This class is not documented in the \texttt{ATG API Reference}, and it is not intended that you access this class directly from Java code. Normally, you access all Repository functionality with the interfaces \texttt{atg.repository.Repository} and \texttt{atg.repository.MutableRepository}. This enables your classes to work with any repository implementation for the greatest flexibility. Some methods like those for cache invalidation are defined on the class \texttt{atg.repository.RepositoryImpl}. It is anticipated that future repositories will extend that class, so you can make your code more reusable and maintainable by accessing those methods on this base class rather than the implementation class of \texttt{atg.adapter.gsa.GSARepository}.

The SQL repository uses an XML repository definition file to describe the item descriptors that compose a repository. The repository definition file also describes the relationships between repository definitions—item descriptors, repository items, and repository item properties, and the corresponding elements of an SQL database—its tables, rows, and columns, respectively. The XML tags of the repository definition file are described in detail in the \texttt{SQL Repository Reference} chapter. The XML tags are also introduced in examples in this chapter and in the \texttt{SQL Repository Item Properties} chapter.

Repositories and Transactions

All SQL repository operations are performed with the current JTA transaction, if one exists. For example, when an application calls the \texttt{Repository.updateItem()} methods, for example, changes are immediately visible only to subsequent \texttt{getItem()} calls that are made in that transaction. When the JTA transaction is committed, the repository item changes are committed to the database.
If you do not have a JTA transaction in place, each SQL repository operation that affects the state of a repository item creates and commits a transaction around the operation. Thus, a `setPropertyVal`ue call by itself with no JTA transaction in place is committed to the database when the call returns.

Here are two examples:

If no transaction exists:

1. Begin JTA transaction.
2. Call `setPropertyVal`ue.
3. Commit JTA transaction. At this point, SQL is issued and the changes are committed.

Using the `updateItem` method:

1. Begin JTA transaction.
2. Call `setPropertyVal`ue.
3. Call `updateItem`. At this point, SQL is issued.
4. Commit JTA transaction. Changes are committed.

Generally, you want to call `updateItem` explicitly. This ensures that if you perform any queries between the change made in the `setPropertyVal`ue call and the commitment of the transaction, those queries have the new property value available to them.

**Distributed cache invalidation**

You can configure the ATG platform to send repository item cache invalidation messages to other remote ATG servers. If you set an item descriptor to one of several distributed caching modes, a cache invalidation message is sent to other ATG servers. You can also set an item descriptor to locked caching mode; when a server gives up ownership of the lock, it also invalidates the cache. For more information see the SQL Repository Caching chapter.

**Transaction isolation**

The SQL repository implements transaction isolation. The first time an item is accessed in a transaction—through `getItem()`, or the first attempt to call `getPropertyVal`ue on an item that was retrieved in a different transaction—it is guaranteed to be up to date at that time. If another transaction changes the item while this transaction is in progress, those changes are visible only to a new transaction.

**Managing Transactions**

By default, a transaction is created and committed for each method call. This is generally not the most efficient way to handle repository item updates. It is generally more efficient if all method calls that create or update a repository item are performed in a single transaction.

The ATG platform provides several transaction demarcation techniques for grouping repository method calls into a single transaction.
Use the Transaction servlet bean

This servlet bean, described in the ATG Page Developer’s Guide, explicitly creates a transaction on a page. For example, the following uses the current transaction, if any exists. If there is no current transaction, one is created before calling the output open parameter, and committed at the end of the droplet:

```xml
<droplet bean="/atg/dynamo/transaction/droplet/Transaction">
    <param name="transAttribute" value="required">
    <oparam name="output">
        ... do repository item work ...
    </oparam>
</droplet>
```

Use JTA

JTA (Java Transaction API) lets you explicitly manage the transaction. For example, you might explicitly create a transaction around a repository item creation or update like this:

```java
TransactionManager tm = ... TransactionDemarcation td = new TransactionDemarcation ();
try {
    try {
        td.begin (tm);
        ... do repository item work ...
    } finally {
        td.end ();
    }
} catch (TransactionDemarcationException exc) {
    ... handle the exception ...
}
```

Use a FormHandler

If you are writing a FormHandler component, you can simply extend the class `atg.droplet.TransactionalFormHandler`. This FormHandler automatically wraps a transaction around all property get method calls made from a page and another transaction around all property set or handle method calls made from a page. See the Working with Forms and Form Handlers chapter of the ATG Programming Guide for more information.

For detailed information about managing transactions, see the Transaction Management chapter in the ATG Programming Guide.
Repository Definition Files

Each repository can be defined with one or more XML repository definition files. The repository definition files used by a repository are specified by the definitionFiles property of the Repository component. The value of the definitionFiles property is the absolute name of an XML file in the application's configuration path.

If more than one XML file is defined with the same path in different configuration path directories, they are combined using the XML combination rules described in the Nucleus: Organizing JavaBean Components chapter of the ATG Programming Guide. This lets you modify one XML file by adding or removing item descriptors, properties, and tables in another layer of the configuration path.

XML file combination matches the item-descriptor, table, and property tags by name, operating from the outside tag inward. The changes that you make must exactly match the item descriptor, table, and property that you want to modify.

The following example file modifies the repository definition for the Profile repository, located at /atg/userprofiling/userProfile.xml:

```xml
<gsa-template>
  <item-descriptor name="user" cache-mode="locked" item-cache-size="500">
    <table name="dps_user">
      <property name="userType" data-type="enumerated">
        <option value="investor" code="1">
          <option value="broker" code="2">
            <option value="guest" code="3">
          </option>
        </option>
      </property>
    </table>
  </item-descriptor>
</gsa-template>
```

This modifies the standard repository definition as follows:

- Sets the user item descriptor's cache-mode to locked.
- Changes the data-type attribute of the userType property from int to enumerated, and sets several options for its value.
- Sets the item-cache-size to 500 (the default setting is 1000).

**Default Values and XML File Combination**

If a repository's DTD provides a default setting for an XML element, that default setting applies to all XML files that do not explicitly set the element. For example, the SQL Repository DTD sets the property element's expert attribute to false:

```xml
expert %flag; 'false'
```
If a base SQL repository definition file sets a property's `expert` attribute to true, and supplemental SQL repository definition files—that is, files that are later in the configuration path—modify that property, the supplemental files must also set the `expert` attribute to true; otherwise, the attribute's value reverts to the DTD's default setting of `false`.

**SQL Repository Items**

Repository items are Dynamic Beans that implement the `RepositoryItem` interface. Because they are registered as Dynamic Beans, RepositoryItems do not need to define `setX` and `getX` methods for each property. The properties of RepositoryItems are defined at application startup when the XML file that defines the repository template is parsed.

Each repository item has a unique repository ID. The ATG IdGenerator generates unique repository IDs and should be used instead of sequences generated directly by the database system. See the Core Dynamo Services chapter of the ATG Programming Guide.

A repository organizes repository items into types that have the same set of properties. Each item type is defined by an item descriptor.

**SQL Repository Item Descriptors**

An SQL repository can define multiple named item types. Each item type is defined by an item descriptor. You can define different kinds of objects with different item descriptors, and ATG manages them in a single repository. Each named type corresponds to an item descriptor, and each item descriptor corresponds to a `RepositoryView` of the same name.

For example, a simple database might have two kinds of entities, `book` and `author`, where a book has one author, and an author has zero or more books. This is depicted in the figure below.

![Diagram showing `book` and `author` relationships](image)

DAF lets you represent this in a single repository using two independent types named `book` and `author`, each type defined by its own item descriptor. These item types are independent in that they do not share any properties in their item descriptors. They might both have properties such as name or weight, but these properties are independently defined. Another way to look at it is that they each have their own item descriptor.

The SQL repository also supports a simplified form of inheritance for item descriptors. See Item Descriptor Inheritance in the SQL Repository Data Models chapter.
6 SQL Repository Data Models

Repository items correspond to business objects, like customers, and elements of business objects, like a customer's shipping address. An item descriptor in an SQL repository defines a repository item type. It specifies repository item properties and the database tables and columns that store the data of those properties. This chapter describes how to define item descriptors and how to represent the relationships between item descriptors in an SQL repository definition.

Note that an SQL repository cannot necessarily work with any arbitrary SQL database model. The basic data model patterns are described in the Sample SQL Repository Definition Files section of the SQL Repository Reference chapter.

Primary and Auxiliary Tables

Each item descriptor must specify one primary table. The primary table is specified with the type="primary" XML attribute in a <table> tag. The <table> tag for the primary table sets its id-column-names attribute to the columns that store the repository ID. For example:

```
<table name="user" type="primary" id-column-names="id">
  properties...
</table>
```

id Property

In order to obtain the data of a repository item from a datastore, you must supply the primary table's ID. The repository item descriptor can map to the table's ID column implicitly through the <table> tag id-column-names attribute; or it can explicitly map to the ID column through a <property> tag that is set as follows:

```
<property name="id" column-name="table-id-column"/>
```

For example:

```
<Item-descriptor name="user">
  <table name="user" type="primary" id-column-names="emp_id">
    <property name="id" column-name="emp_id"/>
  </table>
</Item-descriptor>
```
If the item descriptor does not explicitly define an id property, its data type cannot be explicitly set; and it uses the default string data type. In this case, the following constraints apply:

- You cannot query against repository item IDs.
- You cannot access the item’s ID by calling getPropertyValue(); it is, however, accessible through getRepositoryId().

However, if you explicitly define the id property in the item descriptor with a <property> tag, you can query repository items by their ID and you can explicitly set the id property’s data type. The columns specified by the id-column-names attribute are not required to use the same data type; a composite repository ID can be composed of strings, integers, and longs.

**Note:** You cannot change a repository item’s ID after it is saved to the database.

**Constraints**

Avoid using the following special characters in repository IDs:

<table>
<thead>
<tr>
<th>Characters used in:</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>URLs</td>
<td>+ (plus)</td>
</tr>
<tr>
<td></td>
<td>/ (forward slash)</td>
</tr>
<tr>
<td></td>
<td>? (question mark)</td>
</tr>
<tr>
<td></td>
<td>% (percent sign)</td>
</tr>
<tr>
<td></td>
<td># (hash)</td>
</tr>
<tr>
<td></td>
<td>$ (ampersand)</td>
</tr>
<tr>
<td>XML</td>
<td>&lt; (left angle bracket)</td>
</tr>
<tr>
<td></td>
<td>&gt; (right angle bracket)</td>
</tr>
<tr>
<td></td>
<td>&quot; (double quote)</td>
</tr>
<tr>
<td></td>
<td>’ (single quote)</td>
</tr>
<tr>
<td></td>
<td>$ (ampersand)</td>
</tr>
</tbody>
</table>

**Compound Repository IDs**

A repository ID can span multiple database columns, where the data type of each column corresponds to the Java type String, Integer, or Long. An item descriptor can define one property that combines the data from all database ID columns, or define a property for each database ID column. The two examples that follow show how two database ID columns, folder_id and doc_id, can be specified by one item descriptor property, and by two properties.

Here, the ID property combines the data from the two database ID columns:
Here, two properties, folder and document, are defined to represent both database ID columns:

```xml
<table name="doc" type="primary" id-column-names="folder_id,doc_id">
  <property name="folder" column-names="folder_id" data-type="string"/>
  <property name="document" column-names="doc_id" data-type="int"/>
</table>
```

### Concatenation of Multi-Column IDs

By default, a multi-column ID is encoded as a String that concatenates ID elements in the order specified by the item descriptor’s `id-column-names` attribute. Each element is separated by a separator character—by default, colon (:).

The item descriptor’s `id-separator` attribute can specify a different separator character. For example, an item descriptor might define its ID separator character as an asterisk (*):

```xml
<item-descriptor name="employee" id-separator="*">
  <table name="user" type="primary" id-column-names="dept_id,emp_id">
    properties...
  </table>
</item-descriptor>
```

In this case, the repository ID for a `user` item might look like this:

```
sales*bbanzai
```

The following constraints apply to multi-column ID separator characters:

- Do not use brackets or comma as separator character. These characters are used by RQL and the SQL repository when specifying lists of IDs.
- Repository IDs in both columns must exclude the ID separator—by default, colon (:).

### Accessing Items with Compound Repository IDs

In order to retrieve or remove an item that uses a compound repository ID, you can supply the concatenated string ID as a parameter to `atg.adapter.gsa.GSARespository` methods such as `getItems()`.

For example, given the following item descriptor:
<item-descriptor name="employees">
  <table name="employee_table" type="primary" id-column-names="dept_id,emp_id">
    <property name="id" column-names="dept_id,emp_id" data-types="string,int"/>
  </table>
</item-descriptor>

You might obtain an employee repository item as follows:

```
RepositoryItem employee = rep.getItems("IS:100002", "employee");
```

**IdSpaces and the id Property**

IDs for repository items are requested from the appropriate IdSpace for the repository item. The id-space-names attribute in the primary table of an item descriptor specifies which IdSpaces supply repository IDs for items of that item type. For item types with single-column IDs, the default name for the IdSpace is the item descriptor name. For item types with multi-column IDs, the default name for the IdSpace is derived from the primary table name and ID column:

```
primary-table-name.id-column-names
```

For example, in an item descriptor defined like this:

```
<item-descriptor name="user">
  <table name="users" type="primary" id-column-names="id">
    properties...
  </table>
</item-descriptor>
```

the default IdSpace is named user. In an item descriptor with a composite repository ID defined like this:

```
<table name="user" type="primary" id-column-names="dept_id,emp_id">
    properties...
</table>
```

the default IdSpaces is named user.dept_id and user.emp_id. In any case, you can override the default IdSpace names with the id-space-names attribute in the item descriptor definition:

```
<table name="user" type="primary" id-column-names="dept_id,emp_id"
    id-space-names="DepartmentId,EmployeeId">
    properties...
</table>
```

See the Core Dynamo Services chapter in the *ATG Programming Guide* for more information about ID space names and how they affect the IDs of newly generated items.
Database Sequences and Repository IDs

Tables in a relational database must have a primary key. When designing a database, the primary key can often be chosen from intrinsic data. For example, a person's social security number or the hardware address of a network interface card are unique identifiers that can be good choices for a repository ID.

Sometimes, there is no natural ID and you must generate one to serve as the primary key. Typically an integer counter is used for this. The major relational database management system vendors have facilities to automatically generate IDs internally. These IDs (called sequences in some systems) differ from each other in how they are generated and retrieved. Database-generated sequences are not supported as repository IDs in the ATG platform. Instead, use an ID generated by the IdGenerator, as described above and in the Core Dynamo Services chapter of the ATG Programming Guide.

Auxiliary Tables

You can handle some data relationships with auxiliary attribute tables. For example, you can store users and their addresses in two related database tables, as described in the following piece of an XML repository definition:

```xml
<item-descriptor name="user">
    <table name="dps_user" type="primary" id-column-names="id">
        <property name="login" data-type="string"/>
    </table>
    <table name="dps_address" type="auxiliary" id-column-names="id">
        <property name="address1"/>
        <property name="city"/>
        <property name="state"/>
        <property name="zip"/>
    </table>
</item-descriptor>
```

Each user has a single address. For the purposes of this example, the user information is stored in a separate table from the user's address information.

Note that if you use auxiliary tables, each table definition, whether primary or not, must define an id-column-names attribute. This attribute defines the column names in the table that represent the repository ID. This indicates how to join auxiliary tables to the primary table. The columns in the id-column-names attribute must be listed in the same order as they are in the id-column-names attribute of the primary table.

References Constraints

In general, auxiliary and multi tables should not have REFERENCES constraints that point to each other. Instead, each of these tables can have a REFERENCES constraint that points to the primary table for the repository item. This limitation exists because the SQL repository processes insert and delete statements...
for auxiliary and multi tables in the same order. As a result, if you specify REFERENCES constraints between an auxiliary and a multi table or vice versa, a constraint error results on the insert or the delete.

## Properties and Database Columns

All repository item properties are described in the XML repository definition with `<property>` tags. You can explicitly map an item property to a database column name through the `column-names` attribute; otherwise, the property name and database column name are assumed to be the same. For example, you can map the `login` property to the `login_name` property as follows:

```
<property name="login" column-names="login_name" data-types="string"/>
```

The `column-names` attribute of a property specifies the database column that stores the property. Each property must also define its data type with the `data-types` attribute. In the case of a multi-column id property, the `data-types` attribute is a comma-separated list of data types, where each entry corresponds to an entry in the `column-names` attribute. Each column can have a different data type. If no data type is specified, the string data type is used by default. The valid data type names and their Java and SQL representations are listed in the Data Type Mappings: Java and SQL section in the SQL Repository Reference chapter.

## One-to-Many Relationships: Multi-Valued Properties

The SQL repository supports one-to-many relationships between two tables, and does not interpret the results according to any specific paradigm. This allows your application to apply whatever meaning you want to one-to-many relationships.

The SQL repository implements one-to-many relationships as multi-valued properties. To implement a multi-valued property, define the property with several `<table>` and `<property>` attributes:

### `<table>` attributes

The `<table>` tag for a multi-valued property must set the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Set to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>multi</td>
</tr>
</tbody>
</table>

For example:

```
<table name="..." type="multi" ...>
```
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Set to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi-column-name</td>
<td>The appropriate table column name. For example:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;table name=&quot;...&quot; type=&quot;multi&quot; multi-column-name=&quot;idx&quot; ...</code></td>
</tr>
<tr>
<td></td>
<td>The <code>multi-column-name</code> attribute ensures that the ordering of the multi-values are maintained. The column specified by the <code>multi-column-name</code> attribute is used for multi-valued properties of data type array, map, and list and is not used for sets (which are unordered). For map type properties, the values in the column specified by the <code>multi-column-name</code> attribute must be a string. For list or array type properties, these values should be an integer or numeric type, and must be sequential.</td>
</tr>
<tr>
<td>id-column-names</td>
<td>The appropriate table column names.</td>
</tr>
<tr>
<td></td>
<td>As with auxiliary tables, the ordering of the ID column names is important. The columns specified by this attribute must list table columns in the same order as the <code>id-column-names</code> attribute of the primary table.</td>
</tr>
</tbody>
</table>

**<property> attributes**

The `<property>` tag for a multi-valued property sets the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Set to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-type</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td>array</td>
</tr>
<tr>
<td></td>
<td>set</td>
</tr>
<tr>
<td></td>
<td>map</td>
</tr>
<tr>
<td></td>
<td>list</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;property ... data-type=&quot;array&quot; ...</code></td>
</tr>
<tr>
<td>component-data-type</td>
<td>A primitive data type such as int and string, or the name of a user-defined property type (see User-Defined Property Types). For example:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;property name=&quot;interests&quot; column-name=&quot;interest&quot; data-type=&quot;array&quot; component-data-type=&quot;string&quot;/&gt;</code></td>
</tr>
<tr>
<td></td>
<td>Note that the SQL repository does not support references to binary types.</td>
</tr>
<tr>
<td>component-item-type</td>
<td>The item descriptor name of the referenced repository items. For example:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;property name=&quot;...&quot; column-name=&quot;designers&quot; data-type=&quot;array&quot; component-item-type=&quot;user&quot;/&gt;</code></td>
</tr>
</tbody>
</table>
**Note:** You cannot establish a default value for multi-valued properties.

The following example shows how the XML repository definition might specify the multi-valued property interests:

```xml
<item-descriptor name="user">
  <table name="dps_user" id-column-names="id" type="primary">
    <property name="login" data-type="string"/>
  </table>
  <table name="dps_interest" type="multi" id-column-names="id"
         multi-column-name="idx">
    <property name="interests" column-name="interest" data-type="array"
                component-data-type="string"/>
  </table>
</item-descriptor>
```

See also the Sample SQL Repository Definition Files section in the SQL Repository Reference chapter for more examples of one-to-many relationships in repository definitions.

**Allow null values**

By default, null values are not allowed in multi-valued properties. You can specify to allow null values at two levels:

- Enable all multi-valued properties in a repository to accept null values by setting the repository property `allowNullValues` to `true`.
- Allow null values for an individual property by setting its `<property>` tag attribute `allowNullValues` to `true`.

### Operating on Multi-Valued Properties

When you operate on the returned value from a `List`, `Set`, or `Map` property, do not rely on the concrete implementation of this class. You should not serialize this value, or use it directly in the `setPropertyValue` call for another `List`, `Set`, or `Map` property. Instead, you can copy these values into another List that you create and use that value. For example:

```java
List ls = (List) item.getPropertyValue("someListProperty");
ArrayList toUseElsewhere = new ArrayList();
toUseElsewhere.addAll(ls);
```

Now you can use `toUseElsewhere` in a `writeObject` call or in another `setPropertyValue` call.
Many-to-Many Relationships

You can represent many-to-many data relationships in an SQL repository. For example, an author may have written multiple books, and a book may have multiple authors. Representing this kind of relationship depends on the `type="multi"` attribute in a `<table>` tag. You can represent a many-to-many relationship with two one-to-many relationships that point to the same intermediate table. The following example represents a many-to-many relationship between the authors of a book and the books written by an author:

```xml
<item-descriptor name="author">
  <table type="primary" name="author">
    ...
  </table>
  <table type="multi" name="author_book" id-column-names="author_id">
    <property name="booksWritten" column-name="book_id" data-type="set" component-item-type="book"/>
  </table>
</item-descriptor>

<item-descriptor name="book">
  <table type="primary" name="book">
    ...
  </table>
  <table type="multi" name="author_book" id-column-names="book_id">
    <property name="authors" column-name="author_id" data-type="set" component-item-type="author"/>
  </table>
</item-descriptor>
```

This example uses three tables:

- `author` items use the primary database table `author`.
- `book` items use the primary database table `book`.
- `author` and `book` items both use the intermediate multi table `author_book` to handle the relationship between authors and books.

The data type of the properties in the intermediate multi table must be Set, not array, map or list.

Tables can have columns other than the ones referenced in the repository definition file, provided two conditions are true:

- The columns allow null values.
• There is no design requirement that the repository recognize the existence of such columns.

**Default Item Descriptor**

You can identify a repository's default item descriptor, by setting its `<item-descriptor>` default attribute to `true`. Each repository can have a single default item descriptor. If a repository has only one item descriptor definition, it is the default. If no default item descriptor is explicitly identified, the first item descriptor in the XML file is the default.

When you use Repository methods such as `getItem`, `createItem` without specifying an item descriptor, you use the default item descriptor. These methods are not recommended for most applications, unless there is only one item type in the repository.

**Cascading Data Relationships**

The SQL repository uses the `cascade` attribute in a `<property>` tag to better handle hierarchical properties—that is, properties that have the attribute `item-type` or `component-item-type`. The `cascade` attribute can be set to one or more of these values:

- `insert`
- `update`
- `delete`

For example:

```
<property name="scenarios" item-type="scenario" cascade="update,delete"/>
```

**Cascade Insert**

When you create a repository item that contains a property with the `item-type` attribute and the property's `cascade` attribute is set to `insert`, the following actions occur:

- An item of the type declared by the `item-type` attribute is also created.
- The property is set to point to the other item created.

`insert` is typically used together with `update` and `delete`, so the referenced item is automatically created, updated, and deleted with the parent item.

**Note:** `insert` is ignored for properties that use the attribute `component-item-type`. 
Cascade Update

If a repository item property references other items and the property’s cascade attribute is set to update, the following actions occur:

- When you call addItem(), any new (transient) items referenced by this property are added automatically to the repository.
- When you call updateItem(), any modified referenced items are automatically updated. Any referenced items that are new (transient) items are added.

Cascade Delete

If a repository item property references other items and the property’s cascade attribute is set to delete, removal of the repository item triggers removal of the referenced items. Also, when you remove a reference to this item, the item is automatically removed.

You should exercise caution when using cascading deletion in one-to-many relationships. Specifically, never set cascade to delete in properties on the “many” side of the relationship where those properties refer to items on the “one” side of the relationship. The item on the “one” side of the relationship cannot be deleted safely, because multiple items may be referring to it.

For example, an item descriptor company has an employee property that references many repository items defined by an employee item descriptor. The employee item descriptor itself defines a company property. In this one-to-many relationship, the employee property in the company item descriptor can set cascade to delete. However, the company property in the employee item descriptor must not set its own cascade attribute to delete; otherwise, the removal of one employee item would also entail removal of the company item that references all other employee items.

Removing null references

It sometimes happens that an item property references other items and one of those items is removed without explicitly removing the reference to it. This can occur when the database has no references constraint on the pertinent columns, so the referenced item can be removed without updating the referencing item’s property. It can also occur in a case where the referenced item might actually exist, but is currently filtered out by an RQL filter so it appears not to exist.

A repository item ignores references to missing items rather than returning an error if the referencing property sets its removeNullValues attribute to true. In this case, the missing item is returned as null to a scalar reference, and is omitted from the items that are returned for multi-valued references.

For example, a user profile might have a multi-valued property that is a list of favorite articles. Any given article might be deleted or become out of date. You can remove references to articles that are no longer available with the removeNullValues attribute like this:

```xml
<property name="favoriteArticles" data-type="list" component-item-type="articles">
  <attribute name="removeNullValues" value="true"/>
</property>
```
Order of deletion

Depending on how the database schema defines reference constraints, you might need to control whether a cascading deletion occurs before or after deletion of the referencing item. You can specify the desired behavior in an item descriptor by setting the `cascadeDeleteOrder` attribute:

```
<item-descriptor name="biographies" ...>
  <attribute name="cascadeDeleteOrder" value="last"/>
  <table name="...">
    <property name="publisher" item-type="publisher" cascade="delete"/>
    <property name="...">
  </table>
</item-descriptor>
```

You can set `cascadeDeleteOrder` to one of these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>afterAuxiliaryBeforePrimary</code></td>
<td>The default setting: cascading deletion is performed after deleting auxiliary multi-table rows, but before deleting the primary table row. This is the default behavior.</td>
</tr>
<tr>
<td><code>first</code></td>
<td>Cascading deletion is performed before any deletions in tables of this item.</td>
</tr>
<tr>
<td><code>last</code></td>
<td>Cascading deletion is performed after all deletions in tables of this item.</td>
</tr>
</tbody>
</table>

Cascade Example

The following item descriptors define two item types: `author` and `address`. The `author` item type references the address item type as follows:

- author defines an address property, which sets the attribute `item-type` to address.
- The address property sets its `cascade` attribute to `insert, update and delete`.

```
<item-descriptor name="author">
  <table name="author" id-column-names="author_id" type="primary">
    <property name="name"/>
    <property name="address" item-type="address" cascade="insert, update, delete"/>
  </table>
</item-descriptor>
```

<item-descriptor name="address" item-type -->

---

64 - SQL Repository Data Models
Given these definitions, whenever an author type repository item is created, added, updated, or deleted, the same actions apply to the corresponding address repository item.

**Item Descriptor Inheritance**

The SQL repository supports a simplified form of inheritance that uses an optional one-to-one relationship between the primary table and an auxiliary table. The same repository can define one item descriptor that inherits properties from another.

For example, a clothing store catalog might offer shirts and shorts. Because the two items are likely to have common properties, you might use inheritance as follows:

- Define a base item descriptor class clothing which defines the properties common to shirts and shorts.
- Define item descriptors shirt and shorts as sub-types of clothing, so they inherit the properties defined in clothing.

The data model for the clothing catalog can be represented as follows:

```
<item-descriptor name="address">
<table name="address" id-column-names="address_id" type="primary">
  <property name="streetAddress"/>
  <property name="city"/>
  <property name="state"/>
  <property name="zip"/>
</table>
</item-descriptor>
```

This approach has several advantages

- Avoids duplicated database columns and code.
- Facilitates queries across multiple sub-types, such as shirt and shorts. For example:

  ```
  find all clothing items where description contains 'shorts'
  ```

The XML repository definition (with inheritance-related tags in bold face) looks like this:
These definitions utilize inheritance as follows:

- The parent item descriptor `clothing` defines properties that are common to its sub-types `shirt` and `shorts`—for example, `color` and `size`. This item descriptor’s `sub-type-property` attribute points to the enumerated property `type`, which specifies this item descriptor’s sub-types.

- The item descriptors `shirt` and `shorts` define themselves as sub-types of the `clothing` item descriptor through the attributes `super-type` (set to `clothing`) and `sub-type-value` (set to `shirt` and `shorts`, respectively).

**Note:** Instances of objects are associated with their superclasses by ID. So, in this example, a shirt ID always has a matching clothing ID.

A sub-type item descriptor must never set `sub-type-value` to NULL. Given the previous example: some clothing items might be neither shirts nor shorts. In this case, the `clothing` item descriptor should set its `sub-type-value` attribute to `clothing` and add `clothing` as an option to its `sub-type-property`:
From the Repository API point of view, each ItemDescriptor maps to a single RepositoryView. When an SQL repository uses item type inheritance, each parent item type results in a RepositoryViewContainer that contains its subtype views as children.

**Benefits of Item Descriptor Inheritance**

The built-in inheritance capability of SQL repositories lets you easily query across a complex hierarchy of sub-types, and helps optimize performance. For example, given the previous model, the following query can return both shirt and shorts items:

get all clothing items with a shipping weight > 2 pounds

The code for this query looks like this:

```java
// get hold of the repository
Repository gsa = ...;

// get the view to use for querying "clothing" type items
RepositoryView clothingView = gsa.getView("clothing");

// get a query builder
QueryBuilder qb = clothingView.getQueryBuilder();

// build the query
QueryExpression weightLimit = qb.createConstantQueryExpression(new Integer(2));
QueryExpression itemWeight = qb.createPropertyQueryExpression("shippingWeight");
Query q = qb.createComparisonQuery(itemWeight,
                                 weightLimit,
                                 QueryBuilder.GREATER_THAN);

// run the query
RepositoryItem[] items = clothingView.executeQuery(q);

// separate the shirts and shorts and do whatever with them
for (int i=0; i<items.length; i++) {
```
RepositoryItem item = items[i];

// all clothing items have a name and a description
logDebug("clothing: " + item.getPropertyValue("name") + "' + item.getPropertyValue("description");

// the ItemDescriptor defines the "type" of an item
RepositoryItemDescriptor desc = item.getItemDescriptor();

// now we do different things, depending on the
type of clothing item we have
if (desc.getItemDescriptorName().equals("shirt")) {
    // shirts have a property called "season"
    logDebug("tshirt, season = " + item.getPropertyValue("season");

    // do shirt-related things
    myShirtProcessor(item);
}
else {
    // shorts have a property called "pleated"
    logDebug("tshort, season = " + item.getPropertyValue("pleated");

    // do shorts-related things
    myShortsProcessor(item);
}

This example uses the name of the item descriptor to determine the item type. You can also look at the
value of the type property declared in your template. In the example, the enumerated properties are
defined with the useCodeForValue attribute set to true. As result, the query looks like this:

... 
RepositoryItem item = items[i];

Integer itemTypeCode = (Integer)item.getPropertyValue("type");
if (itemTypeCode.intValue() == 0)
{
    ... shirts ...
} else
{
    ... shorts ...
}

Choice of one approach over another is largely a matter of style. The item descriptor approach uses the
actual name like shirt or shorts. The type attribute approach uses the type code stored in the clothing
table: typically something like 0 or 1, as in this case.
Queries and Item Descriptor Inheritance

The SQL repository query system lets you create a query against a parent item descriptor that returns items of a child item descriptor. For example, a repository might define the following item descriptors and properties:

Given these inheritance relationships, you can create the following queries against items of type `products`, although `products` does not contain the queried properties—`waterproofRating`, `size`, and `channelCount`:

- `products whose waterproofRating is 'JIS-4'`
- `products whose channelCount = 7`
- `products whose waterproofRating is 'JIS-4' OR whose size > 7`

Item Descriptor Inheritance with the copy-from Attribute

You may want to make a copy of an item descriptor without doing any dynamic typing of that item descriptor. If so, you can use the `copy-from` attribute. This creates a copy of the specified item descriptor, to which you can add additional properties. This is a way to share property definitions between different item descriptors, or perhaps to add additional properties to an existing item descriptor definition. Because these item descriptors share the same tables (and thus the same data), it might be unwise to use both old and new item descriptors. Instead, it might be better to use the `super-type` attribute and `sub-type-property` attribute if you want to use both.

Limitations of SQL Repository Inheritance

The SQL repository’s inheritance support has the following constraints:

- A type can only inherit from one parent.
- A class hierarchy can only have one `sub-type-property` value. You can define a second level of sub-classing—for example, you might define an item descriptor named `bermuda-shorts` that has `shorts` as its super-type—but you cannot have another different `sub-type-property`.
All parent item descriptors (item descriptors that are used in `super-type` or `copy-from` attributes) must be fully defined by the time they are referenced in the XML repository definition file. They can be defined in front of the new XML file in the same file, or specified in an XML file that is parsed before this XML file.

You should avoid using too many levels of inheritance. Queries against items whose properties span multiple sub-types may require joins of all tables in the hierarchy. If you use these kinds of queries, keep in mind that performance decreases as the number of tables joined increases.

**Derived Properties**

An SQL repository can define derived properties, where one repository item derives property values from another repository item or from another property in the same item. Derived properties are important to data models that use a tree structure, where certain property values are passed down from other properties.

**Note:** Using derived properties can affect performance: the more complex the derivation, the greater the likely impact.

**Derivation Syntax**

An item descriptor defines a derived property value through `<derivation>` and `<expression>` tags, as follows:

```xml
<property name="target-property" [attributes]...>
  <derivation [attributes]...>
    <expression> source-property </expression>
  ...
</derivation>
</property>
```

The `<derivation>` tag encloses one or more `<expression>` tags. Each `<expression>` tag encloses a repository item property name, which provides one potential source of the derived value. `<derivation>` tag attributes specify how the expressions are parsed to provide a value.

For example, an organization might be a hierarchy of divisions, departments, and employees. A repository represents this hierarchy with the item descriptors `division`, `department`, and `employee`, respectively. Each item descriptor defines a `spendingLimit` property. A business rule might specify that an employee’s spending limit, unless explicitly set for that employee, is derived from the employee’s department. If no department spending limit is set, it is derived from the department’s division.

This derived property relationship is represented in a repository definition file as follows:

```xml
<item-descriptor name="employee">
  ...
</item-descriptor>
```
Recursive Properties Notation

A derived property expression can specify multiple levels of subproperties. For example, an employee's spending limit might be derived as follows:

```xml
<item-descriptor name="employee">
    <property name="department" item-type="department"/>
    <property name="spendingLimit" data-type="int" writable="false">
        <derivation>
            <expression>department.employeeDefaultInfo.spendingLimit</expression>
        </derivation>
    </property>
</item-descriptor>

<item-descriptor name="department">
    <property name="employeeDefaultInfo" item-type="employeeInfo"/>
    <property name="deptSpendingLimit" data-type="int"/>
</item-descriptor>

<item-descriptor name="employeeInfo">
    <property name="spendingLimit" data-type="int" writable="false"/>
</item-descriptor>
```
Writable and Non-writable Derivations

The firstNonNull derivation method must be non-writable, unless you set a writable override property for the derived property. You can set a property to be not writable like this:

```xml
<property name="spendingLimit" data-type="int" writable="false"/>
```

The derivation methods firstWithAttribute and FirstWithLocale can be writable, even if the property does not define a writable override property. See Override Properties in this section for more information.

Override Properties

You can explicitly set a property value rather than have the property derivation logic supply one, by specifying an override-property attribute in the <derivation> tag:

```xml
<derivation override-property="empSpendingLimit">
  <expression>department.spendingLimit</expression>
</derivation>
```

The derivation tag here specifies that if the empSpendingLimit property is not null, it is used as the value of the spendingLimit property, otherwise the spendingLimit property is derived as before. The override-property attribute lets you edit in a Repository Editor a property that is otherwise derived from another property.

Properties Derived from the Same Item

Properties do not have to derive from subproperties. They can also derive from properties in the same item. For example, suppose a user item descriptor defines a home address and a shipping address. The ship-to address can inherit its value from the home address, like this:

```xml
<derivation>
  <expression>shippingAddress</expression>
  <expression>homeAddress</expression>
</derivation>
```
Complex Derivations

The previous examples show properties that derive their value from a simple hierarchy. Property derivation expressions can also specify various unrelated properties. In the following example, a shipping address is derived from one of the following sources:

- Shipping address
- Billing address
- Home address
- Company address

To determine the value of the user's `shipToAddress` for a user, the expressions specified in the derivation are searched in order. Any expression may also refer to properties that are themselves derived.

Derivation Methods

A derived property definition can specify one of several different derivation methods to determine the appropriate property value. The SQL repository traverses in order each of the expressions in the `<derivation>` tag, applying the specified derivation method. There are six derivation methods included in the ATG platform:
- `firstNonNull`
- `firstWithAttribute`
- `firstWithLocale`
- `alias`
- `union`
- `collectiveUnion`

**firstNonNull**

By default, the SQL repository derives a property by traversing the expressions in order, starting with the property itself. The first non-null value found is used as the property value. This is the `firstNonNull` derivation method.

The `firstNonNull` method is the default derivation method, and so it is not necessary to specify it in the XML. However, the derivation method can be specified in the `method` attribute of a `<derivation>` tag in the SQL repository definition file, as in this example:

```xml
<item-descriptor name="employee">
    <property name="department" item-type="department"/>
    <property name="empSpendingLimit" data-type="int"/>
    <property name="spendingLimit" writable="false">
        <derivation method="firstNonNull">
            <expression>empSpendingLimit</expression>
            <expression>department.spendingLimit</expression>
        </derivation>
    </property>
</item-descriptor>
```

**firstWithAttribute**

The `firstWithAttribute` method requires you to specify an attribute named `derivationAttribute`. The code iterates through the expressions in order, and uses the first property with an attribute that matches the value of the `derivationAttribute`. If the value with the real key is null, the value of the `defaultKey` is used.

For example:

```xml
<item-descriptor name="myItem">
    <property name="name">
        <derivation method="firstWithAttribute">
            <expression>englishName</expression>
            <expression>icelandicName</expression>
            <expression>shonaName</expression>
        </derivation>
        <attribute name="derivationAttribute" value="language"/>
        <attribute name="defaultKey" value="en"/>
    </property>
</item-descriptor>
```
If `getKey` returns `sn` (the user is in Zimbabwe, for example) `myItem.name` returns the same value as `myItem.shonaName`.

**firstWithLocale**

The `firstWithLocale` method is a subclass of `firstWithAttribute`. It performs the following actions:

1. Gets the user's current locale as the key from the Nucleus component defined by a `keyService` attribute.
2. Compares this locale to each expression's `locale` value.
3. Returns the first property whose attribute matches.

The locale is searched in a locale-specific way. For example, if `locale=fr_FR_EURO`, it first looks for a property where the locale attribute is `fr_FR_EURO`, then looks for `fr_FR`, and finally looks for `fr`.

There is also a `defaultKey`, which the `keyService` uses if the value with the real key is null. In other words, if the real key is `de_DE` and you are looking for `displayName`, but `displayName_de` is null, `displayName_en` is returned instead (assuming its `locale` is `en` and the `defaultKey` is `en` or `en_US`).

Using a `defaultKey` can slow performance. If no default key is defined, it is not used. If the default key is the same as the current key, there are no performance implications. In all other cases, there is an extra clause on all search terms, which can result in a slower search.

The following example of a derived property definition uses the `firstWithLocale` derivation method:

```xml
<property name="displayName" data-type="string">
  <derivation method="firstWithLocale">
    <expression>displayName_en</expression>
    <expression>displayName_de</expression>
  </derivation>
  <attribute name="derivationAttribute" value="locale"/>
  <attribute name="keyService" value="/atg/userprofiling/LocaleService"/>
  <attribute name="keySubProperty" value="locale"/>
  <attribute name="defaultKey" value="en"/>
</property>
```
**alias**

The Alias derivation method lets you define an alternate name for a repository item property and use either name to access the property. This can be useful in situations where different application modules use the same property, but want to use different names for the property. A single alternate name can be defined in an `<expression>` element within a `<derivation>` element.

For example, suppose an item descriptor defines a property named `firstName`. You want some application code to refer to this property as `name1`. You can use the Alias derivation method to define `name1` to be the equivalent of `firstName`, as follows:

```xml
<item-descriptor name="user" ...>
  <table name="USER" ...>
    <property name="firstName" ...>
      ....
    </property>
  </table>
  <property name="name1">
    <derivation method="alias">
      <expression>firstName</expression>
    </derivation>
  </property>
</item-descriptor>
```

In this example, when the `name1` property is accessed, the `firstName` property of the item is returned.

**union**

The Union derivation method enables the combination of several properties of a repository item into a single property. The class takes two or more set or list type properties, and combines the values of those properties in the current repository item to create the new derived property. The members of the set or list can be of any data type supported by the Repository API. For example, suppose you have set type properties named `brothers` and `sisters`. You can use the Union derivation method to create a derived property named `siblings` that combines the values of the `brothers` and `sisters` properties into a single property.

```xml
<property name="siblings">
  <derivation method="union">
    <expression>brothers</expression>
    <expression>sisters</expression>
  </derivation>
</property>
```

The `siblings` property represents a union of values in the sets `brothers` and `sisters`. The data type of the values in the collections defined in all the expressions of the derived property must be the same.

If two or more of the properties to be combined in the Union derived property include the same element, the Union derived property has duplicate elements if the property is of type `list`, but has unique elements if the property is of type `set`. 

76
**collectiveUnion**

The CollectiveUnion derivation method enables a property to be derived from a union of subproperties. The `expression` element is a property in the item descriptor that represents a collection of values. The derived property returns a union of the properties indicated by the value of the `collectionProperty` attribute.

For example, suppose you have an item descriptor named `sku`. Each `sku` type item has a `parentProducts` property, which is a collection of `product` type repository items. The following defines a `catalogs` property that is derived from a union of the `catalogs` properties of the items that make up the `parentProducts` property.

```xml
<item-descriptor name="sku" ...
<property name="catalogs">
   <derivation method="collectiveUnion">
      <expression>parentProducts</expression>
   </derivation>
   <attribute name="collectionProperty" value="catalogs"/>
</property>
<property name="parentProducts" data-type="set" component-item-type="product">
   ...
</property>
<item-descriptor name="product" ...
<property name="catalogs" ... />
<property name="parentProducts" data-type="set" component-item-type="product">
   ...
</property>
</item-descriptor>
```

In this example, the union of `product.catalogs` is returned for each product in the `parentProducts` property of the `sku` item. The derived property is accessible through the `sku` item's `catalogs` property.

---

**Repository Items and Session Backup**

Serialized JavaBeans can be saved on a session backup server. If the ATG server originally handling a session becomes unavailable, important elements of a user's session can be recreated from these serialized beans. A property's `serialize` attribute tag lets you customize how to handle that property when a repository item is serialized.

If you serialize a transient item, its properties are serialized along with it. You can exclude individual properties from serialization as follows:
If you serialize a persistent item, only transient properties (those not in a table tag) are serialized unless their `serialize` attribute is set as follows:

```xml
<property ...>
  <attribute name="serialize" value="false"/>
</property>
```

If you serialize a persistent item, only transient properties (those not in a table tag) are serialized unless their `serialize` attribute is set as follows:

```xml
<property ...>
  <attribute name="serialize" value="true"/>
</property>
```

If the item is persistent, its persistent properties are written to the database and can be retrieved if the session needs to be restored.
7 SQL Repository Item Properties

An item descriptor in an SQL repository can define special types of properties. The following sections describe some of these special property types, as well as other useful repository property attributes:

- Enumerated properties
- Required properties
- Unique properties
- Date and timestamp properties
- Null properties
- Property validation with a property editor class
- Maintaining item concurrency with the version property
- Repository items as properties
- Transient properties
- Assigning FeatureDescriptorValues with the <attribute> tag
- Linking between repositories
- SQL types and repository data types
- User-defined property types
- Property fetching
- Handling large database columns

Enumerated Properties

Enumerated item properties are string properties that are constrained to a predefined list of valid values. Generally, an enumerated item property should provide access to a small list of valid values. A TaggedPropertyEditor is registered for enumerated properties so components like user interfaces can access the list of valid values.

ATG supports two enumerated data types:

- \texttt{enumerated}: Stores integer codes to the database.
- \texttt{enumerated String}: Stores string codes to the database.
enumerated

The following item descriptor definition creates an enumerated property named `transactionType`. The definition provides a list of valid String values; the SQL repository generates the corresponding integer codes when the template is initialized:

```xml
<!-- The "transaction" item type -->
<item-descriptor name="transaction">
  <table name="transaction" id-column-names="xact_id">
    <property name="amount" data-type="int"/>
    <property name="transactionType" data-type="enumerated">
      <option value="credit"/>
      <option value="debit"/>
      <option value="purchase"/>
    </property>
  </table>
</item-descriptor>

Setting Integer Codes

You can explicitly specify the integer codes with the `<option>` tag's `code` attribute:

```xml
<property name="transactionType" data-type="enumerated">
  <option value="credit" code="200"/>
  <option value="debit" code="201"/>
  <option value="purchase" code="202"/>
</property>
```

Reserved Enumerated Property Integer Codes

Avoid assigning integer codes to enumerated properties that collide with integer codes that are used or reserved for future use by ATG products. In general, it is safe to assign enumerated properties integer codes within the range of 101-999 (some older ATG versions use integer code values between 0-100). You can also safely use negative integers.

The following option codes are reserved for use by ATG modules and products:

<table>
<thead>
<tr>
<th>Module/Product</th>
<th>Reserved Option Code Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAF</td>
<td>1000 - 1999</td>
</tr>
<tr>
<td>DPS</td>
<td>2000 - 2999</td>
</tr>
<tr>
<td>DSS</td>
<td>3000 - 3999</td>
</tr>
<tr>
<td>ATG Commerce B2C</td>
<td>4000 - 4999</td>
</tr>
<tr>
<td>ATG Commerce B2B</td>
<td>5000 - 5999</td>
</tr>
</tbody>
</table>
# ATG Repository Guide

## Module/Product Reserved Option Code Values

<table>
<thead>
<tr>
<th>Module/Product</th>
<th>Reserved Option Code Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATG Portal</td>
<td>6000 - 6999</td>
</tr>
<tr>
<td>ATG Outreach</td>
<td>7000 - 7999</td>
</tr>
<tr>
<td>ATG Content Administration</td>
<td>8000 - 8999</td>
</tr>
<tr>
<td>ATG Ticketing</td>
<td>9000 - 9999</td>
</tr>
<tr>
<td>ATG Knowledge and Self Service</td>
<td>10000 - 10999</td>
</tr>
<tr>
<td>ATG Commerce Service Center</td>
<td>11000 - 11999</td>
</tr>
<tr>
<td>ATG Response Management*</td>
<td>12000 - 12999</td>
</tr>
<tr>
<td>ATG Search</td>
<td>13000 - 13999</td>
</tr>
<tr>
<td>Agent</td>
<td>14000 - 14999</td>
</tr>
<tr>
<td>ATG Chat*</td>
<td>15000 - 15999</td>
</tr>
<tr>
<td>future ATG use</td>
<td>16000 +</td>
</tr>
</tbody>
</table>

* Product no longer supported

### Converting Integer Codes to Strings

By default, an enumerated property returns its value as an integer code. You can configure an enumerated property so the repository converts the integer code into a string value by setting the `useCodeForValue` attribute to `false`. For example, you might modify the previous definition as follows:

```xml
<property name="transactionType" data-type="enumerated">
  <attribute name="useCodeForValue" value="false"/>
  <option value="credit" code="200"/>
  <option value="debit" code="201"/>
  <option value="purchase" code="202"/>
</property>
```

Given this definition, the string value `credit`, `debit`, or `purchase` is returned when you get the `transactionType` property.

Conversely, if `useCodeForValue` is set to `true` (the default), the integer code is returned. If an enumerated property returns an integer code, you can get the property editor for an enumerated property and use it to create a property editor that can convert between string and integer codes. See the JavaBeans specification for a description of `PropertyEditors`.

---

**7 - SQL Repository Item Properties**

81
**enumerated String**

By defining a property as an *enumerated String* instead of an *enumerated* data type, the following options are available to you:

- Save enumerated codes as String values in the database.
- Save one or more strings from a list of multiple choice strings.

You provide the property with the values by adding the following attribute:

```xml
<attribute name="stringEnumProvider" value="string-enum-provider-value"/>
```

where *string-enum-provider-value* is an implementation of *atg.adapter.gsa.StringEnumProvider*. The ATG installation provides two implementations for this interface, which are referenced by two properties in the site repository:

- `/atg/multisite/SiteTypesProvider`: Specified by `siteConfiguration.siteTypes`, obtains a list of site types from the SiteManager.
- `/atg/multisite/ShareableTypeStringEnumProvider`: Specified by `siteGroup.shareableTypes` properties, obtains a list of shareable types from the SiteGroupManager.

For example, the site repository's item descriptor `siteConfiguration` defines its `siteTypes` property as a collection of *enumerated String* values as follows:

```xml
<item-descriptor name="siteConfiguration" ... >
...

<property name="siteTypes"
  column-names="site_type"
  data-type="set"
  component-data-type="enumerated string"
  category-resource="categoryBasics"
  display-name-resource="siteTypes">
  <attribute name="stringEnumProvider" value="/atg/multisite/SiteTypesProvider"/>
  <attribute name="propertySortPriority" value="100"/>
</property>
...
</item-descriptor>
```

**Required Properties**

You can require that a repository item property is always set to a non-null value as follows:

- Set property tag's *required* attribute. For example:

  ```xml
  <property name="lastName" data-type="string" required="true"/>
  ```
Set the repository component’s `enforceRequiredProperties` property component to `true` (the default value). This ensures that the repository checks that all required properties are present when adding repository items, and forbids the setting of a required property to null.

The repository definition must conform to its database schema; if the schema defines a property as NOT NULL, the repository definition must set the property’s `required` attribute to `true`.

**Constraints**

The following constraints apply to usage of the `required` attribute:

- If a property references an item that is defined in the database as NOT NULL but you cannot mark the property as required, indicate this by adding the `references` attribute tag and set its value to `true` and its data type to `boolean`. For example:

  ```xml
  <property name="myProperty">
    <attribute name="references" value="true" data-type="boolean"/>
  </property>
  ```

- The `required` attribute is not supported for items that reference collections—that is, item descriptor definitions where `data-type="set"`.

**Unique Properties**

Some repository item properties require unique values—for example, a user profile’s login property should not be shared by other user profiles. You mark a property as unique as follows:

```xml
<property name="login" data-type="string" required="true">
  <attribute name="unique" value="true"/>
</property>
```

Repository editors in the ATG Control Center enforce the requirement that the value be unique.

**Date and Timestamp Properties**

A repository item can have properties that are set to the current date or time, using the `java.util.Date`, `java.sql.Date`, or `java.sql.Timestamp` classes. You can have a property whose value is set to the current time or date by setting the feature attribute `useNowForDefault` to `true`. For example:

```xml
<property name="creationDate" data-type="timestamp">
  <attribute name="useNowForDefault" value="true"/>
</property>
```
You can query on date and timestamp properties by using the RQL functions date and timestamp. For more information, see Date and Timestamp Queries in the Repository Query Language chapter.

For more information about this technique, see the Assigning FeatureDescriptorValues with the <attribute> Tag section in this chapter.

**Last-Modified Properties**

In some applications, it is useful to know when a repository item was most recently modified. The following item descriptor contains a last-modified property:

```xml
<item-descriptor name="article" last-modified-property="lastActivity">
  <attribute name="updateLastModified" value="true"/>
  <table name="ARTICLES" type="primary" ...>
    <property name="lastActivity" data-type="timestamp"/>
  ...  
</table>
</item-descriptor>
```

Three requirements apply:

- The item descriptor contains a date or timestamp property that stores the last-modified value. This property must be persistent and single-valued:
  ```xml
  <property name="lastActivity" data-type="timestamp"/>
  ```

- The item descriptor sets the last-modified-property attribute to the name of the last-modified property:
  ```xml
  <item-descriptor name="article" last-modified-property="lastActivity">
  ```

- The item descriptor sets the updateLastModified <attribute> element to true:
  ```xml
  <attribute name="updateLastModified" value="true"/>
  ```

Given this example, an article item’s lastActivity property is updated with the current time when the item is added or updated.

**Null Properties**

A property is set to null if its definition does not set a default value. For example:

```xml
<property name="favoriteColor" data-type="string"/>
```

You can also explicitly set a property’s default value to null as follows:

```xml
<property name="favoriteColor" data-type="string" default="__NULL__"/>
```
This technique is useful if, for example, you combine two or more repository definition files into a single template and need to override a non-null value and restore the default to null.

**Grouping and Sorting Properties**

In order to group similar properties in a Repository user interface, include the `category` attribute in their respective `<property>` tags and set the attribute to the same value. For example, the `<property>` tags that define login and password properties each set their `category` attribute to `Login`; thus, the login name and password are listed together under the heading Login.

By default, properties with the same `category` setting are listed in ascending alphabetical order of their `display-name` settings. You can explicitly control the display order of grouped properties by setting their `propertySortPriority` attributes to the desired integer values; in that case, the properties are displayed in ascending numeric order. The default `propertySortPriority` setting is 0.

For example, ATG user profile definitions set three name properties (first, middle, last) as follows:

```xml
<property category="Basics" name="firstName" data-type="string"
  display-name="First name">
  <attribute name="propertySortPriority" value="-3"/>
</property>

<property category="Basics" name="middleName" data-type="string"
  display-name="Middle name">
  <attribute name="propertySortPriority" value="-2"/>
</property>

<property category="Basics" name="lastName" data-type="string"
  display-name="Last name">
  <attribute name="propertySortPriority" value="-1"/>
</property>
```

Given these settings, the Basics category for a user profile the ATG Control Center displays name properties in the following order:

- First name
- Middle name
- Last name

**Category Ordering in the ACC**

The ACC lists categories in the following order:

1. All named categories that contain required properties (properties with the attribute `required="true"`).
2. All named categories that do not contain required properties.
3. A special “anonymous” category containing properties that are not assigned to any category.

4. A special “Groups” category containing boolean properties that correspond to content or profile groups.

Within each of these four sets, categories are listed in alphabetical order.

Property Validation with a Property Editor Class

You can specify a property editor class to use with a property, using the editor-class attribute. For example, the following tag associates a special property editor with the password property:

```xml
<property name="password" data-type="string" required="true"
   editor-class="atg.beans.PasswordPropertyEditor"/>
```

A property editor can be used to validate a property value. Note, however, that a property editor does not have access to the repository item, only the property. Therefore, you cannot use a property editor to make comparisons with other properties.

You can also limit a property’s values to a list of valid choices with an enumerated property. See the Enumerated Properties section.

Maintaining Item Concurrency with the Version Property

The SQL repository can use a system of optimistic locking to maintain consistent versions of repository items. This optimistic locking system can be used in combination with any repository caching mode: disabled, simple, locked, and distributed.

To use the optimistic locking system for an item descriptor, add a version property to the item descriptor and a corresponding version column to the primary database table for the item descriptor. This version property must use data-type="int" or data-type="long" and the database column must be of a type that is compatible with the repository int or long type. The version property is identified in the item-descriptor tag with an attribute named version-property, the value of which is the name of the version property. For example:

```xml
<item-descriptor name="news" version-property="version">
  <table name="business_news" id-column-names="id">
    <property name="version" data-type="int"/>
    ...
  </table>
</item-descriptor>
```
The value of the version property is incremented every time the item is updated. Its value starts as 0 when the item is created, is set to 1 when the item is added, and is incremented in each subsequent update.

The version number for a particular item is read and associated with that transaction the first time that item is referenced in a transaction. If you try to update the item from a transaction whose version does not match the current version number in the database, a `ConcurrentUpdateException` is thrown to abort that update. This exception is a subclass of `RepositoryException`.

Here is a sample scenario that shows how the SQL repository uses the version property to implement optimistic locking:

1. Dynamo1 reads a repository item for update. It obtains the item’s version property, which has a value of 2.

2. Dynamo2 reads the same repository item for update. Because Dynamo1 has not yet committed any changes to the item, Dynamo2 gets the same value for the item’s version property, 2.

3. Dynamo1 updates the repository item. In the course of the update, the value for the version property in the repository item is checked to see whether it is the same as what is found in the corresponding database column. In this case, both values are still 2. The update to the repository item is committed, with the version property incremented by 1, so the value of the version property is now 3.

4. Dynamo2 tries to update the repository item. When the value for the version property in the repository item is checked to see whether it is the same as what is found in the corresponding database column, the values do not match. Dynamo2 is holding a value of 2, while the value of the version property in the database is now 3. Dynamo2 throws a `ConcurrentUpdateException` and does not apply the changes in the update.

This can be very useful for simple and distributed caching modes where there is a possibility of overwriting another Dynamo’s changes.

You can take advantage of optimistic locking in pages that include forms. Often in a form, you read the data in one transaction and update the data in another transaction. There is a possibility that another process might try to update an item in an intermediate transaction. To handle this case, you can place the version property value as a hidden field in your form. Then, you can either check that it is still the same yourself after you start the transaction which updates the item, or you just set the version property (along with the other properties in the item) and deal with the `ConcurrentUpdateException` when it occurs. For example, you can include in a page a hidden input tag like this:

```html
<input type="hidden" bean="FormHandler.value.version"/>
```

You can also use a `RepositoryFormHandler`, which can set a version property just like it sets any other property of a repository item.

---

**Repository Items as Properties**

The value of a property of a repository item can be another repository item. Both multi-valued properties and single-valued properties can have repository items as property values. This is a powerful feature, and
allows you much greater flexibility in defining a database schema that your application accesses as a repository.

Consider as a simple example a repository that contains books and authors. You can represent both books and authors as repository items, which enables you to do things like this:

```java
    Repository gsa = ...;
    String myBookId = ...;

    // get my book from the db
    RepositoryItem book = gsa.getItem(myBookId, descriptorName);

    // get the author of my book (it's a dynamic bean too!)
    RepositoryItem author = (RepositoryItem)book.getPropertyValue("author");
```

Without support for objects as properties, the application must get an `authorId` from the book and perform another lookup to get the actual author.

You can specify that a repository item property is another repository item, rather than a primitive, with the `item-type` attribute instead of the `data-type` attribute. The following example shows a portion of a template that defines two item descriptors, book and author:

- Repository items of the book item descriptor have an author property whose value is another repository item, an author.
- Repository items of the author item descriptor have a book property whose value is another repository item, a book.

```xml
    <!-- The "book" item type -->
    <item-descriptor name="book" default="true">
        <table name="book" type="primary" id-column-names="book_id">
            <property name="title"/>
            <property name="author" column-name="author_id" item-type="author"/>
        </table>
    </item-descriptor>

    <!-- The "author" item type -->
    <item-descriptor name="author">
        <table name="author" id-column-names="author_id" type="primary">
            <property name="lastName"/>
            <property name="firstName"/>
        </table>
        <table name="book" id-column-names="author_id" type="auxiliary">
            <property name="book" item-type="book" column-name="book_id"/>
        </table>
    </item-descriptor>
```
Multiple Item Properties

You can also specify that a repository item property references other repository items. In the previous example, an author may have written more than one book. Instead of the book property in the preceding example, this next example uses a books_written property whose value is a Set of book repository items. The <property> tag for the books_written property uses the following attributes:

- `data-type="set"` 
  Specifies that the property value is a Set of items

- `component-item-type="book"` 
  Specifies that the items making up the set are items of the book item descriptor

- `column-name="book_id"` 
  Specifies that the database column is named book_id, rather than books_written.

```xml
<!-- The "book" item type -->
<item-descriptor name="book" default="true">
    <table name="book" type="primary" id-column-names="book_id">
        <property name="title"/>
        <property name="author" item-type="author" column-name="author_id"/>
    </table>
</item-descriptor>

<!-- The "author" item type -->
<item-descriptor name="author">
    <table name="author" id-column-names="author_id" type="primary">
        <property name="lastName"/>
        <property name="city"/>
        <property name="state"/>
        <property name="zip"/>
    </table>
    <table name="book" id-column-names="author_id" type="multi">
        <property name="books_written" data-type="set"
            component-item-type="book"
            column-name="book_id"/>
    </table>
</item-descriptor>
```

In this example, the repository definition XML defines the book table twice: in the book item descriptor and in the author item descriptor. In the author item descriptor, the <table> tag for the book table includes the attribute type="multi", indicating that each author item can have more than one row with the id column. In a multi-table, all defined attributes are multi-valued types. To define Array, List and Map types, you also must specify a multi-column-name attribute on the table tag. This specifies which column to use as the sorting value in order to determine the order of the List and the key for the Map.

Now the properties author and books_written are actually real beans (in this case RepositoryItems) instead of just simple Java primitives. In the author item descriptor, the books_written property is a Set of RepositoryItems that correspond to books. The other types supported are List, Map, and Array.
Adding an Item to a Multi-Item Property

How you add a repository item to a `Set`, `List`, `Map`, or `Array` partly depends on how you use the `cascade` attribute. For example, the previous definition of the `books_written` property might be modified to set `cascade` to `update`:

```xml
<property name="books_written" data-type="set"
    component-item-type="book"
    column-name="book_id"
    cascade="update"/>
```

This setting makes it easier for you to keep the book repository items synchronized with the author repository items that refer to them. See the Cascading Data Relationships section. You can add a book item to a set of items in the `books_written` property like this:

```java
Repository gsa = ...;
RepositoryItem newBook = getRepository().createItem("book");
Set books_written = (Set) author.getPropertyValue("books_written");
books_written.add(newBook);
```

If the `books_written` property does not have `cascade="update"`, you must add the item with the `addItem()` method (thus inserting the row in the database) before you add it to the list:

```java
Repository gsa = ...;
RepositoryItem newBook = getRepository().createItem("book");
getProfileRepository().addItem(newBook);
Set books_written = (Set) author.getPropertyValue("books_written");
books_written.add(newBook);
```

Remember that in each of these cases, it is most efficient to ensure that all method calls are performed in a single transaction. See Repositories and Transactions in the SQL Repository Architecture section.

Prohibiting Duplicate Values

You can prohibit duplicate values in multi-item properties of type `List` and `Array` by setting the repository component property `prohibitCollectionDuplicates` to `true`. By default, this property is set to `false`. The repository definition of individual properties can override the repository-level setting. For example:

```xml
<property name="color"
    column-names="color" data-type="list"
    component-data-type="string">
    <attribute name="prohibitDuplicates" value="true"/>
</property>
```
Querying Subproperties

The SQL repository allows applications to query repository items based on attributes of their attributes, sometimes referred to as subproperties.

Continuing the book example, subproperty querying means that you can write repository queries like:

Get me all the books that were written by authors living in NY

This query can be represented using RQL in a `<query-items>` tag, as follows:

```
<query-items item-descriptor="book">author.state="NY"</query-items>
```

Transient Properties

The SQL repository lets you define properties of a repository item that are transient. Transient properties are never stored or read from the persistent data store. They are readable and writable, but not queryable. Transient properties provide applications a hook for custom objects that are not persisted by the repository.

You can specify a transient property by defining a `<property>` tag that is not associated with any database table, but which is instead a direct child of an `<item-descriptor>` tag. For example, in the following example, the `user` item descriptor has a transient property that specifies whether the user is logged in at that time:

```
<item-descriptor name="user" sub-type-property="userType">
  <property name="loggedIn" data-type="boolean">
    <table name="user" type="primary" id-column-names="id">
      <property name="userType" data-type="enumerated" column-name="user_type">
      ...
    </property>
  </property>
</item-descriptor>
```

You can also define an entire item descriptor to be transient. Such an item descriptor has no `<table>` tags and no properties that are direct children of a `<table>` tag. The properties of transient item descriptor are queryable by default, unlike a transient property of an item descriptor with other properties that are persistent properties. In the case of a transient item descriptor, no indexing is used, so queries against large repositories are slow. Using transient repositories is sometimes a useful testing tool during application development.
Assigning FeatureDescriptorValues with the `<attribute>` Tag

You can use the `<attribute>` tag as a child tag in a `<property>` or `<item-descriptor>` tag to associate arbitrary name/string value pairs with the property. These named values correspond to those specified in `java.beans.FeatureDescriptor`, which `RepositoryPropertyDescriptor` implements.

This is simply a way of letting applications associate more metadata with individual properties. The SQL repository does not do anything with the data expressed in the `<attribute>` tag; it remembers the values defined in the template and allows one to read them at runtime. This is the same mechanism that the `@beaninfo` Javadoc tag system uses.

Here is an example, which assigns to the `author` property the name/value pair of `maxLength="30"`:

```xml
<!-- The "book" item type -->
<item-descriptor name="book">
   <table name="book" id-column-names="book_id" type="primary">
      <property name="title"/>
      <property name="author">
         <attribute name="maxLength" value="30"/>
      </property>
   </table>
</item-descriptor>
```

It is also useful to refer to values of Nucleus components as attributes. You can do this with the `bean` attribute of the `<attribute>` tag. For example:

```xml
<attribute name="documentRootPath"
   bean="/atg/demo/QuincyFunds/repositories/FeaturesDataStore.relativePathPrefix"/>
```

If you use a relative Nucleus address for the `bean` attribute, it refers to a component relative to the `Repository` component.

You can access property attributes programmatically with the `RepositoryPropertyDescriptor.getValue` method. For example:

```java
RepositoryPropertyDescriptor.getValue("maxLength");
```

Attributes Used in the ACC

ATG defines a set of feature descriptor attributes that modify how a repository item property is treated in the ATG Control Center. These attributes are defined by an `<attribute>` tag within a `<property>` tag. For example:
The default value for each of these attributes is `true`:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uiqueryable</code></td>
<td>If this attribute is set to <code>false</code>, the property is not available to a targeting UI element or an ATG expression editor in the ACC.</td>
</tr>
<tr>
<td><code>uiwritable</code></td>
<td>If this attribute is set to <code>false</code>, the property cannot be modified or set in the ACC. Unless the property tag also has the <code>writable=false</code> attribute, the property can still be modified or set programmatically or through a form.</td>
</tr>
</tbody>
</table>

**Linking between Repositories**

A property value can refer not just to another type of repository item, but also to a repository item in another repository. When you define a property that refers to an item in a different repository, use the `repository` attribute as part of the property tag. For example, if you had a `workAddress` item type in an LDAP repository, you might refer to it in an SQL repository like this:

```xml
<table name="employees" id-column-names="id">
  ...
  <property name="work_address"
    item-type="workAddress"
    repository="/atg/userprofiling/LDAPRepository"/>
</table>
```

The `repository` attribute can be used with the attributes `item-type` or the `component-item-type` to indicate that this item is in a different repository, not the current repository. The value of the linked property in the database is the repository ID of the item in the other repository.

The `repository` attribute specifies a Nucleus component name relative to the location of the current repository. This enables you to create a composite repository that combines the item descriptors from more than one repository. Note, however, that a single repository query cannot span repositories with different data sources.

When you use composite repositories, make sure that your affiliated repositories do not use the same item descriptor names. In this way, your application can go to the composite repository, get an item descriptor named `products`, and not have to know that it is dealing with a separate repository.
SQL Types and Repository Data Types

At application startup, the SQL repository accesses the database to determine the JDBC type of each property in the repository template. This enables the SQL repository to use the appropriate JDBC type in creating SQL queries. For large database schemas, however, getting the JDBC type of every column in your schema can take an unacceptably long time. If this is a problem, you can solve it in several ways:

**Set the sql-type attribute for All Repository Item Properties**

You can explicitly set the `sql-type` or `sql-types` attribute for every non-transient property in your repository template. These attributes are attributes of the `<property>` element. For example:

```xml
<property name="winning_number" data-type="long" sql-type="numeric" />
```

If you set the `sql-type` or `sql-types` attribute for every non-transient property in your repository template, the SQL repository does not try to check the JDBC type of your properties.

One drawback of this approach is that the `sql-type` can vary depending on your database software vendor. If you set the `sql-type` attribute explicitly, you must review and update the value if you move your repository implementation to a different vendor’s database software.

**Set the safeSQLTypes Property in the GSARepository Component**

The `GSARepository` class includes a property named `safeSQLTypes`. This property can be set to a comma-separated list of SQL types for which the repository always uses the default type. You can set this property to string values of SQL types like `varchar`, or to the corresponding integer values specified in the class `java.sql.Types` (for example, `-4`). The default value of this property is empty.

The SQL repository does not try to determine the JDBC types for properties in a database table if every property has its `sql-type` attribute set explicitly, or is of a type included in the `safeSQLTypes` property for that repository.

**Cache the Schema Information**

You can configure the SQL repository to cache information about your repository database schemas. Do this by setting the `updateSchemaInfoCache` property in the `GSARepository` component to `true`. The default value for this property is `false`. If you set this property to `true`, after you reassemble, redeploy, and restart your application, the server creates the directory `{atg.dynamo.home}/data/schemaInfoCache/`. This directory contains a series of files with names like `repositoryName.properties`, one file for each repository in your application. Each file in the `/data/schemaInfoCache` directory specifies the SQL type of each column in that repository’s schema, in the following format:

```
tablename.colname=SQLtype
```

The SQL type in this case is the integer type code specified in the class `java.sql.Types`. For example, setting the `updateSchemaInfoCache` to `true` in the `ProfileAdapterRepository` component might generate a file that begins like this:
These files can be generated against one database and copied to another as long as the schemas are compatible. If you change the schema and this property is enabled, you must remove the `/data/schemaInfoCache` directory so it is regenerated. The `/data/schemaInfoCache` directory does get regenerated automatically if you add a property that refers to a new column or a new table. If those files exist, they are used even if the `updateSchemaInfoCache` property is set to `false`. This enables a live server to use the schema info cache generated by a different staging server. This can be useful if one server in a cluster is unable to generate the schema information.

### User-Defined Property Types

In addition to the standard data types of repository item properties, described in Data-type Mappings: Java and SQL, the SQL repository lets you add your own types of properties. The new property types can implement both `getX` and `setX` functionality in one of two ways, depending on your requirements:

- If the property is transient, extend:
  ```java
  atg.repository.RepositoryPropertyDescriptor
  ```
- If the property is to be stored in the database and needs to appear inside a `<table>` tag, modify the SQL repository’s default property implementation and extend:
  ```java
  atg.adapter.gsa.GSAPropertyDescriptor
  ```

In either case, you can add additional configuration information to the XML file unique to your type. Also, the `PropertyDescriptor` `get` and `set` methods can set and get other property values in the same item.

### Identifying a User-Defined Property Type

You can identify your property type in one of two ways:

- Directly: Use the name of your Java class.
- Indirectly: Specify a type name of your type and register it with this method:
  ```java
  atg.repository.RepositoryPropertyDescriptor.registerPropertyDescriptorClass(String typeName, Class pPropertyDescriptorClass)
  ```

If you use the indirect method, set the `userPropertyDescriptors` property of the `GSARepository` component to include your type (or, in the alternative, call the `registerPropertyDescriptorClass()` method at some point before the XML repository definition file is loaded).
Most users find the direct Java class approach simpler. In either case, you set the `<property>` tag attribute `property-type` to the user-defined property type.

**Using the property-type Attribute**

The `property-type` attribute specifies the `PropertyDescriptor` class that defines your property type, as in this example:

```xml
<property name="contentFile"
    property-type="atg.repository.FilePropertyDescriptor">
    <attribute name="pathNameProperty" value="contentFileName"/>
</property>
```

Generally, if you specify a type with the `property-type` attribute, you do not need to use the `data-type` or `item-type` attribute to specify the type. However, in some cases, you might create user-defined properties that can represent more than one data type. In such cases, you can use the `data-type` attribute to further constrain the property.

**Implementing a User-Defined Property Type**

As the SQL repository parses the XML repository definition file, it creates an instance of your `RepositoryPropertyDescriptor` class and stores it directly in the SQL repository's list of property descriptors for each item descriptor. The SQL repository calls `setItemDescriptor()` to associate your property with its item descriptor. The SQL repository then calls one or more methods corresponding to the property's type:

- `RepositoryPropertyDescriptor.setPropertyType()`
- `GSAPropertyDescriptor.setComponentPropertyType()`
- `RepositoryPropertyDescriptor.setPropertyItemDescriptor()`

If the property refers to another item's type, this method:

`RepositoryPropertyDescriptor.setComponentItemDescriptor()`

sets the item descriptor for that type, depending on which attributes are set in the `<property>` tag:
`data-type`, `item-type`, `component-data-type`, or `component-item-type`. They define the property's Java class, the component's property class (if the property is a multi-valued property), the `RepositoryItemDescriptor` for a scalar or a multi-valued property that refers to other item types.

If your property type can accept any of these values, you do not need to override these methods. If your property is constrained in what data types it supports (which is generally the case), you should put error checking logic into these methods to throw errors if an invalid type is specified. Your property descriptor should throw the unchecked `IllegalArgumentException` to provide details about what type is required. If your property type is very restrictive, you can implement these methods to return the appropriate values:

- `RepositoryPropertyDescriptor.getPropertyType()`
• GSAPropertyDescriptor.getComponentPropertyType()
• RepositoryPropertyDescriptor.getPropertyItemDescriptor()
• RepositoryPropertyDescriptor.getComponentItemDescriptor()

This prevents developers of repository definitions from having to set the data-type, component-data-type, item-type, and component-item-type attributes. You may still want to put error checking in these methods to signal errors if they do provide invalid values.

Caution: When an application calls the repository item method to retrieve or set a user-defined property, it calls the repository item's own getPropertyValue() and setPropertyValue() methods, which, in turn, call the getPropertyValue() and setPropertyValue() methods that you implemented for that property. The second set of get and set methods should avoid calling the first set; otherwise, an infinite loop can occur.

The getPropertyValue method receives an extra Object pValue argument. This is set to any value found in the cache for this property name, if any, or null if no value is in the cache. The call to setPropertyValue can embed a call to setPropertyValueInCache(this, yourvalue) in order to cache this property value for subsequent method calls.

If your property is not set, you may choose to return the value of the getDefault() method on the RepositoryPropertyDescriptor. This allows the user to set the default value for this property with the default attribute in the XML tag. This method calls setDefaultValueString, which converts the default value based on the class returned by getPropertyType, which calls setDefaultValue. You may choose to modify this behavior by overriding these methods though typically this functionality is sufficient.

Note that user defined properties must be serializable. The getPropertyValue and setPropertyValue methods do not need to work on an unserialized version, but the getPropertyType, getComponentType, getPropertyItemDescriptor, and getComponentItemDescriptor methods in particular do need to work. This is important so that the ATG Control Center can understand the type of property it is editing.

To make your user-defined property queryable, it should represent a database column. Unless your user-defined property extends GSAPropertyDescriptor, the property is not queryable and you should implement the method isQueryable to return false. If you want a user-defined property to be queryable, make sure it extends GSAPropertyDescriptor. You may also override the methods isWritable and isReadable to turn off write access or read access to your property respectively. Other methods such as isHidden, isExpert can also be overridden if you want to set additional Bean attributes. The method setValue(String pName, Object pValue) is called if any feature descriptor attributes are supplied with this property.

**Property Conversion Methods**

User-defined properties that correspond directly to a column in an SQL table must extend the class atg.adapter.gsa.GSAPropertyDescriptor. These classes must implement three additional methods:

• rawToReal
• realToRaw
- **createDBPropertyEditor**

**rawToReal**

This method converts from the database version of the property value to the property value as it is returned by `RepositoryItem.getPropertyValue`. For example, for you might convert the ID of an item into the item itself. You do not need to implement this method if the default behavior of the SQL repository is what you want.

**realToRaw**

This does the opposite of `rawToReal`. It converts from the version of the value given to `setPropertyValue` into the value given to the `setObject` call in JDBC. For example, if you have a property that specifies a reference to another item, you convert from the `RepositoryItem` to its ID.

For examples of these methods, see the source code for the `atg.adapter.gsa.EnumPropertyDescriptor` class at:

`<ATG10dir>/DAS/src/Java/atg/adapter/gsa/EnumPropertyDescriptor`

- **createDBPropertyEditor**

This method is used by some reporting UIs or other tools which need to get the database value directly from SQL, but want to convert that value to or from a String representation. For example, you might perform a query against the Profile Repository, and receive the code value for an enumerated property. You can use the `createDBPropertyEditor` to convert the code value to its String representation for display in a UI, for instance. This method is like the method `createPropertyEditor`, but the property editor returned from `createDBPropertyEditor` should operate on the raw value (the value returned from the JDBC `getObject` call), not the real value (as returned by the `RepositoryItem.getPropertyValue` call).

**Null Values in User-Defined Property Types**

When the `getPropertyValue(Item, pValue)` method gets called for a user-defined property descriptor, it is given the currently cached value in the variable `pValue`. If you have previously stored a value for this property in the cache, it is given to you here. Your implementation may just choose to return that value.

Two different representations of a null value can be returned:

- `pValue = null` indicates there is no value in the cache for this property.
- `pValue = RepositoryItemImpl.NULL_OBJECT` indicates that an explicit null value was already cached for this item.

**User-Defined Properties and the ACC**

In order to property display a user-defined properly in the ATG Control Center, make sure that its Java class is available to the ACC. Package any user-defined property Java classes as part of an ATG application so the ACC can pull the classes across the RMI interface. Otherwise, a remote ACC throws an "unknown block data" error.
User-Defined Property Type Examples

Here's an example that defines a user defined property type in an XML repository definition file:

```xml
<item-descriptor name="images">
  <table name="book" id-column-names="book_id" type="primary">
    <property name="title"/>
    <property name="author"/>
    <property name="lastModifiedTime"/>
    <property name="contentFileName" data-type="string"/>
  </table>
  <property name="contentFile" property-type="atg.repository.FilePropertyDescriptor">
    <attribute name="pathNameProperty" value="contentFileName"/>
  </property>
</item-descriptor>
```

For the user defined property implementation used in this example, see the source code in:

```<ATG10dir>/Das/src/Java/atg/repository/FilePropertyDescriptor.java```

If you extend the `GSPROPERTYDESCRIBER` class, you have two additional methods that you can override. These methods convert data between the type that is stored in the database and the type that is stored in the cache. These methods are called only when the data is loaded from or stored to the database. If a cached value is found, it is returned without calling these methods. Thus it is slightly more efficient to do conversion here than in the `getPropertyValue` or `setPropertyValue` methods.

```java
public Object rawToReal(Object pRawValue)
```

```java
public Object realToRaw(Object pRealValue)
```

The following example, from the `productCatalog.xml` file in ATG Commerce defines two property descriptors:
The `data` property returns a `java.io.File` object when you call `getPropertyValue("data")` on one of the `media-external` items. The path name for this `File` object is computed by concatenating the value of the `url` property of the `media-external` item with the value of the `pathPrefix` attribute below, `./docs`. Thus the path is of the form:

`./docs/<value of the url property>`

The `mimeType` property computes a MIME type from the `url` property. It returns a MIME type string such as `text/html` from the value of the `url` property. It uses a `MimeType` component to convert the suffix to a MIME type.

```xml
<!- Media, which is stored on the external file system -->
<item-descriptor name="media-external" display-name="Media - External"
    super-type="media" sub-type-value="external"
    item-cache-size="1000" query-cache-size="100"
    version-property="version" id-space-name="media"
    content-property="data">
    <table name="dcs_media_ext" type="auxiliary" id-column-names="media_id">
        <property name="url" data-type="string" column-name="url" required="true"/>
    </table>
    <property name="data" property-type="atg.repository.FilePropertyDescriptor"
        writable="false" queryable="false">
        <attribute name="pathNameProperty" value="url"/>
        <attribute name="pathPrefix" value="./docs"/>
    </property>
    <property name="mimeType"
        property-type="atg.repository.MimeTypePropertyDescriptor"
        data-type="String" writable="false" queryable="false">
        <attribute name="identifier" value="url"/>
    </property>
</item-descriptor>
```

A new property type is defined by implementing a sub-class of the `atg.repository.PropertyDescriptor` class. In this class, you can define values for the readable, writable, and queryable properties. They also have the following additional methods that are typically overridden by a user-defined property type:

```java
// Ability to retrieve/save values to the repository item

// Ability to retrieve/save values to the repository item
```
* based on how the property is defined. For example, if null is to
* be returned, we return the default value.
*/
public Object getPropertyValue(RepositoryItemImpl pItem, Object pValue);

//-------------------------------------
/**
 * Sets the property of this type for the item descriptor provided.
 */
public void setPropertyValue(RepositoryItemImpl pItem, Object pValue);

You can register user defined property types in a static registry so they can be defined with a simple
name, like tag converters. List your user defined properties in the userPropertyDescriptors property
of the GSARepository component.

## Property Fetching

Normally, when a repository item is loaded from the database, properties in each table are loaded at the
same time. By default, all primary table properties of a repository item are loaded when getItem is called
for the first time on the item.

You might need to modify this default property fetching behavior. For some applications, the database
activity required to load all primary table properties can adversely affect performance unnecessarily. For
example, an application may want the SQL repository to load a large GIF property only if it is specifically
asked for. This is known as lazy evaluation. On the other end of the spectrum, you might need to load
properties from different tables immediately. For example, an application may want to always load a
user’s last name whenever a profile is read from the database. This is known as prefetching. Finally, some
applications want to group properties so when one value is requested, all values in this group are
loaded—for example, loading a zip code and state code whenever a street address is loaded.

You can achieve a finer level of control over property loading by using cache groups in your repository
definition. By default, the cache group of a property is the same name as the table that the property is
defined in. You can set the cache group of a property with the group attribute in the property’s definition
in the repository definition file. All properties with the same group attribute are fetched whenever any
member of the group is fetched. Only those properties that are in the same cache group as the repository
ID (or, if there is no ID property, all the properties in the primary table) are loaded when getItem is called
for the first time on an item. While generally you define a cache group with the group attribute in the
property’s definition in the repository definition file, you can also define a cache group with the setGroup
method of atg.adapter.gsa.GSAPropertyDescriptor.

For example, an address might be composed of several properties, like this:

```xml
<property name="address1" group="address"/>
<property name="city" group="address"/>
<property name="state" group="address"/>
<property name="zip" group="address"/>
```
The `group="address"` attribute ensures that the whole address is loaded whenever one element of the address is accessed, even if the properties are stored on different database tables. So, if you call `getPropertyValue` for the `city` property, the `address1`, `state`, and `zip` properties are also loaded.

If you want to assure that only the repository ID is returned and none of the repository item's other properties, you can isolate the repository ID in its own cache group:

```xml
<item-descriptor name="user" default="true">
    <table name="usr_tbl" type="primary" id-column-names="id">
        <property name="id" data-type="string" group="id"/>
        <property name="name" data-type="string" group="info"/>
        <property name="age" data-type="int" group="info"/>
    </table>
</item-descriptor>
```

## Handling Large Database Columns

If your SQL repository definition includes properties that might correspond to large objects in the database, you might need to set some properties in the SQL Repository component to handle them.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>useSetBinaryStream</code></td>
<td>If <code>useSetBinaryStream</code> is set to <code>true</code>, the SQL repository always uses <code>setBinaryStream()</code> instead of <code>setBytes()</code> in prepared statements. The <code>setBinaryStream()</code> method is required for large byte arrays in some JDBC drivers.</td>
</tr>
<tr>
<td><code>useSetUnicodeStream</code></td>
<td>If <code>useSetUnicodeStream</code> is set to <code>true</code>, the SQL repository always uses <code>setUnicodeStream()</code> instead of <code>setString()</code> in prepared statements. The <code>setUnicodeStream()</code> method is required for large Strings in some JDBC drivers. Setting <code>useSetUnicodeStream=&quot;true&quot;</code> is recommended if you use Oracle with internationalized content, but is not recommended if you do not have internationalized content in your database. Note that if you use MS SQL Server, you must set useSetUnicodeStream to false.</td>
</tr>
<tr>
<td><code>useSetAsciiStream</code></td>
<td>If <code>useSetAsciiStream</code> is set to <code>true</code>, the SQL repository always uses <code>setAsciiStream()</code> instead of <code>setString()</code> in prepared statements. You can use <code>setAsciiStream</code> instead of <code>setUnicodeStream</code>, but you lose the ability to handle internationalized values in the database.</td>
</tr>
<tr>
<td><code>useSetObject</code></td>
<td>If <code>useSetObject</code> is set to <code>true</code>, the SQL repository always uses <code>setObject()</code> instead of <code>setInt()</code>, <code>setFloat()</code>, <code>setDouble()</code>, or <code>setString()</code> in prepared statements.</td>
</tr>
</tbody>
</table>
The SQL repository adds a number of features to the basic query architecture of the Repository API described in the Repository Queries chapter. This chapter discusses the following topics:

- Repository Filtering
- Overriding RQL-Generated SQL
- Parameterized Queries
- Named Queries
- Text Search Queries
- Wildcards in Queries
- Not Queries and Null Values
- Outer Joins
- Table Ownership Issues
- Unsupported Queries in the SQL Repository

Repository Filtering

The SQL repository lets you filter database read operations. For example, you might want a database lookup always to return only items whose `activeFlag` property is `true`. You can filter the repository by defining a repository query that specifies the filter criteria you want and associating it with the appropriate item descriptor. The filter is automatically applied to all of the following operations:

- `Repository.getItem()`
- `Repository.getItems()`
- `MutableRepository.getItemForUpdate()`
- `MutableRepository.getItemsForUpdate()`
- `RepositoryView.executeQuery()`
- `RepositoryView.executeCountQuery()`

You can define a repository filter in three ways:

- Use the `<rql-filter>` tag in the definition file for an item descriptor.
- Set the `filterQuery` property of the item descriptor to a Query object.
- Set the `rqlFilterString` property of the item descriptor to an RQL string, which is compiled into the Query object that defines the filter.

In most cases, the first method, using the `<rql-filter>` tag, is easiest and preferable.

**<rql-filter>**

You can create a repository filter with the `<rql-filter>` tag in the definition file for an item descriptor. The `<rql-filter>` tag encloses an RQL statement, as in the following example:

```xml
<item-descriptor name="article">
  <rql-filter>
    <rql>name starts with "n"</rql>
  </rql-filter>
  <table name="article" id-column-names="article_id">
    <property name="name"/>
    <property name="date"/>
  </table>
</item-descriptor>
```

This setting causes queries and item lookups for this item descriptor to return only items whose `name` property starts with `n`. The SQL repository issues SQL in the form of an extra WHERE clause condition to implement filtering so any given query or item lookup should be no slower with a reasonable filter tacked on.

You can also use RQL substitution parameters in your filter query. For example:

```xml
<item-descriptor name="article">
  <rql-filter>
    <rql>name starts with ?0 or availabilityDate &lt; ?1</rql>
    <param value="n"></param>
    <param bean="/myApp.IssueDate"></param>
  </rql-filter>
  <table name="article" id-column-names="article_id">
    <property name="name"/>
    <property name="availabilityDate" data-type="timestamp"/>
  </table>
</item-descriptor>
```

In this example, the RQL parameters are substituted into the query:

- The first parameter is a simple constant value. Typically it is not necessarily to substitute constant values as they can be inlined in the RQL query string.
- The second parameter is a Nucleus component. If an object of type `atg.service.util.CurrentDate` is used as a parameter (as in this example), the
filtering logic calls `getTimestamp()` on that object and uses that as the value of the parameter. This lets you have a `CurrentDate` service used in a filter. Also note, as in this example, that the “less than” symbol (<) is a markup character and must be escaped in your XML file as &lt;.

For information about RQL, see the Repository Query Language section in the Repository Queries chapter.

**filterQuery and rqlFilterString Properties**

In the great majority of cases, it is easiest to set a filter query with the `<rql-filter>` tag. However, there are two other ways of defining the filter used by an item descriptor. You might want to use one of these techniques if you need to set the filter query at runtime:

- Set the `filterQuery` property of the item descriptor to a Query created by the same repository. Do this by creating a Query object and calling `GSAItemDescriptor.setFilterQuery()` on it.
- Set the item descriptor's `rqlFilterString` to an RQL string that expresses the query. If the `filterQuery` property of the item descriptor is null, the SQL repository tries to use the `rqlFilterString` and compile it into the filter query. If both properties are null, filtering is disabled.

**Note:** For best performance, two constraints apply:

- `filterQuery` or `rqlFilterString` should refer only to properties in the primary table for the item descriptor. Otherwise, joins are required for every item access, which can dramatically degrade repository performance.
- Do not change the filter too often—more than once or twice a day. Each query executed by the SQL repository is AND’d to the filter query before being executed (or looked up in the cache). Too frequent changes to the filter or the RQL filter parameters diminish effectiveness of the query cache.

**Overriding RQL-Generated SQL**

The SQL created through RQL may in some cases not be optimized for your needs. Should such a situation arise, you can use the `atg.adapter.gsa.query.SqlPassthroughQuery` class. A `SqlPassthroughQuery` is used with the `QueryBuilder` to specify the SQL statement to pass directly to the database. Note that the SQL repository cannot use the results of arbitrary queries to generate repository items. The query must return the ID column or columns in their declared order from the item descriptor's `id-column-names` attribute.

Here is an example of how a `SqlPassthroughQuery` might be used in a page. Note that the SQL statement is sent to the database “as is.”

```java
import atg.adapter.gsa.query.*;

GSARepository repo =
```
(GSARepository) request.resolveName("/examples/TestRepository");
RepositoryView view = repo.getView("canard");
Object params[] = new Object[4];
   params[0] = new Integer (25);
   params[1] = new Integer (75);
   params[2] = "french";
   params[3] = "greek";
Builder builder = (Builder)view.getQueryBuilder();
String str = "SELECT * FROM usr_tbl WHERE (age_col > ?0 AND age_col < ?1
   AND EXISTS (SELECT * from subjects_tbl where id = usr_tbl.id AND subject
   IN (?2, ?3)))";
RepositoryItem[] items =
   view.executeQuery (builder.createSqlPassthroughQuery(str, params));
if (items == null)
   out.println(" Is null.");
else{
   out.println(" Is not null: " + items.length + "<p>");
   for (int i = 0; i < items.length; i++){
      out.println(items[i].toString() + "<br>");
   }
}

Parameterized Queries

A parameterized query is a Repository Query that is incomplete (that is, missing some data) when it is created, and is supplied with that data when the query is executed. This is very similar to a PreparedStatement in JDBC. Parameterized queries are supported only in the SQL repository. You can substitute a parameter only for constant values, and not column specifications in a Repository Query. The use of parameters in a Query enables developers to reuse a single instance of that Query over and over again, supplying different parameter values at execution time. For example, if your goal is to create a Query like this:

```
select id from dps_user where first_name = 'keith'
```

only the value 'keith' can be parameterized; the column name first_name cannot. Sorting and range information also cannot be parameterized, so only constraints can use parameters. Furthermore, parameterized queries are used only in queries against the database; you cannot use parameterized queries against transient properties or in cases where the SQL Repository component's useDatabaseQueries property is set to false.

Parameterized Query API

The use of parameters in a Query is implemented by a basic interface, atg.repository.ParameterSupportView, which extends atg.repository.RepositoryView. The
ParameterSupportView interface provides additional method signatures to the executeQuery() methods provided in RepositoryView. This means that for every executeQuery() method, there is a similar one with an optional Object[] argument representing parameter values for any parameters in the Query that is passed in. For example:

```java
public RepositoryItem[] executeQuery(Query pQuery,
    Object[] pParameterValues)
```

Each element in the pParameterValues array corresponds to a parameter in the given Query. pParameterValues[0] corresponds to the first parameter in the Query, pParameterValues[1] corresponds to the second parameter in the Query, and so on. When you create a Query, you must remember the number of parameters and their order, especially in the case of compound queries that use AND and OR operators. You can use the Query.getQueryRepresentation() method to obtain a string of the query representation including all parameter locations.

Also, the atg.repository.QueryBuilder interface is extended by an interface named atg.repository.ParameterSupportQueryBuilder. This interface adds a single method to create a parameter QueryExpression that can be used in queries created by the ParameterSupportQueryBuilder:

```java
public QueryExpression createParameterQueryExpression()
    throws RepositoryException
```

The atg.adapter.gsa.GSAView class used by the SQL repository implements the atg.repository.ParameterSupportView interface, and the atg.adapter.gsa.query.Builder class implements the atg.repository.ParameterSupportQueryBuilder interface. This makes parameterized queries available in the SQL repository.

### Query Types that Support Parameters

Not every query type in the SQL repository can have parameters in every argument. The following QueryBuilder methods support parameters, with the specified limitations:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter Locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>createComparisonQuery</td>
<td>Either expression or both expressions can be a parameter.</td>
</tr>
<tr>
<td>createPatternMatchQuery</td>
<td>Only the pattern argument can be a parameter.</td>
</tr>
<tr>
<td>createTextSearchQuery</td>
<td>Only the searchstring argument can be a parameter.</td>
</tr>
<tr>
<td>createIncludesQuery</td>
<td>Only the collection argument (the first argument) can be a parameter, and it must be multi-valued.</td>
</tr>
<tr>
<td>createIncludesAnyQuery</td>
<td>Either expression or both expressions can be a parameter (and they must be multi-valued in all cases).</td>
</tr>
</tbody>
</table>
QueryCache and Parameterized Queries

When a query is cached, any parameter values entered at execution time are included in the QueryCacheEntryKey, and queries with dissimilar values are cached separately. For example, if the same query is executed twice, each time with different parameter values, two entries are created in the QueryCache. If the query is executed again with parameters that were already used, the query should be found in cache.

Parameterized Query Example

The following is an example of how you might use a parameterized query. Note that error handling is not dealt with in these examples.

Suppose you wanted to create a query like this:

```
firstName = 'Jerry'
```

Then, you want to change that to

```
firstName = 'Phil'
```

Query Example without Parameters

The first example shows how to do this without the use of parameters:

```java
// Get the repository through our made up getRepository() call
Repository rep = getRepository();
RepositoryItemDescriptor desc = rep.getItemDescriptor("user");
RepositoryView view = desc.getRepositoryView();
QueryBuilder qb = view.getQueryBuilder();

// Build our first Query
// firstName = 'Jerry'
QueryExpression firstNameProp = qb.createPropertyQueryExpression("firstName");
QueryExpression jerryValue = qb.createConstantQueryExpression(new String("Jerry"));
Query firstNameQuery = qb.createComparisonQuery(firstNameProp, jerryValue, QueryBuilder.EQUALS);

// Execute our first Query
RepositoryItem[] jerryItems = view.executeQuery(firstNameQuery);

// Set up our second Query now
QueryExpression philValue = qb.createConstantQueryExpression(new String("Phil"));
firstNameQuery = qb.createComparisonQuery(firstNameProp, philValue, QueryBuilder.EQUALS);

// Execute our second Query
RepositoryItem[] philItems = view.executeQuery(firstNameQuery);
```
Query Example with Parameters

With the use of parameters in your queries, you can create a reusable Query as in the example that follows. Note that the view used is a ParameterSupportView instead of a RepositoryView:

```java
// Get the repository through our made up getRepository() call
Repository rep = getRepository();
RepositoryItemDescriptor desc = rep.getItemDescriptor("user");
// Our RepositoryView is a ParameterSupportView this time, so we know it supports
// parameters in Queries
// Note - this assumes we have advanced knowledge that this view is an instance of
// a ParameterSupportView
ParameterSupportView view = (ParameterSupportView)desc.getRepositoryView();
ParameterSupportQueryBuilder qb = view.getQueryBuilder();

// Builder our first Query up
// firstName = 'Jerry'
QueryExpression firstNameProp = qb.createPropertyQueryExpression("firstName");
QueryExpression parameterValue = qb.createParameterQueryExpression();
Query firstNameQuery = qb.createComparisonQuery(firstNameProp, parameterValue, QueryBuilder.EQUALS);

// Execute our first Query
Object[] args = new Object[1];
args[0] = new String("Jerry");
RepositoryItem[] jerryItems = view.executeQuery(firstNameQuery, args);

// Set up our second Query now
args[0] = new String("Phil");
RepositoryItem[] philItems = view.executeQuery(firstNameQuery, args);
```

The first example creates a new constant QueryExpression in order to change the name from Jerry to Phil. This also requires a new instance of a comparison Query to use the new QueryExpression. The second example increases efficiency by creating just one Query object, and changing the value of the desired name in an Object array that is passed to the executeQuery method. This also lets you cache a Query in your internal application (above the query cache layer), and pass in varying parameter values at execution time.

Named Queries

A named query is a RepositoryQuery or SQL statement that can be invoked by name. With named queries, you can reuse the same query object as often as needed. You can define named queries in an item descriptor; you can also create them in Java code through repository APIs.

Note: Named queries are supported only in SQL and Integration repositories.
**Named Queries and Item Inheritance**

Item descriptor subtypes inherit the named queries defined for their parent item types. For example, a financial application might define a user item type with investor and broker subtypes. If the user item descriptor defines a named query getUsersByLogin, the investor and broker subtypes also have access to the same named query.

**Named Queries in an SQL Repository Definition File**

You can define a named query in an `<item-descriptor>` element in an SQL repository definition file. A named query is defined in a `<named-query>` element. For example, given the following RQL query:

`lastName ENDS WITH "son"`

you can define a named query as follows:

```xml
<item-descriptor name=...>
  ...
  <named-query>
    <rql-query>
      <query-name>myQuery</query-name>
      <rql>lastName ENDS WITH "son"</rql>
    </rql-query>
  </named-query>
</item-descriptor>
```

An item descriptor can define named queries in three ways:

- **RQL named queries**
- **SQL named queries**
- **Stored procedures**

**RQL Named Queries**

An RQL named query can use all syntactical options that available to the Repository Query Language, including parameters and fields in parameters.

The RQL statement is defined in an `<rql-query>` element. For example:

```xml
<item-descriptor name=...>
  ...
  <named-query>
    <rql-query>
      <query-name>myQuery</query-name>
      <rql>name = ?0.name AND age = ?1.age</rql>
    </rql-query>
  </named-query>
</item-descriptor>
```
SQL Named Queries

An SQL named query defines a query's SQL statement in an `<sql-query>` element. SQL named queries can include database-specific keywords.

For example:

```xml
<item-descriptor name="...">
  ...
  <named-query>
    <sql-query>
      <query-name>myQuery</query-name>
      <sql>
        select id, first_name, last_name from dps_user WHERE login=?
      </sql>
      <returns>id,firstName,lastName</returns>
      <input-parameter-types>java.lang.String</input-parameter-types>
      <dependencies>login</dependencies>
    </sql-query>
  </named-query>
  ...
</item-descriptor>
```

The following table describes the child elements that are valid within an `<sql-query>` element:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sql&gt;</code></td>
<td>Contains the SQL column and table names (not the property names defined by the repository). The <code>&lt;sql&gt;</code> element must also include the ID from the primary table in the item descriptor. If the ID for an item descriptor is defined as a composite ID (using two or more columns), all columns in the composite ID must be selected. <strong>Caution:</strong> Named queries must be read-only. The SQL statement must not include actions that update the datastore—for example, INSERT or DELETE. Doing so can yield unpredictable results.</td>
</tr>
<tr>
<td><code>&lt;returns&gt;</code></td>
<td>A comma-separated list of Repository property names returned by this query. These property names let you know the type of the column when reading values from the returned ResultSet.</td>
</tr>
<tr>
<td><code>&lt;dependencies&gt;</code></td>
<td>Indicates which properties this query depends on. If any properties in the <code>&lt;dependencies&gt;</code> element change, remove this query from the query cache. These properties are typically those referenced in the SQL statement's WHERE clause.</td>
</tr>
</tbody>
</table>
<input-parameter-types>
A comma-separated list of classes that must be instantiated for query parameters. There must be one value for each parameter. For example, given three query String parameters, the <input-parameter-types> element must be set as follows:

<j ava . lang . String, java . lang . String, java . lang . String>
</input-parameter-types>

This tag serves two purposes.
- Enables type checking during query execution.
- Specifies the number of required query parameters.

The properties that are used in the <returns> element must be:

- Defined as readable in the repository.
- Persistent properties defined in a table tag and not transient properties.
- Single-valued (multi-valued properties are valid only if they are composite IDs).

The property columns are returned in the same order as specified in the <returns> tag. Because users do not need to define an explicit RepositoryPropertyDescriptor for the ID property, the ID property can be omitted from the <returns> element, but it must exist in the SQL statement as the first column(s) selected.

The <returns> element is optional: select statements can return only the ID property. The <returns> element should specify the ID property only if the item descriptor explicitly defines it with the <property> element. Otherwise, the value in the id-column-name attribute of the <table> tag is used as the name of the ID column.

Stored Procedures

An SQL named query can reference a stored procedure by qualifying the <sql> element with the attribute setting stored-procedure=true. In this case, the <returns> and <dependencies> tags must conform to the stored procedure’s returns and constraints. For example:

<i t e m - d e s c r i p t o r name=...>
...
<n a m e d - q u e r y>
<sql - query>
<query-name>myQuery</query-name>
<sql stored-procedure="true">
    { call myStoredProcedure (?, ?) }
</sql>
<returns>id, firstName, lastName</returns>
</input-parameter-types>
The body of the `<sql>` tag for a stored procedure must use the syntax required by `java.sql.CallableStatement`. The following requirements apply:

- Curly braces must enclose the `CallableStatement`.
- A `CallableStatement` typically has two formats: for procedures, which do not have an explicit return value, and one for functions, which do. Non-Oracle stored procedures in the SQL repository must use `procedure` syntax, as in the previous example.
- Question (?) marks indicate parameters. In stored procedures, parameters can be one of the following:
  - IN: Values go in the database.
  - OUT: Values come out of the database.
  - INOUT: Values can go in and come out.

The SQL repository supports only IN parameters. OUT and INOUT parameters are valid only for Oracle stored procedures.

A stored procedure must return a `java.sql.ResultSet`. Most JDBC drivers do so by default, but there are exceptions. Oracle, for example, requires some special tuning, as described in the next section.

**Caution:** The stored procedure must be read-only; it must not include actions that update the datastore—for example, `INSERT` or `DELETE`. Doing so can yield unpredictable results.

**Using Stored Procedures with Oracle Databases**

An Oracle stored procedure returns a `ResultSet` only if it is defined to do so (see *Returning a JDBC result set from an Oracle stored procedure*). The body of the `<sql>` element must reference the stored procedure with the `function` syntax. For example:

```java
{ ? = call myOracleProcedure (?, ?) }
```

You indicate the returned value through this notation:

```java
? =
```

**Note:** You should consider wrapping existing stored procedures with procedures so the results are formatted to conform with ATG SQL repository requirements.

**Java Code Access to Named Queries**

The following Java code can access and execute any parameterized SQL named query:
import atg.repository.*;
...

// somehow get item descriptor
RepositoryView view = itemDesc.getRepositoryView();
...

try {
    if(!view instanceof NamedQueryView)
        throw new ServletException
            ("view " + view.getViewName() + " is not a NamedQueryView");
    NamedQueryView nView = (NamedQueryView)view;
    Query namedQuery = nView.getNamedQuery(queryName);

    ...

    if(!view instanceof ParameterSupportView)
        throw new ServletException
            ("view " + view.getViewName() + " is not a ParameterSupportView");
    return ((ParameterSupportView)view).executeQuery(namedQuery, args);
}

catch (RepositoryException re) {
    if (pServlet.isLoggingError())
        pServlet.logError("unable to execute query due to RepositoryException",re);
    throw re;
}

Any repository that extends atg.repository.RepositoryViewImpl can access the base NamedQuery feature.

The following sections describe the named query API:

- NamedQueryView Interface
- QueryDescriptor

**NamedQueryView Interface**

Interface atg.repository.NamedQueryView provides methods to create and access named queries. This interface extends atg.repository.RepositoryView. ATG provides an implementation: the class atg.repository.RepositoryViewImpl, which supports the use of named queries.

The NamedQueryView interface includes the methods described in the following table:
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createNamedQuery</code></td>
<td><code>public void createNamedQuery(String pQueryName, Query pQuery)</code>&lt;br&gt;Create an association in the <code>RepositoryView</code> between a name and a <code>Query</code> object. After this association is created, you can call <code>getNamedQuery</code> to get the <code>Query</code> object to be used for execution. If this method is called with a <code>pQueryName</code> that is already assigned a <code>Query</code>, the existing <code>Query</code> is overwritten with the new <code>Query</code>.</td>
</tr>
<tr>
<td><code>getNamedQuery</code></td>
<td><code>public Query getNamedQuery(String pQueryName)</code>&lt;br&gt;Gets the <code>Query</code> object associated with the given name. If no such entry was created for the given String, null is returned.</td>
</tr>
<tr>
<td><code>getNamedQueryNames</code></td>
<td><code>public String[] getNamedQueryNames()</code>&lt;br&gt;Returns the names of all Named Queries that this <code>RepositoryView</code> knows about, or null if there are none.</td>
</tr>
<tr>
<td><code>getQueryDescriptor</code></td>
<td><code>public QueryDescriptor getQueryDescriptor(String pQueryName)</code>&lt;br&gt;Returns a <code>QueryDescriptor</code> object that describes aspects of the requested <code>NamedQuery</code>. If there is no named query by the given name, null is returned.</td>
</tr>
<tr>
<td><code>getQueryName</code></td>
<td><code>public String getQueryName(Query pQuery)</code>&lt;br&gt;Returns the name of the given <code>Query</code>, if any exists. Otherwise, null is returned.</td>
</tr>
</tbody>
</table>

### QueryDescriptor

Interface `atg.repository.QueryDescriptor` defines methods for an object that describes a Repository Query and associates a Repository Query object with a user-defined String. It defines the following methods:

```java
public Query getQuery()
public String getQueryName()
```

The `atg.repository.QueryDescriptorImpl` class is the base implementation of the `QueryDescriptor` interface and the `atg.adapter.gsa.query.GSAQueryDescriptor` class is the SQL repository’s subclass of `QueryDescriptorImpl`.

### Text Search Queries

The SQL repository supports full text searches via the `QueryBuilder.createTextSearchQuery()` method. In order for full text search queries to work, you must have a supported full text search engine installed and configured.

The following example demonstrates use of the full-text query feature. This class can be found in:
It must be run from DYNAMO_HOME. It also requires a repository definition file named gsa-template.xml to be in the current directory.

```java
class FullTextQuery
extends Harness {
  // ---------------------------
  /** Class version string */
  public static final String CLASS_VERSION =
  // ---------------------------

  /**
   * Run our sample.
   * @param pRepository repository to use
   * @exception RepositoryException if there is repository trouble
   **/
  public void go(Repository pRepository)
    throws RepositoryException {
    // print header
    pln("### Running Sample Full-Text Query ###
    pln(CLASS_VERSION);
    pln("\n
    /**
    * This example demonstrates how do perform some simple full-text repository
    * queries. In the repository API all queries are performed using Query
    * or QueryExpression objects. A QueryExpression is a building block you
    * can use to create simple or complex queries. A Query is a repository
    * query that can be executed. A Query can also be used as a building
    * block to create more complicated queries. Here we perform a simple
    * query to find user repository items whose story property
    * includes text in which the word 'dog' appears within 10 words of the
    * word 'cat'.
    */
    // queries are created using QueryBuilders and executed by
    // RepositoryViews. A Query is defined in the context of a specific item
    // descriptor and thus must be built and executed with the right
    // QueryBuilder and RepositoryView.
    RepositoryItemDescriptor userDesc = pRepository.getItemDescriptor("user");
    RepositoryView userView = userDesc.getRepositoryView();
```
QueryBuilder userBuilder = userView.getQueryBuilder();

// create a QueryExpression that represents the property, story
QueryExpression comment =
    userBuilder.createPropertyQueryExpression("story");

// create a QueryExpression that represents a search expression
// using the NEAR operator.
QueryExpression dogNearCat =
    userBuilder.createConstantQueryExpression("NEAR((dog, cat), 10)");

    // define the format being used by the search expression
    // appropriate to the database being used. This assumes an Oracle
    // database with the interMedia/Context full-text search option
    // installed.
    QueryExpression format =
        userBuilder.createConstantQueryExpression("ORACLE_CONTEXT");

    // pick a minimum required score that the results must meet or exceed
    // in order to be returned by the full-text search engine.
    // See your search engine vendor's docs for more information on the meaning
    // and use of the score value.
    QueryExpression minScore =
        userBuilder.createConstantQueryExpression(new Integer(1));

    // now we build our query: comment contains 'dog' within 10 words of 'cat'
    Query dogTenWordsNearCat =
        userBuilder.createTextSearchQuery(comment, dogNearCat, format, minScore);

    // finally, execute the query and get the results
    RepositoryItem[] answer = userView.executeQuery(dogTenWordsNearCat);

    pln("running query: story contains 'dog' within 10 words of 'cat'");
    if (answer == null)
    {
        pln("no items were found");
    }
    else
    {
        for (int i=0; i<answer.length; i++)
            pln("id: " + answer[i].getRepositoryId());
    }

    //-------------------------------------
    /**
    * Main routine. This example uses no command line arguments
    **/
public static void main(String[] pArgs)
    throws Exception
{
    runParser(FullTextQuery.class.getName(), pArgs);
}
} // end of class FullTextQuery

You can specify what properties a text search query should search, with the text-search-properties attribute in the <item-descriptor> tag that defines an item type. For example, the following value indicates that a text search should examine the keywords and content properties for matches:

```xml
<item-descriptor name="newsItems
text-search-properties="keywords, content">
```

Simulating Text Search Queries

As a convenience feature, the SQL repository can simulate full text searches with the SQL LIKE operator. If full text searching is not available for your database, you can substitute pattern matching queries for text search queries by setting the following property in the GSARepository component:

`simulateTextSearchQueries=true`

The SQL repository converts text search queries into CONTAINS pattern match queries, which are implemented with the SQL LIKE operator.

Simulated text search queries are useful for demos and standalone development when one wants to put in place the createTextSearchQuery() API calls without having to set up a text search engine. However, simulated text queries are extremely inefficient and are not supported for production systems. A simulated text search query with LIKE typically causes a table scan, so avoid simulated queries in production.

Wildcards in Queries

Databases often treat `%` and `_` as wildcard characters. Pattern-match queries in the SQL repository (such as CONTAINS, STARTS WITH, or ENDS WITH), assume that a query that includes `%` or `_` is intended as a literal search including those characters and is not intended to include wildcard characters. The query generated therefore uses an escape character in front of the characters `%` and `_` in pattern-match queries. One exception applies: a pattern-match query can be used to simulate a text search query; in that case wildcards should be passed through.

You can disable this behavior by setting the escapeWildcards property of the SQL Repository component to `false`.
The escape character is \ (backslash) by default. You can set a different escape character through the **wildcardEscapeCharacter** property of the SQL repository component.

**Not Queries and Null Values**

Comparison and pattern-match repository queries do not return items where the property queried is null. The following queries are the operators of the comparison or pattern-match queries that exhibit this behavior:

=, !=, <, <=, >, >=, CONTAINS, STARTS_WITH, ENDS_WITH

For example, if your query is `balance = 101` or `balance < 101`, the query does not return an item whose `balance` property is null. However, if your query is `balance != 101`, the query still does not return an item whose `balance` property is null.

If you wish your query to return items whose queried property is null, you may use an IS NULL query, or an IS NULL clause as part of an OR query, for example:

`balance != 101` OR `(balance IS NULL)

**Outer Joins**

By default, the SQL repository uses outer joins in queries that involve auxiliary tables. Different database vendors use different syntax to create outer joins. The ATG platform automatically sets the **outerJoinSupport** property of the **GSARepository** component to specify the appropriate type of outer join to be used by the SQL repository. You can also configure this property manually, using the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Database Vendor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssql92</td>
<td></td>
<td><code>USE FROM tablex x LEFT OUTER JOIN tabley y ON x.id = y.id</code></td>
</tr>
<tr>
<td>jdbc</td>
<td></td>
<td>Similar to ssql92 but uses JDBC escape syntax <code>{oj ... }</code> to tell the JDBC driver to convert to native join syntax.</td>
</tr>
<tr>
<td>plus-equals</td>
<td>Oracle</td>
<td><code>x += y</code></td>
</tr>
<tr>
<td>star-equals</td>
<td>Microsoft</td>
<td><code>x =* y</code></td>
</tr>
<tr>
<td>informix</td>
<td>Informix</td>
<td><code>FROM OUTER tablex</code></td>
</tr>
<tr>
<td>none</td>
<td></td>
<td>Use inner joins rather than outer joins.</td>
</tr>
</tbody>
</table>
Table Ownership Issues

If the user does not own the database tables used by the SQL repository, you must configure the repository so when the repository is initialized, it can determine the column types in the tables. If you have not configured the table ownership correctly, you may get an “unknown JDBC types for property” error.

The `<table>` tag supports several attributes that let you configure table ownership:

- `metaDataSchemaPattern`
- `metaDataCatalogName`
- `tablePrefix`
- `metaDataSynonymTableName`

`metaDataSchemaPattern`

Specifies the name of the database account used to create the tables that underlie the repository.

`metaDataCatalogName`

Specifies a catalog name. If the user does not own the table to be used by the repository, this attribute can be used once during repository initialization in a call to determine the column types.

`tablePrefix`

If the user does not own the table used by the repository, the `tablePrefix` lets you construct a qualified table name. This attribute is not used during the initial metadata query, but if present is prepended to the table name when inserts or updates are made. For example:

```xml
<attribute name="tablePrefix" value="myPrefix."/>
```

For instance, the following snippet sets `dps_user` to use `testing2` as the schema name for the metadata call. The string `testing2.` prepended to the table name for all other queries.

```xml
<gsa-template>
  ...
  <table name="dps_user" type="primary" id-column-name="id">
    <attribute name="tablePrefix" value="testing2."/>
    <attribute name="metaDataSchemaPattern" value="testing2"/>
  </table>
  ...
</gsa-template>
```

`metaDataSynonymTableName`

If a database schema uses a synonym to reference a table in another schema, and the synonym and table names are different, the `<table>` tag must set the `metaDataSynonymTableName` attribute to the source table’s name. On startup, a repository uses this attribute to identify the source table so it can obtain the database metadata and validate the table definition.
Constraints

In using these attributes, be sure use the same case (upper, lower, mixed) that your database uses to store object identifiers. For example, Oracle stores its identifiers in uppercase. So, you set `metaDataSchemaPattern` to `DYNAMO` instead of `dynamo`. See the Javadoc for `java.sql.DatabaseMetaData.getColumns()` for more information. See also the Logging and Data Collection chapter of the ATG Programming Guide for more information about these attributes.

Setting Ownership at the Repository Level

You can specify ownership for all repository tables by setting the SQL repository component properties `metaDataSchemaPattern`, `metaDataCatalogName`, and `tablePrefix`. These settings can be overridden by individual tables that set the corresponding attributes.

Unsupported Queries in the SQL Repository

The SQL repository does not support queries of the following types:

- `includesAll`
- `elementAt`
- `indexOf`
9 Localizing SQL Repository Definitions

You can use Java resource bundles to make it easier to localize an SQL repository definition. By using resources for repository item properties used in the ATG Control Center's Repository Editors, you can display labels and values suitable for the Repository Editor user's locale.

The SQL repository localization feature enables you to use resource files to localize:

- The values of the display-name and description of both item descriptors and properties
- The category of properties
- The strings used for representing values of enumerated property types

Defining a Resource Bundle

To localize these values, first you must associate a resource bundle using an `<attribute name="resourceBundle".../>` tag like:

```
<attribute name="resourceBundle" value="resourceBundleName"/>
```

You can use the `<attribute>` tag to set the resource bundle at the property, table, or item descriptor level. A property uses its own resourceBundle attribute if it is set. If not, it looks for a resourceBundle attribute set in its `<table>` tag, then for a resourceBundle attribute set in its `<item-descriptor>` tag. For example:

```
<item-descriptor name="user"....>
  <attribute name="resourceBundle" value="atg.userprofiling.ProfileResources"/>
...```

If you use xml-combine="append" to add properties to an item descriptor defined in a different configuration layer, do not set the resourceBundle attribute in the item descriptor, as it overwrites the setting of resourceBundle in the other configuration level. Set the resourceBundle at the table or property level instead.
Localizing Properties

To localize labels used in a Repository Editor, use the localizable attributes, as follows:

<table>
<thead>
<tr>
<th>Standard Label Attribute</th>
<th>Localizable Label Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>display-name</td>
<td>display-name-resource</td>
</tr>
<tr>
<td>description</td>
<td>description-resource</td>
</tr>
<tr>
<td>category</td>
<td>category-resource</td>
</tr>
</tbody>
</table>

For example, to localize the `display-name`, use the `display-name-resource` attribute instead of the `display-name` attribute:

```xml
<item-descriptor name="user" ...
  display-name-resource="itemDescriptorUser">
  <attribute name="resourceBundle"
    value="atg.userprofiling.UserProfileTemplateResources"/>
</item-descriptor>
```

Then, for each locale you want to support, create resource bundle properties files for each repository definition. Each resource bundle consists of a list of keys defined in the resource label attributes, with the localized value.

The `UserProfileTemplateResources.properties` resource bundle referred to in the preceding example contains this entry:

`itemDescriptorUser=User`

Localizing Enumerated Properties

You can also localize the string values that correspond to each option value in an enumerated property, with the `resource` attribute in the `<option>` tag. As with label attributes, a localized enumerated property needs to have a resource bundle defined for it at the property, table, or item descriptor level. Then, you can specify the resource key that holds the localized string value with the `resource` attribute, as in this example:

```xml
<property name="emailStatus" ... data-type="enumerated" ...>
  <option resource="emailStatusUnknown" code="0"/>
  ...
</property>
```

When you specify a default, use the resource name as the value, such as:
Use caution when localizing the strings used for enumerated types. If you have `useCodeForValue` set to `true`, calling `getPropertyValue` does not return the localized property value. To display the localized value on a page, include the localized string in your page, using a `Switch` servlet bean to choose the proper value.

For more information about resource bundles and localization, see the *Internationalizing a Dynamo Web Site* chapter in the *ATG Programming Guide*. 

```xml
<property name="emailStatus" data-type="enumerated">
  <default>emailStatusUnknown</default>
  <attribute name="useCodeForValue" value="false"/>
  <option resource="emailStatusUnknown" code="0"/>
  <option resource="emailStatusValid" code="1"/>
  <option resource="emailStatusInvalid" code="2"/>
</property>
```
10 SQL Repository Caching

Efficient database access is important to many ATG applications. You should design an application so it requires minimal access to the database and ensures data integrity. An intelligent caching strategy is central to achieving these goals. The following sections describe how to use SQL repository caches:

- Item and Query Caches
- Caching Modes
- Simple Caching
- Locked Caching
- Distributed Caching Modes
- Distributed TCP Caching
- Distributed JMS Caching
- Distributed Hybrid Caching
- Cache Configuration
- Monitoring Cache Usage
- Caching by Repository IDs
- Restoring Item Caches
- Preloading Caches
- Enabling Lazy Loading
- Cache Flushing

Item and Query Caches

For each item descriptor, an SQL repository generally maintains two caches:

- Item caches
- Query caches

Note: Item descriptors within an inheritance tree share the same item cache. For more information, see Cache Configuration.
Item Caches

Item caches hold the values of repository items, indexed by repository IDs. Item caching can be explicitly enabled for each item descriptor. Even if caching is explicitly disabled, item caching occurs within the scope of each transaction (see Disabling Caching).

An item cache entry is invalidated when that item is updated. The scope of an entry's invalidation depends on its caching mode. For example, when an item is changed under simple caching mode, only the local cache entry is invalidated; other ATG instances are not notified. ATG provides several different distributed caching modes to invalidate items across multiple instances.

Query Caches

Query caches hold the repository IDs of items that match given queries. When a query returns repository items whose item descriptor enables query caching, the result set is cached as follows:

- The query cache stores the repository IDs.
- The item cache stores the corresponding repository items.

Subsequent iterations of this query use the query cache's result set and cached items. Any items that are missing from the item cache are fetched again from the database.

Query caching is turned off by default. If items in your repository are updated frequently, or if repeated queries are rare, the benefits of query caching might not justify the overhead that is incurred by maintaining the cache.

A query cache entry can be invalidated for two reasons:

- A cached item property that was specified in the original query is modified.
- Items of a queried item type are added to or removed from the repository.

Note: Queries that include derived item properties are never cached.

Caching Modes

The SQL repository supports the following caching modes:

- **Simple caching** handles caches in each server locally; no attempt is made to synchronize updates across multiple server instances.
- **Locked caching** uses read and write locks to synchronize access to items stored by multiple caches.
- **Distributed TCP caching** uses TCP to broadcast cache invalidation events to all servers in a cluster.
- **Distributed JMS caching** uses JMS to broadcast cache invalidation events to all servers in a cluster.
- **Distributed hybrid caching** uses TCP to send cache invalidation events only to those servers that are known to cache the target items.
Setting Caching Mode

Caching modes are set at the item descriptor level, through the `<item-descriptor>` tag's `cache-mode` attribute. The default caching mode is `simple caching`. To set a different caching mode on an item descriptor, set `cache-mode` to one of the following values:

- `simple`
- `locked`
- `distributed`
- `distributedJMS`
- `distributedHybrid`

Disabling Caching

When caching is disabled for an item, its property values are cached only during the current transaction, and only if the transaction requires one or more of that item's properties. This ensures a consistent view of item data while the transaction is in progress. Thus, multiple calls to `getPropertyValue()` for the same property within the same transaction require only one database query. Cached item properties are reloaded from the datastore for each transaction.

Caching should generally be disabled for application data that is exposed to changes by non-ATG repository applications—for example, online banking data, where caching might need to be disabled in order to ensure display of up-to-date user account balances. In some circumstances, you might configure integration so the repository cache is invalidated when data is modified by an external application.

You can disable caching for items of a specified type, or for specific item properties:

- To set an item type's caching mode, set its `<item-descriptor>` tag's `cache-mode` attribute to `disabled`.
- To disable caching for an individual property within an item descriptor, set the `<property>` tag's `cache-mode` attribute to `disabled`. Each property's definition overrides the caching mode set for its item descriptor. For example:

```xml
<item-descriptor name="user" cache-mode="simple">
  <table name="dps_user">
    <property name="password" cache-mode="disabled">
      ...
    </property>
    ...
  </table>
  ...
</item-descriptor>
```

**Caution:** If caching is disabled for an item type or individual item properties, any code that retrieves that item requires access to the database, which can noticeably degrade application performance.

Global disabling

You can globally set item and query cache sizes to 0 for the entire repository, which effectively disables caching; this is typically done for debugging purposes only. In order to set cache sizes to 0 on application startup, set two SQL Repository component properties to true:
• `disableItemCachesAtStartup` disables all item caches by setting their size to 0.

• `disableQueryCachesAtStartup` disables all query caches by setting their size to 0.

**Inherited Caching Modes**

You can set a property to inherit the default caching mode by setting its `cache-mode` attribute to `inherit`. This setting can be useful when a property's caching mode is set to disabled at one point in the configuration path and you want to reset the property to the default caching mode at a later point.

**Simple Caching**

When caching mode is set to simple, each server maintains its own cache in memory. A server obtains changes to an item’s persistent state only after the cached entry for that item is invalidated. This mode is suitable for read-only repositories such as product catalogs, where changes are confined to a staging server, and for architectures where only one server handles a given repository item type.

You can ensure that an item’s cached data is regularly refreshed by setting its item descriptor’s `item-cache-timeout` attribute. This approach can prevent stale data from accumulating in the item cache, and avoids the overhead of other caching modes. For many multi-server sites, setting this attribute to a low threshold, such as one minute, incurs only a low-risk delay in caching the latest data. See Cache Timeout for more information.

**Locked Caching**

A multi-server application might require locked caching, where only one ATG instance at a time has write access to the cached data of a given item type. You can use locked caching to prevent multiple servers from trying to update the same item simultaneously—for example, Commerce order items, which can be updated by customers on an external-facing server and by customer service agents on an internal-facing server. By restricting write access, locked caching ensures a consistent view of cached data among all ATG instances.

**Prerequisites**

Locked caching has the following prerequisites:

- Item descriptors that specify locked caching must disable query caching by setting their `query-cache-size` attribute to 0.

- A repository with item descriptors that use locked caching must be configured to use a `ClientLockManager` component; otherwise, caching is disabled for those item descriptors. The repository’s `lockManager` property is set to a component of type `atg.service.lockmanager.ClientLockManager`.

- At least one `ClientLockManager` on each ATG instance where repositories participate in locked caching must be configured to use a `ServerLockManager`. 
- A **ServerLockManager component** must be configured to manage the locks among participating ATG instances.

**ClientLockManager Component**

If an SQL repository contains item descriptors that use locked caching, set the Repository component's `lockManager` property to a component of type `atg.service.lockmanager.ClientLockManager`. ATG provides a default ClientLockManager component:

```
/atg/dynamo/service/ClientLockManager
```

Thus, you can set an SQL Repository's `lockManager` property as follows:

```
lockManager=/atg/dynamo/service/ClientLockManager
```

**ClientLockManager Properties**

A ClientLockManager component must be configured as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>useLockServer</code></td>
<td><code>true</code> enables this component to connect to a ServerLockManager</td>
</tr>
<tr>
<td><code>lockServerAddress</code></td>
<td>Host address of the ServerLockManager and, if set, the backup ServerLockManager</td>
</tr>
<tr>
<td><code>lockServerPort</code></td>
<td>The ports used on the ServerLockManager hosts, listed in the same order as <code>lockServerAddress</code></td>
</tr>
</tbody>
</table>

For example, given two ServerLockManagers on hosts `tartini.acme-widgets.com` and `corelli.acme-widgets.com`, where both use port 9010, the ClientLockManager is configured as follows:

```
$class=atg.service.lockmanager.ClientLockManager
lockServerAddress=tartini.acme-widgets.com,corelli.acme-widgets.com
lockServerPort=9010,9010
useLockServer=true
```

**Note:** The `liveconfig` configuration layer always sets `useLockServer` to true.

**ServerLockManager Component**

The server lock manager synchronizes locking among various ATG servers, so only one at a time can modify the same item. At least one ATG server must be configured to start the `/atg/dynamo/service/ServerLockManager` component on application startup.
To do this, add the ServerLockManager to the `initialServices` property of `/atg/dynamo/service/Initial` in the ServerLockManager server’s configuration layer. For example:

```
<ATG10dir>/home/localconfig/atg/dynamo/service/Initial.properties
```

This properties file sets the `initialServices` property as follows:

```
#/home/localconfig/atg/dynamo/service/Initial.properties:
initialServices+=ServerLockManager
```

### ServerLockManager Failover

You can configure two ServerLockManagers, where one acts as the primary lock server and the other serves as backup. The primary ServerLockManager is determined by a string comparison of two lock server property settings, `lockServerAddress` and `lockServerPort`, where the server with the lower string value is designated as the primary ServerLockManager.

On detecting failure of the primary ServerLockManager, the backup ServerLockManager takes over and clients redirect lock requests to it. If both ServerLockManagers fail, caching is disabled and all data is accessed directly from the database. Caching resumes when the one of the ServerLockManagers restarts.

### ServerLockManager Properties

A ServerLockManager component is configured with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>port</code></td>
<td>This server’s port</td>
</tr>
<tr>
<td><code>otherLockServerAddress</code></td>
<td>The other ServerLockManager’s address</td>
</tr>
<tr>
<td><code>otherLockServerPort</code></td>
<td>The port of the ServerLockManager specified in <code>otherLockServerAddress</code></td>
</tr>
<tr>
<td><code>otherServerPollInterval</code></td>
<td>The interval in milliseconds that this server waits before polling the server specified in <code>otherLockServerAddress</code></td>
</tr>
<tr>
<td><code>waitTimeBeforeSwitchingFromBackup</code></td>
<td>The time in milliseconds that this server waits after detecting that the primary ServerLockManager has failed, before taking over as the primary ServerLockManager</td>
</tr>
</tbody>
</table>

For example, given ServerLockManagers `tartini.acme-widgets.com` and `corelli.acme-widgets.com` running on port 9010, their respective configurations might look like this:
# tartini:9010
$\texttt{class}=$atg.service.lockmanager.ServerLockManager 
$\texttt{port}=9010$
\texttt{otherLockServerAddress}=corelli.acme-widgets.com 
\texttt{otherLockServerPort}=9010 
\texttt{otherServerPollInterval}=2000 
\texttt{waitTimeBeforeSwitchingFromBackup}=10000

# corelli:9010
$\texttt{class}=$atg.service.lockmanager.ServerLockManager 
$\texttt{port}=9010$
\texttt{otherLockServerAddress}=tartini.acme-widgets.com 
\texttt{otherLockServerPort}=9010 
\texttt{otherServerPollInterval}=2000 
\texttt{waitTimeBeforeSwitchingFromBackup}=10000

## Running ServerLockManager and Page Servers
A ServerLockManager should not run in the same ATG instance as a page server that handles user sessions. By running the ServerLockManager in a separate ATG instance, the overhead of managing locks has no impact on user sessions; and the page server can restart independently of the ServerLockManager.

## Processing Lock Requests
When an item type's caching mode is set to locked, the repository must obtain read or write locks to items before it can make them accessible to an application. Items that use locked caching mode can have one writer or multiple readers at a time.

### Write Lock Request
When an application updates an item, the item’s repository requests a write lock from its ClientLockManager. If another transaction on the same ATG instance owns a write lock on the same item, the ClientLockManager can transfer the write lock, when available, to the pending request. Otherwise, it relays the request to its ServerLockManager, which determines whether it has any lock entries for the item. The ServerLockManager then processes the lock request as follows:

<table>
<thead>
<tr>
<th>Existing lock entry</th>
<th>ServerLockManager action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Grants write lock request, creates write lock entry for the item.</td>
</tr>
<tr>
<td>Read</td>
<td>Asks ClientLockManagers to release their read locks on the item. When all read locks are released, grants the write lock request and creates a write lock entry for the item.</td>
</tr>
</tbody>
</table>
### Existing lock entry | ServerLockManager action
--- | ---
Write | Denies write lock request and adds the request to the queue of other pending requests for the item. Grants the request when the item becomes available.

**Note:** A write lock request always has precedence over pending read lock requests.

### Read Lock Request
When an application looks up an item, the item's repository requests a read lock from its ClientLockManager. The ClientLockManager relays this request to its ServerLockManager, which determines whether it already has any lock entries for the item. It then processes the lock request as follows:

<table>
<thead>
<tr>
<th>Existing lock entry</th>
<th>ServerLockManager action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Grants read lock, creates read lock entry for the item.</td>
</tr>
<tr>
<td>Read</td>
<td>Grants read lock, adds this ClientLockManager to list of read locks for the item.</td>
</tr>
<tr>
<td>Write</td>
<td>Denies read lock and adds the read lock request to the queue of other pending read requests for the item. Grants the request when the item becomes available.</td>
</tr>
</tbody>
</table>

### Processing Competing Lock Requests
When a transaction releases a write lock and multiple lock requests are pending for the locked item, the requests are processed in the following order:

1. If any write lock requests are pending on the local ATG instance for the item, the ClientLockManager transfers the write lock to one of them.
2. If no write lock requests are pending locally for the item, the ClientLockManager checks whether any lock requests are pending on other ATG instances; if so, it releases the lock to the ServerLockManager so it can grant one of those requests.
3. If no remote lock requests are pending, the ClientLockManager checks whether any local read lock requests are pending; if so, it releases the write lock and grants one of those requests.

### Lock Lifespan
Lock ownership information is cached on the ClientLockManager as long as the item itself remains in the item cache. The lock can be passed among different transactions or threads on the ATG instance without informing the ServerLockManager.
A read lock remains valid until one of the following events occurs:

- The item is removed from the item cache.
- The ServerLockManager requests that the lock be released—for example, in response to a write lock request.

Isolation Levels

When an item type's cache-mode is set to locked, ATG uses read and write locks to control which threads can access and change items of that type. The exact behavior depends on how you set the isolation level for the item descriptor.

To minimize deadlocks when you use locked caching, configure item descriptors to use one of the following repository isolation levels:

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readCommitted</td>
<td>ATG obtains a read lock on an item when a transaction calls getItem or getPropertyValue. If the transaction tries to update the item, ATG releases the read lock and tries to acquire a write lock.</td>
</tr>
<tr>
<td>repeatableRead</td>
<td>ATG obtains a read lock on an item when a transaction first calls getItem or getPropertyValue. If the transaction tries to update the item, ATG tries to convert the read lock into a write lock. Unlike readCommitted, the repeatableRead isolation level prevents another transaction from obtaining a write lock on the item.</td>
</tr>
<tr>
<td>serializable</td>
<td>Prevents different transactions from reading an item at the same time, whether from the same server, or from different servers.</td>
</tr>
</tbody>
</table>

You set an item type's isolation level in its <item-descriptor> tag as in the following example:

```xml
<item-descriptor name="myItem" cache-mode="locked">
  <attribute name="isolationLevel" value="readCommitted"/>
 ...
</item-descriptor>
```

Locking Exceptions

Attempts to obtain an item lock can yield one of the following exceptions:
### Exception and Cause

<table>
<thead>
<tr>
<th>Exception</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>RepositoryException</td>
<td>Deadlock occurs on calls to <code>getItem</code> or <code>getItemForUpdate</code>. This exception is a wrapper for <code>DeadlockException</code>.</td>
</tr>
<tr>
<td>ConcurrentUpdateException</td>
<td>The isolation level is set to <code>repeatableRead</code> and the transaction cannot convert a read lock to a write lock—typically because another transaction is trying to obtain a write lock at the same time, or is also trying to convert a read lock to a write lock.</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>A transaction calls a method such as <code>setProperty Value</code> that does not throw a <code>RepositoryException</code>. This exception is a wrapper for <code>RepositoryException</code>, which wraps the pertinent exception.</td>
</tr>
</tbody>
</table>

### Resolving Lock Contention

If excessive lock contention causes the lock server to become a bottleneck, you can distribute the load by instantiating multiple server lock managers to handle competing lock requests. For example, the following diagram shows a site with the following setup:

- Repositories A, B, Y, and Z, where A and B each have two instances on different ATG instances
- Four client lock managers, where the client lock managers for repositories A and B reference server lock manager SLM_AB, and client lock managers for repositories Y and Z reference server lock manager SLM_YZ.
- Two server lock managers that handle different sets of lock requests: SLM_AB and SLM_YZ
In a distributed application where the same repository—for example, the ProfileAdapterRepository—runs on multiple ATG instances, all repository instances must use the same ServerLockManager component. In the previous diagram, two instances of repository A run on separate ATG instances. Thus, their client lock managers must be set up to use the same server lock manager.

**Monitoring Lock Managers**

The Dynamo Administration Interface pages for ClientLockManager and ServerLockManager components display the state of the internal tables for each lock entry. To view this information, click Display Lock Table under Service Info.

**ClientLockManager lock table**

The ClientLockManager lock table contains these columns:

<table>
<thead>
<tr>
<th>Column heading</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>A string that identifies the locked item with this format: <code>repository-component-path:item-descriptor-name:item-id</code></td>
</tr>
<tr>
<td>Read owned</td>
<td>Set to true or false, indicates whether this client lock manager has a read lock on this item. Multiple client lock managers can simultaneously have the same read lock. One read lock on an item prevents other processes from obtaining a write lock.</td>
</tr>
<tr>
<td>Column heading</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Write owned</td>
<td>Set to true or false, indicates whether this client lock manager has a write lock on the item.</td>
</tr>
<tr>
<td>Read requested</td>
<td>Set to true or false, indicates whether a request is pending from another client to obtain a read/write lock. The ServerLockManager forwards client requests to all ClientLockManagers that own locks on the desired item. This field is set to true only when the Read owned or Write owned field is set to true and the lock request conflicts with active lock owners—for example, a write lock is requested for an item by another client when this client has a thread with active read ownership on that item.</td>
</tr>
<tr>
<td>Globally owned</td>
<td>Set to true if an active read or write lock exists that was acquired from the ServerLockManager. If the ServerLockManager is unavailable, a client can distribute locks but sets this field to false, indicating that the lock is valid only for this client.</td>
</tr>
<tr>
<td>Write owner</td>
<td>The thread that owns this lock in this client.</td>
</tr>
<tr>
<td>Read owners</td>
<td>One or more threads that own this lock in this client.</td>
</tr>
<tr>
<td>Read waiters</td>
<td>The threads that are waiting for this lock.</td>
</tr>
<tr>
<td>Write waiters</td>
<td></td>
</tr>
</tbody>
</table>

**ServerLockManager Lock Table**

The ServerLockManager lock table contains these columns:

<table>
<thead>
<tr>
<th>Column heading</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>A string that identifies the locked item with this format:</td>
</tr>
<tr>
<td></td>
<td>repository-component-path:item-descriptor-name:item-id</td>
</tr>
<tr>
<td>Write owner</td>
<td>The client that owns this lock.</td>
</tr>
<tr>
<td>Read owners</td>
<td>One or more clients that own this lock.</td>
</tr>
<tr>
<td>Read waiters</td>
<td>The client IDs that are waiting for this lock.</td>
</tr>
<tr>
<td>Write waiters</td>
<td></td>
</tr>
</tbody>
</table>

**Locking Scenarios and Workflows**

A site that runs multiple ATG applications and uses ATG scenarios or workflows should enable locked caching for item descriptors that pertain to scenarios and workflows, as described in the ATG Personalization Programming Guide. This setting is enabled when you use the liveconfig configuration layer for the DSS and Publishing modules.
Distributed Caching Modes

ATG provides three caching modes that synchronize item caches across multiple ATG instances:

- Distributed TCP caching
- Distributed JMS caching
- Distributed hybrid caching

Simple versus Distributed Caching

Simple caching mode is generally sufficient if site users do not require immediate access to recent item changes. You can use item descriptor attributes `item-expire-timeout` and `query-expire-timeout` to specify how long items can stay in the item and query caches, respectively, before they are invalidated. For many multi-server sites, setting this attribute to a low threshold provides a reasonable response time to item changes, while avoiding the network overhead incurred by distributed caching modes—especially distributed TCP caching and distributed JMS caching. See Cache Timeout for more information.

Distributed Caching Mode Options

Sites that require timely or reliable access to the latest data should use one of the distributed caching modes that ATG provides. The choice of a distributed caching mode depends on a number of requirements.

Distributed TCP

The following requirements apply:

- Real-time access to item changes
- Infrequent updates to cached items
- Modest number of item caches to monitor

Distributed JMS

The following requirements apply:

- Reliable delivery of invalidation messages
- Infrequent updates to cached items
- Large number of item caches to monitor

Distributed hybrid

The following requirements apply:

- Real-time access to item changes
- Large number of cached items to monitor across many clients
- Infrequent updates to cached items
A site that uses distributed hybrid caching must provide enough server memory to support a large GSACacheServerManager. For more information, see Optimizing Performance later in this chapter.

**Distributed TCP Caching**

If an application modifies an item whose item descriptor specifies distributed TCP caching mode, a cache invalidation event is broadcast from that ATG instance to all other ATG instances that use distributed TCP caching. The event message supplies the nature of the change, the changed item’s type, and repository ID. Receiving repositories respond by invalidating that cached item.

Distributed TCP caching is suitable for sites where the following conditions are true:

- Items of a type that specifies distributed TCP caching are likely to be cached across most ATG instances in the application.
- Items are subject to frequent reads.
- Items are rarely changed, added or deleted.

An item that changes frequently—say, more than 50 or 100 times per second—is not suitable for distributed TCP caching mode, because the extra network activity incurred by cache invalidation messages outweighs caching benefits. Cache invalidation events are broadcast to all ATG instances that enable distributed caching, even if they do not cache the invalidated item; as the number of these ATG instances increases, so too increases the network activity associated with each cache invalidation event.

**Implementation**

Distributed TCP caching is implemented by the following elements:

- Cache invalidation events of class `atg.adapter.gsa.event.GSAEvent` convey cache changes to other ATG instances, identifying the event type, item descriptor name, and repository ID.
- Event server components of class `atg.adapter.gsa.event.GSAEventServer` send and receive cache invalidation events over TCP among participating ATG instances. All repositories in an ATG instance reference the same event server. When instantiated, the event server opens up a server socket on the specified port or, if unspecified, on one that is randomly assigned.
- `das_gsa_subscriber` is a database table that maintains routing information for each item descriptor that uses distributed TCP caching. This table provides the address and port of each event server that provides access to a given item descriptor. When a change occurs on one ATG instance to an item that uses distributed TCP caching, its repository uses `das_gsa_subscriber` to look up the same item descriptors on other ATG instances. It then generates a cache invalidation event with their routing information.

The following graphic shows how updates to a cached repository item on ATG server Alamo trigger an invalidation event that is broadcast to other ATG servers, whose GSAEventServers relay it to the corresponding repositories:
Distributed TCP Caching Setup

As installed, the ATG default configuration enables event messaging for distributed TCP caching mode; you only need to ensure that the `das_gsa_subscriber` table is properly created in your database.

**Event server**

All SQL repositories in a given ATG instance set their `eventServer` property to the same event server component. The default component path is:

```
/atg/dynamo/server/SQLRepositoryEventServer
```

**Event server ports**

By default, each event server listens for cache invalidation events on a port that is randomly assigned by the operating system. Alternatively, you can set an event server component's `port` property to a permanent port number—for example, in order to bypass a firewall.

**Event server connection timeout**

Occasionally, an ATG instance's event server is inaccessible—for example, the host machine has gone down. You can control the length of time allowed to establish a socket connection to an event server through its `connectTimeout` property—by default, set to 60000 milliseconds.
If a connection attempt times out, the event server attempting to connect prints a warning message. Also, the routing information for the item descriptor on the unresponsive ATG instance is removed from das_gsa_subscriber. The event server starts a new connection attempt after the time span specified in its connectRetryTimeout property, initially set to 120000 milliseconds.

**GSAEvent timeout**

An event server determines whether other event servers are alive by their responses to the GSAEvents that it sends. If an event server does not respond within the time span specified by its sendRemoteEventTimeout property—by default, 60000 milliseconds—the sending event server assumes an invalid socket connection to the non-responding event server and drops the corresponding GSAConnection from its connection table. It then attempts to reconnect within the time span specified by its connectTimeout property.

An event server uses its sendRemoteEventTimeout property only if its enableSendRemoteEventTimeout property is set to true (the default setting).

**Restoring Subscriber Data**

When an event server is removed from das_gsa_subscriber for a given item descriptor—for example, due to a connection timeout—it can be restored in two ways:

- Each time the event server sends a cache invalidation event for an item descriptor, it checks whether it is itself listed for that item descriptor in das_gsa_subscriber. If it is missing, it prints a warning and adds itself to the table.

- The GSARepository class's checkSubscriptions() method can be called periodically on a repository—for example, by a scheduled service—to ensure that all item descriptors defined in that repository to use distributed TCP caching are registered in the das_gsa_subscriber table. checkSubscriptions() queries das_gsa_subscriber for each item descriptor. If the method finds that the repository's event server is not listed for an item descriptor, it issues a warning and flushes the item type's cache, in order to safeguard against invalid data. It also adds itself to das_gsa_subscriber.

*Note:* das_gsa_subscriber is updated with repository data only if the repository's autoUpdateSubscribers property is set to true (the default).

**Invalidating Cached Items**

When an ATG application modifies an item that uses distributed TCP caching, the following occurs:

1. When the local repository item cache is updated, the repository looks up the specified item descriptor in the das_gsa_subscriber table, and determines whether other ATG instances define this item descriptor.

2. If the item descriptor is found, the event server connects to the other event servers identified in das_gsa_subscriber.

3. An invalidation event is sent to the target event servers with the pertinent information: event type, item descriptor, and repository ID.
4. The receiving repositories invalidate caches for that item descriptor.

**Disabling Automatic Updates to das_gsa_subscriber**

You can configure an SQL repository so it does not automatically update the `das_gsa_subscriber` table, by setting the SQL Repository component's `autoUpdateSubscribers` property to `false`. In general, you do this in order to protect a live site from updates that are liable to degrade performance.

When updates to `das_gsa_subscriber` are disabled, ATG does not add or remove items from the `das_gsa_subscriber` table. In this case, it only prints warnings when an event server:

- Tries to send a cache invalidation event and cannot find itself in the table.
- Cannot send a cache invalidation event to a server that is listed in the table.

You can confirm that cache invalidation events are distributed correctly if you can modify items that use distributed TCP caching on each server and avoid any warnings.

**Populating das_gsa_subscriber**

A site that prohibits updates to `das_gsa_subscriber` assumes that the table is fully populated with valid caching data. However, this table can only be populated if updates are initially enabled. In order to populate the `das_gsa_subscriber` and disable it for subsequent updates, follow these steps:

1. On each participating ATG instance, set the event server component's `port` property to an available port.
2. For each repository that uses distributed TCP caching mode, make sure that its SQL Repository component's `autoUpdateSubscribers` property is set to `true` (the default).
3. Start all ATG instances in the cluster.
4. After all ATG instances start, the contents of `das_gsa_subscriber` should be populated with unique port numbers.
5. Dump the contents of `das_gsa_subscriber` to a backup file, so the data can be restored later.
6. Stop all ATG instances that participate in distributed TCP caching.
7. For each repository that uses distributed TCP caching mode, reset its SQL Repository component's `autoUpdateSubscribers` property to `false`.
8. Insert the dumped output into `das_gsa_subscriber`.
9. Restart the ATG instances.

**Distributed JMS Caching**

When an item descriptor's caching mode is set to `distributedJMS`, all cache invalidation events sent for items of that type use JMS, which persists cache invalidation event messages in the SQL database until delivery is complete. By contrast, `distributed TCP caching` cannot guarantee that all servers receive cache
invalidation events. For example, a server might fail to connect to another server, and there is no guarantee that all delayed events are delivered after the connection is reestablished.

**Note:** Use distributed JMS caching only for items that are infrequently updated, as its performance is much slower than using distributed TCP caching.

### Distributed JMS Caching Setup

Under distributed JMS caching, participating ATG instances act as message sources and sinks for invalidation events. As installed, ATG provides two components that are already configured as Patch Bay message sources and sinks:

- **/atg/dynamo/service/GSAInvalidatorService** is configured as a Patch Bay message source. When a repository item that uses distributed JMS caching is updated, the GSAInvalidatorService component generates a JMS cache invalidation event of class *atg.adapter.gsa.invalidator.MultiTypeInvalidationMessage*.

- **/atg/dynamo/service/GSAInvalidatorReceiver** is configured as a Patch Bay message sink and is a durable subscriber to invalidation topics. When one ATG instance generates a JMS cache invalidation event, the GSAInvalidatorReceiver on other instances receives the message and invalidates the appropriate item caches.

As installed, ATG defines the SQL JMS topics that are used by the GSAInvalidatorService and GSAInvalidatorReceiver components.

To set up distributed JMS caching:

- On each ATG instance that uses distributed JMS caching, enable the GSAInvalidatorService by configuring the **/atg/dynamo/Configuration** component as follows:
  
  ```
  gsaInvalidatorEnabled=true
  ```

- Optionally, configure each SQL Repository’s **invalidatorService** property to a GSAInvalidatorService component. If this property is not set, the repository uses the default GSAInvalidatorService component as a Patch Bay message source:

  ```
  /atg/dynamo/service/GSAInvalidatorService
  ```

- Optionally, configure each GSAInvalidatorService component’s **maxItemsPerEvent** property. This property specifies the maximum number of cached items that a single MultiTypeInvalidationMessage can invalidate—by default, 200. If an event exceeds this limit, the entire cache of the invalidated item is flushed. A value of 0 ensures that every event always flushes the entire cache of the invalidated item.

  This mechanism can significantly reduce message payload, as a message that affects large numbers of items only needs to contain information about the item types to invalidate, rather than individual items.

- Configure PatchBay in each participating ATG instance to access the same SQL JMS database tables.
Distributed Hybrid Caching

Distributed hybrid caching provides intelligent cache invalidation across multiple ATG instances. Unlike distributed JMS and TCP caching, which broadcast invalidation events to all participating servers, distributed hybrid caching sends invalidation events only to servers where the items are cached, which can significantly reduce network traffic.

Distributed hybrid caching is suitable for sites with the following requirements:

- Real-time access to item changes
- Large number of items to monitor across many clients

To achieve optimal performance, a site that uses distributed hybrid caching must provide enough server memory to support a GSACacheServerManager that can monitor all distributed hybrid items; and clients must have enough memory to cache locally all distributed hybrid items. For more information, see Optimizing Performance later in this chapter.

Distributed Hybrid Caching Setup

Distributed hybrid caching relies on four elements:

- GSACacheClientManager must be configured on each ATG instance that participates in distributed hybrid caching. It initiates invalidation events for repository items that use distributed hybrid caching, and sends event messages to the GSACacheServerManager for distribution to other ATG instances.
- GSACacheServerManager directs cache invalidation events to ATG instances that contain the affected repository items.
- ServerLockManager connects each GSACacheClientManager to the GSACacheServerManager. The ServerLockManager and GSACacheServerManager must be configured on the same ATG instance.
- An invalidation event of class GSACacheEvent conveys a cache invalidation event for repository items that use distributed hybrid caching.

GSACacheClientManager

A GSACacheClientManager must be configured on each ATG instance where one or more repository item descriptors have their caching mode set to distributedHybrid. The GSACacheClientManager has the following Nucleus component path:

/atg/dynamo/service/GSACacheClientManager

Each GSACacheClientManager must set the following properties:

lockServerAddress=host[, host]
lockServerPort=port-num[, port-num]
enabled=true
host and port-num specify the ServerLockManager’s address and port. If two ServerLockManagers are specified for failover purposes, lockServerAddress and lockServerPort must list their respective addresses and ports in the same order.

For example, given two ServerLockManagers on hosts tartini.acme-widgets.com and corelli.acme-widgets.com, where both use port 9010, you can configure a GSACacheClientManager component as follows:

```
lockServerAddress=tartini.acme-widgets.com,corelli.acme-widgets.com
lockServerPort=9010,9010
enabled=true
```

Each ServerLockManager must run in the same ATG instance as a GSACacheServerManager, and must be configured to support distributed hybrid caching. During repository startup, the GSACacheClientManager connects to the ServerLockManager, which hands over the connection to the GSACacheServerManager; this connection becomes the communication channel between the GSACacheClientManager and GSACacheServerManager for cache invalidation events.

For debugging purposes, you can also set the loggingDebug property to true.

**GSACacheServerManager**

A GSACacheServerManager maintains routing information for all cached repository items that use distributed hybrid caching. When a cache invalidation event occurs on a repository, that repository’s GSACacheClientManager sends the event to its GSACacheServerManager; the GSACacheServerManager relays this event to the appropriate clients.

The GSACacheServerManager has the following Nucleus component path:

```
/atg/dynamo/service/GSACacheServerManager
```

To enable a GSACacheServerManager, set its enabled property to true. You can also configure its defaultItemCacheSize property, which sets the maximum number of items that are mapped for each item descriptor. This property is initially set to 1000. For more information on setting this property, see Optimizing Performance later in this chapter.

Each GSACacheServerManager is associated with a ServerLockManager that runs in the same ATG instance, via the connection information specified by GSACacheClientManagers. You can provide a backup GSACacheServerManager by configuring those GSACacheClientManagers to connect to the same secondary ServerLockManager. Precedence of GSACacheServerManagers is set by their respective ServerLockManagers (see ServerLockManager failover).

For debugging purposes, you can also set the loggingDebug property to true.

**ServerLockManager**

In order to enable distributed hybrid caching, a ServerLockManager must be configured on the same instance as the GSACacheServerManager with the following setting:
handleDistributedHybridCacheEvent = true

When thus enabled, the ServerLockManager hands over the connection from a GSACacheClientManager to the GSACacheServerManager; this connection becomes the communication channel between them for all subsequent cache invalidation events.

**GSACacheEvent**

GSACacheClientManagers and GSACacheServerManager communicate with each other by passing GSACacheEvent objects, which encapsulate invalidation events with the following information:

- Repository item ID
- Item descriptor name
- Event type

The following table describes the four types of GSACacheEvents:

<table>
<thead>
<tr>
<th>GSACacheEvent type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALIDATE_CACHE_ENTRY</td>
<td>Sent when a repository item or item descriptor is invalidated. The GSACacheServerManager determines which clients cache the item; it then relays the event to their respective GSACacheClientManagers for delivery to the target repositories. If the GSACacheEvent provides a repository item ID, only the specified repository item is invalidated. If the GSACacheEvent omits a repository item ID, the entire item descriptor is invalidated, and the corresponding item caches are flushed.</td>
</tr>
<tr>
<td>LOAD_INTO_CACHE</td>
<td>Sent when a repository item is cached.</td>
</tr>
<tr>
<td>REMOVE_CACHE_ENTRY</td>
<td>Sent when a repository item is deleted. On receiving this GSACacheEvent, the GSACacheServerManager determines which clients cache the item; it then relays the event to their respective GSACacheClientManagers for delivery to the target repositories.</td>
</tr>
<tr>
<td>REMOVE_FROM_CACHE</td>
<td>Confirms that a repository item is removed from the cache. On receiving this event, the GSACacheServerManager removes the item from the list of items that it manages for that GSACacheClientManager.</td>
</tr>
</tbody>
</table>

All events except LOAD_INTO_CACHE are sent asynchronously. LOAD_INTO_CACHE is sent synchronously in order to ensure a consistent view of item data across all repositories and handle rare race conditions. The GSACacheClientManager property synchronousMessageTimeout determines the maximum amount of time each GSACacheClientManager waits for the GSACacheServerManager to reply before caching an item. This property ensures threads wait only a finite period of time for a reply from the GSACacheServerManager. The default setting is 500 milliseconds.
The following graphic shows how removal of repository item on ATG server Alamo triggers an invalidation event for all repository caches that also contain that item—in this case, on ATG server Houston.

Distributed Hybrid Caching Initialization

A distributed hybrid caching system initializes itself as follows:

1. The GSACacheClientManager tries to connect to a ServerLockManager, as specified in its lockServerAddress and lockServerPort properties, within the time allowed by its connectTimeout property. If it cannot connect to a ServerLockManager, it issues a warning message and the item descriptors use simple caching.

2. Immediately after the ServerLockManager connects to the GSACacheClientManager, it hands over the connection to the GSACacheServerManager.
**Optimizing Performance**

In order to obtain maximum benefit from distributed hybrid caching mode, it is important to set the following attributes correctly:

- Set the size of each item cache that participates in distributed hybrid caching to the maximum number of items. This must be done on each repository instance, on all servers.

- Set the GSACacheServerManager’s `defaultItemCacheSize` property to the maximum number of items that use distributed hybrid caching across all client servers. This can be expressed by the following formula:

  \[ \text{item-cache-size} \times \text{num-clients} + \text{item-cache-size} \times \text{num-clients} + \ldots \]

For example, a site might be set up as follows:

- The item descriptor `user` in the ProfileAdapterRepository is set to `distributedHybrid` caching mode.

- The site has 10 servers in the page serving cluster and 5 servers in the agent cluster, and they all require access to `user` items; thus, each server must have an instance of the ProfileAdapterRepository.

- The number of stored `user` items is currently just over 700. Allowing for future growth, each instance of ProfileAdapterRepository has its `user` cache size set to 1000. This ensures that all user caches have the capacity to store all potential `user` items.

If no other item descriptors specify distributed hybrid caching, the GSACacheServerManager should set `defaultItemCacheSize` to 15 thousand. With this setting, it can keep track of all the `user` items that might be cached across all servers.

Setting the item cache size or `defaultItemCacheSize` too low is liable to incur increased network activity as follows:

- **Insufficient item cache capacity:** For each new item that is loaded into the cache, an old item must be unloaded; each action requires a separate call to the GSACacheServerManager to update its routing information.

- **defaultItemCacheSize is less than the total number of cached (distributed hybrid) items on all servers:** For each newly cached item, the GSACacheServerManager must unload routing information for another (older) cached item. As a result, routing information is liable to be missing for some cached items; attempts to access those items requires an invalidation message to be broadcast to all servers, whether or not they cache the item. This behavior is similar to distributed TCP caching.

Overall, the additional network activity can cause a slowdown in site performance that significantly overshadows the benefit of real-time updates. In general, if memory is limited, it is best to limit usage of distributed hybrid caching to item types that store a relatively small number of items, in order to minimize network overhead.
Monitoring Cache Manager Activity

You can use the Component Browser of the ATG Dynamo Server Admin to monitor the current state of GSACacheClientManagers and the GSACacheServerManager. Data for these components is accessible from this page:

http://host:port/dyn/admin/nucleus/atg/dynamo/service/

The GSACacheClientManager and GSACacheServerManager pages provide links that can help you troubleshoot distributed hybrid caching.

GSACacheClientManager page

Provides a link Display ItemDescriptor List that lists all item descriptors managed by this GSACacheClientManager.

GSACacheServerManager page

Provides two links:

- Display Client List lists all of GSACacheClientManagers currently connected to this GSACacheServerManager.
- Display Repository List provides lists all repositories that this server manages. From this page, you can drill down to specific item information:
  - Click on a repository to list all item descriptors that use hybrid caching.
  - Click on an item descriptor to display an input field for entering a repository item ID.
  - Enter a repository item ID and click Get Clients to obtain all GSACacheClientManagers that currently cache this item.

Cache Configuration

The SQL repository maintains separate item caches and query caches for unrelated item descriptors. Thus, each one can be configured independently—for example, each item cache can have its own size and caching mode. Item descriptors within an inheritance tree also maintain their own query caches. Thus, each item descriptor can set its own query cache attributes: query-cache-size and query-expire-timeout.

Item descriptors within an inheritance tree (see Item Descriptor Inheritance) share the same item cache; however, related item descriptors set item cache attributes independently, with one exception: the last-read item descriptor's item-cache-size setting applies to all item descriptors within the inheritance tree. In order to ensure the desired item cache size, be sure to assign the same item-cache-size to all related item descriptors.

Item cache attributes

You can configure item caches with the following <item-descriptor> attributes:
### Attribute Description

#### item-cache-size

The maximum number of items of this type to store in the item cache. When this maximum is exceeded, the oldest items are removed from the cache.

**Note:** Within an inheritance tree, the last-read item descriptor’s `item-cache-size` setting applies to all item descriptors within the inheritance tree. In order to ensure the desired item cache size, be sure to assign the same `item-cache-size` to all related item descriptors.

Default: 1000

#### item-expire-timeout

The maximum time in milliseconds that an entry can remain in the item cache before its content becomes stale. After turning stale, the item cache entry is reloaded from the database the next time it is accessed. See Cache Timeout for more information.

Default 0 (cached data is not refreshed)

#### item-cache-timeout

The time in milliseconds that an item cache entry can remain unused before its content becomes stale. After turning stale, the item cache entry is reloaded from the database the next time it is accessed. See Cache Timeout for more information.

Default: 0 (cached data is not refreshed)

---

### Query cache attributes

You can configure query caches with the following `<item-descriptor>` attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>query-cache-size</td>
<td>The maximum number of queries for items of this type to store in the query cache. When this maximum is exceeded, the oldest queries are removed from the cache.</td>
</tr>
<tr>
<td></td>
<td>Default: 0</td>
</tr>
<tr>
<td>query-expire-timeout</td>
<td>The maximum time in milliseconds that an entry can remain in the query cache before its content becomes stale. After turning stale, the result set is reloaded from the database the next time the entry is accessed. See Cache Timeout for more information.</td>
</tr>
<tr>
<td></td>
<td>Default: 0 (items remain cached indefinitely or until invalidated)</td>
</tr>
</tbody>
</table>
Note: While it might be useful to disable caching during evaluation and development by setting cache sizes to 0, be sure to set caches to appropriate sizes for testing and deployment. For information about analyzing cache usage, see Monitoring Cache Usage in this chapter.

Query Cache Tuning

A query cache size should anticipate the number of queries that are typically executed against that repository. For example, a query cache is probably not effective for the profile repository, as the most common type of query is the login query, which is executed once for each login/password combination. A query whose parameters are subject to frequent changes—for example, a query parameter that is set to system time—is also not a good candidate for caching.

It is generally safe to set the size of the query cache to 1000 or higher. Query caches only contain the query parameters and string IDs of the result set items, so large query cache sizes can usually be handled comfortably without running out of memory.

Item Cache Tuning: ATG Commerce

In an ATG Commerce application, caches for the product catalog repository should be large enough to accommodate the entire catalog—its categories, products and SKUs—or the data that is most frequently accessed. In general, a site can comfortably cache a catalog that contains up to 2 thousand categories, 10 thousand products, and 100 thousand SKUs.

In order to set cache sizes for /atg/commerce/order/OrderRepository, estimate the number of concurrent sessions, and the number of expected orders, items, and shipping groups. The following table provides some guidance on how to set cache sizes for various item descriptors:

<table>
<thead>
<tr>
<th>Item descriptor</th>
<th>Typical site usage</th>
<th>Item cache size</th>
</tr>
</thead>
<tbody>
<tr>
<td>orders</td>
<td>500 concurrent user sessions</td>
<td>500-700</td>
</tr>
<tr>
<td>commerce</td>
<td>3 items per order (average)</td>
<td>1500</td>
</tr>
<tr>
<td>shippingGroup</td>
<td>2 shipping groups per order</td>
<td>1000</td>
</tr>
</tbody>
</table>

Cache Timeout

An item descriptor can limit the lifetime of cached items in two ways:

- Force refreshes of items in the item and query caches
- Refresh unused item cache entries

**Force refreshes of items in the item and query caches**

An item descriptor's item-expire-timeout and query-expire-timeout attributes specify how long items can stay in the item and query caches, respectively, before they are invalidated. For example, if the
item-expire-timeout for a cached item is set to 60000 (milliseconds), its data becomes stale after 60 seconds; and the item must be reloaded from the database when it is next accessed.

The following item-descriptor tag sets attributes item-expire-timeout and query-expire-timeout to 180 seconds:

```xml
  <item-descriptor name="order" cache-mode="simple"
                  item-expire-timeout="180000"
                  query-expire-timeout="180000">
    ...
  </item-descriptor>
```

**Refresh unused item cache entries**

An item cache entry is regarded as stale if it is not accessed within the time span specified in its item-cache-timeout attribute. A stale item must be reloaded from the database the next time it is accessed. If set to 0 (the default), the item can remain indefinitely in the item cache until it is otherwise invalidated.

The following item-descriptor tag sets attribute item-cache-timeout to 180 seconds:

```xml
  <item-descriptor name="order" cache-mode="simple"
                  item-cache-timeout="180000">
    ...
  </item-descriptor>
```

You can use cache timeout attributes together with simple caching to control the behavior of repository caches. Cache timeout attributes are useful for caching user data associated with a particular session. A user session is typically handled by a single ATG server for as long as the session lasts. If the user session expires and the user moves to another ATG server, the cached data expires before the user can log back on to a server that might have previously cached stale data for that user.

**Monitoring Cache Usage**

You can view details on usage of repository caches with the Administrative Interface Component Browser. For example, the Profile Repository’s page in the Component Browser can be found at:

```text
   http://hostname:port/dyn/admin/nucleus/atg/userprofiling/ProfileAdapterRepository/
```

where hostname is the name of the application server’s host machine, and port is the port where the application server listens for HTTP requests. Cache metrics are displayed under the heading Cache Usage Statistics.

You should monitor cache usage during testing and after deployment, in order to determine the cache sizes required to optimize application performance. For example, if the item cache records a high number of failed access tries and no successful tries, it is likely that caching items of this type yields no benefit and
Caching can be disabled; if it shows a mix of successful and unsuccessful access tries, the cache is probably too small; and if it records a high number of successful access tries and no failures, the cache is probably big enough.

The cache usage statistics table contains the following data:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entryCount</td>
<td>Current number of cache entries</td>
</tr>
<tr>
<td>cacheSize</td>
<td>Maximum cache size</td>
</tr>
<tr>
<td>usedRatio</td>
<td>Percent of the cache in use. If the value is close to 100, you should probably increase the item descriptor's cacheSize setting.</td>
</tr>
<tr>
<td>accessCount</td>
<td>Total tries, successful and unsuccessful, to retrieve items or query results since cache startup</td>
</tr>
<tr>
<td>hitCount</td>
<td>Total number of successful access tries since cache startup</td>
</tr>
<tr>
<td>missCount</td>
<td>Total number of failed access tries since cache startup</td>
</tr>
<tr>
<td>hitRatio</td>
<td>The hitCount percentage of accessCount. For example, if accessCount is 100 and hitCount is 75, hitRatio is 75%.</td>
</tr>
<tr>
<td>cacheInvalidations</td>
<td>Number of times the entire cache was invalidated since cache startup</td>
</tr>
<tr>
<td>entryInvalidations</td>
<td>Number of invalidated entries since cache startup</td>
</tr>
</tbody>
</table>

You can also examine the contents of a cache in the Admin UI, by executing the XML tag `<dump-caches>` on a repository. See Preloading Caches later in this chapter.

### Caching by Repository IDs

If a repository item property references a large sub-tree of items, it can be inefficient to retrieve and cache the child items each time the parent item is cached. In this case, you can specify to cache only the repository IDs of the child items, by setting the property's cacheReferencesById to true. For example:

```
<property name="childProducts" ...>
  ...
  <attribute name="cacheReferencesById" value="true"/>
</property>
```
Restoring Item Caches

You can configure a GSARepository component so it automatically saves its item caches when the repository is stopped; when the repository restarts, it reloads caches with the same items. To enable this, set the following property:

```
restoreCacheOnRestart=true
```

When this property is set to true, the repository saves the names of cached items to the XML file specified by the repository's `cacheRestoreFile` property.

Preloading Caches

You can achieve better performance in an application that uses SQL repositories by preloading caches. You can configure the SQL repository to save and restore caches automatically. It is generally good practice to put cache-loading tags in a separate XML file with the same name as the repository definition file, and rely on XML file combination to invoke the queries. For more about XML file combination, see the Nucleus: Organizing JavaBean Components chapter of the ATG Programming Guide.

If you preload caches, the loading strategy of the preloaded items on startup should be set to eager (the default). Alternatively, you might optimize performance by lazy loading repository items. For more information, see Enabling Lazy Loading later in this chapter.

**Note:** Performance improvement by preloading caches slows application startup.

You can specify to load certain items into repository caches on application startup in several ways:

- Load specific repository items
- Load queried items
- Load from a dump log

**Load specific repository items**

A repository definition file can include a `load-items` tag to specify one or more repository items to cache on application startup. The tag specifies the items to cache through a comma-delimited list of repository IDs.

For example, the following `load-items` tag specifies to load four items of the `products` item type that match the listed repository IDs:

```
<load-items item-descriptor="product">
  prod10001, prod100001, prod100002, prod100003
</load-items>
```
The `<load-items>` tag can restrict the cached data to specific properties through the `properties` attribute. The previous example can be modified to cache only the data of properties `displayName` and `description`, as follows:

```xml
<load-items item-descriptor="product" properties="displayName,description">
    prod10001, prod100001, prod100002, prod100003
</load-items>
```

**Load queried items**

A repository definition file can include `<query-items>` tags in order to cache query results on application startup. For example:

```xml
<query-items item-descriptor="users">ALL</query-items>
```

The `<query-items>` tag can also set the `quiet` attribute to `true`, to suppress log messages that the query otherwise generates:

```xml
<query-items item-descriptor="product" quiet="true">ALL</query-items>
```

You can set the `id-only` attribute to `true` or `false`:

- `true`: Preload only repository IDs of result set items.
- `false`: Include primary item properties in the preload operation.

For example:

```xml
<query-items item-descriptor="product" id-only="true">ALL</query-items>
```

For more information, see the Querying Items section in the Developing and Testing an SQL Repository chapter.

**Load from a dump log**

You can use the Admin UI to dump the contents of a repository’s item cache at runtime, by executing the `<dump-caches>` tag in the Admin UI utility Run XML Operation Tags on the Repository. If the tag’s `dump-type` attribute is set to `queries`, the tag logs a `<load-items>` tag that can be used to reload all the items that were in the cache at the time of the dump. For example:

```xml
<dump-caches item-descriptors="product" dump-type="queries"/>
```

This tag might yield the following log:

```xml
*** begin pre-cache XML output

<load-items item-descriptor="product">
    prod100003, prod100002, prod100001, prod10001
</load-items>
```
Enabling Lazy Loading

You can lazy load multi-valued property items and query result sets in order to minimize database access and enhance application performance. When you enable lazy loading on an item type or a repository, the cache initially loads only the stubs of the multi-valued property items or the query result set; these include only the repository IDs.

To accelerate item access, lazy loading is integrated with batch loading: when a lazy-loaded item is requested, it is loaded into the cache with a number of related—often contiguous—items, thereby facilitating access within the batch.

It makes sense to enable lazy loading in two cases:

- Multi-valued properties contain a large number of items.
- A query returns a very large result set.

In both cases, you should enable lazy loading if the potential number of items returned is very large, and you only need access to a relatively small subset. For example, if a multi-valued property or query is likely to return 10,000 items and users are likely to access only the first 100 of these, it makes sense to enable lazy loading for that item type or its repository. Conversely, eager loading provides better performance if you need access to all items or a wide spectrum of them.

Lazy Loading Settings

You can enable lazy loading at several levels. In ascending levels of precedence, these are:

- **Repository lazyLoadItems property**
- **Item Descriptor loadingStrategy attribute**
- **API**

**Repository lazyLoadItems Property**

Set the repository's `lazyLoadItems` property to `true`. As installed, ATG enables lazy loading on a number of its versioned and non-versioned repositories. To verify whether a given repository uses lazy loading, check its `lazyLoadItems` property.

**Item Descriptor loadingStrategy Attribute**

Set the `loadingStrategy` attribute to `lazy`. You can set the `loadingStrategy` property on the item type to collect, or on the multi-valued property that references that item type. The `loadingStrategy` setting of a multi-valued property takes precedence over the referenced item type.
The following example shows how you might define the multi-valued property `categoryProducts`, which references a list of `product` components, and sets its `loadingStrategy` attribute to `lazy`:

```xml
<property category-resource="categoryProducts" name="fixedChildProducts" data-type="list" component-item-type="product" column-name="child_prd_id" queryable="true" display-name-resource="fixedChildProducts">
  <attribute name="loadingStrategy" value="lazy"/>
  <attribute name="propertySortPriority" value="-4"/>
  <attribute name="references" value="true"/>
</property>
```

Note: If you preload an item type to a repository’s cache on startup, you must disable lazy loading for the item-type until the preload operation is complete (see Preloading Caches earlier in this chapter).

**API**

You can set the loading strategy programmatically any time after startup. The following code shows how you might use the `atg.adapter.gsa.LoadingStrategyContext` methods `pushLoadStrategy()` and `popLoadStrategy()`, to override and restore the current thread’s loading strategy, respectively:

```java
try {
    LoadingStrategyContext.pushLoadStrategy("lazy");  // or set to "eager"

    // access a collection property that references other items
    List listItems =
        (List) someItem.getPropertyValue("collectionProperty");

    // or run a query
    RepositoryItem [] queryResults = rqlStatement.executeQuery(view, null);
}
finally {
    LoadingStrategyContext.popLoadStrategy();
}
```

For full information about the `LoadingStrategyContext` class and all its methods, see the online ATG API Reference.

**Integration with Batch Loading**

If lazy-loading is enabled, batch-loading for lazy-loaded items is deferred until the data of an item is actually required. In that event, it and related items are batch-loaded to the repository cache as follows:

- The size of each batch is set by the repository’s `loadItemBatchSize` and `queryBatchSize` properties.
- Batch-loaded items remain valid for the length of time set by the repository’s `lazyBatchLoadTimeout` property—by default, set to 10 minutes. After a batch’s
timeout period elapses, items in that batch are loaded individually, which can adversely affect performance.

Using Preloading Hints in Lazy-Loaded Queries

By default, lazy-loaded query result sets contain only repository IDs. If the required data is limited to a few properties, and those properties belong only to the primary table, you can modify lazy-loaded caching by embedding preloading hints in the RQL statement. The result set that is lazy-loaded into the cache includes those properties together with repository IDs.

For example, the following RQL specifies to lazy-load the login property:

```java
RepositoryView view = repository.getView("user");

// to display only logins, preload the login property
RqlStatement statement =
    RqlStatement.parseRqlStatement("firstName = ?0 PROPERTY HINTS login");

Object params[] = new Object[1];
params[0] = "Maria";
RepositoryItem[] items = statement.executeQuery(view, params);
```

You can also use the interface `atg.repository.Repository` to embed preloading hints in a query programmatically. The following code excerpt is equivalent to the RQL example shown above:

```java
// Somehow, get the repository
Repository rep = getRepository();
RepositoryItemDescriptor desc = rep.getItemDescriptor("user");

// RepositoryView is a ParameterSupportView, so it supports parameters in queries
// This assumes advanced knowledge that the view is an instance of a
// ParameterSupportView
ParameterSupportView view = (ParameterSupportView)desc.getRepositoryView();
QueryBuilder qb = view.getQueryBuilder();

// firstName = 'Maria'
QueryExpression firstNameProp = qb.createPropertyQueryExpression("firstName");
QueryExpression parameterValue = qb.createParameterQueryExpression();
Query firstNameQuery = qb.createComparisonQuery
    (firstNameProp, parameterValue, QueryBuilder.EQUALS);

// arguments
Object[] args = new Object[1];
args[0] = new String("Maria");

// preload "login" property in order to display it without loading full items
String [] precachedProperties = new String[1];
precachedProperties[0] = "login";
QueryOptions options = new QueryOptions(0, -1, null, precachedProperties);

RepositoryItem[] mariaItems = view.executeQuery(firstNameQuery, options, args);

Cache Flushing

The ATG distribution provides class methods that you can use to explicitly flush item and query caches at several levels, as described in the following sections:

- Flushing All Repository Caches
- Flushing Item Caches
- Flushing Query Caches

The following table summarizes these methods:

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>atg.repository.RepositoryImpl</td>
<td>invalidateCaches()</td>
</tr>
<tr>
<td>atg.repository.ItemDescriptorImpl</td>
<td>removeItemFromCache()</td>
</tr>
<tr>
<td></td>
<td>invalidateItemCache()</td>
</tr>
<tr>
<td></td>
<td>invalidateCaches()</td>
</tr>
<tr>
<td>atg.repository.RepositoryViewImpl</td>
<td>invalidateQueryCache()</td>
</tr>
</tbody>
</table>

Flushing All Repository Caches

The class atg.repository.RepositoryImpl provides the method invalidateCaches(), which clears all caches from the target repository:

void invalidateCaches()
void invalidateCaches(boolean pGlobal)

If you supply an argument of true to the boolean version, the method is invoked on all cluster repositories.

Two exceptions apply:

- If you call the non-boolean version on a repository where item descriptors are set to distributed hybrid caching, a cache invalidation event is triggered for each of those item descriptors, which the GSACacheServerManager distributes accordingly.
The boolean version has no affect on remote item caches that use distributed TCP caching.

Flushing Item Caches

The class `atg.repository.ItemDescriptorImpl` provides three methods that can be called on an item descriptor in order to flush its cache. Each method provides an overloaded version with a boolean parameter, where an argument of `true` specifies to invoke the method across the entire cluster; an argument of `false` limits the flush operation to the local repository.

**Note:** If the item descriptor is set to distributed hybrid caching, any action on the local repository is propagated to other ATG instances, whether invoked by the boolean or non-boolean method version.

In order to use these methods, `atg.repository.RepositoryItemDescriptor` must be cast to `atg.repository.ItemDescriptorImpl`.

`invalidateItemCache()`

Invalidates item caches for this item descriptor:

```java
void invalidateItemCache()
void invalidateItemCache(boolean pGlobal)
```

For example:

```java
RepositoryImpl rep = getRepository();
ItemDescriptorImpl d = (ItemDescriptorImpl)rep.getItemDescriptor("user");
d.invalidateItemCache();
```

**Note:** This method’s boolean has no affect on remote item caches that use distributed TCP caching; it only invalidates the local item cache.

`invalidateCaches()`

Invalidates item and query caches for this item descriptor:

```java
void invalidateCaches()
void invalidateCaches(boolean pGlobal)
```

`removeItemFromCache()`

Removes the specified item from the cache:

```java
void removeItemFromCache(String pId)
void removeItemFromCache(String pId, boolean pGlobal)
void removeItemFromCache(String pId, boolean pGlobal, boolean pRemoveTransientProperties)
```

The version with the boolean parameter `pRemoveTransientProperties` forces removal of transient property values from the cache. The other method versions have no effect on these properties.
**flushing query caches**

The class `atg.repository.RepositoryViewImpl` provides the method `invalidateQueryCache()`, which clears the query cache:

```java
public void invalidateQueryCache()
```

**cache invalidation service**

ATG includes a JMS-based system for explicitly invalidating caches in an SQL repository, where ATG servers in a cache invalidation cluster act as message sinks and sources. Cache invalidation messages are initially created by invoking a client method remotely via RMI to a specific server. All servers in the cluster, configured as message sinks or subscribers to the GSA Invalidation topic, accept the message and perform the appropriate cache invalidation as specified by message parameters.

You can specify several levels of cache invalidation:

- Invalidate the cache of a given repository item.
- Invalidate all repository items of an item descriptor.
- Invalidate all repository items.

The Cache Invalidator can be used in two different ways:

- **Invoke the Cache Invalidator manually** from the command line for repository items that you specify.
- **Use the Cache Invalidator with distributed JMS caching** so it is invoked automatically for repository items whose cache mode is set to `distributedJMS`.

**enabling the cache invalidator**

The Cache Invalidator is disabled by default. If your installation includes the DPS module, the SQL-JMS system is preconfigured to work with the Cache Invalidator. In this case, you can enable the Cache Invalidator by setting the property `gsaInvalidatorEnabled` property to `true` in this component:

```
/atg/dynamo/Configuration
```

**Note:** All ATG instances participating in the Cache Invalidator scheme must be configured to access the same SQL-JMS database with the appropriate JDBC datasource configurations. The SQL repository or repositories that are invalidated might or might not be part of the same datasource.

**invoke the cache invalidator manually**

You can invoke the Cache Invalidator Client from a command shell as follows:
The Cache Invalidator action is initiated by performing a RMI call to the
GSAInvalidatorService.invalidate() method. The RMI call is made by executing the GSA
Invalidator Client, which is supplied one or more of the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repository-path</td>
<td>Required, the Nucleus path of the SQL repository</td>
</tr>
<tr>
<td></td>
<td>If this is the only argument, then the cache is invalidated for the entire repository.</td>
</tr>
<tr>
<td>item-descriptor-name</td>
<td>Optional, invalidates all items of this item type.</td>
</tr>
<tr>
<td>repository-item-id</td>
<td>Optional, invalidates a specific item from the repository.</td>
</tr>
</tbody>
</table>

You can enable additional debugging messages by setting the property loggingDebug=true in the following components:

- /atg/dynamo/service/GSAInvalidatorService: producer or message source debugging
- /atg/dynamo/service/GSAInvalidationReceiver: consumer or message sink debugging

**Use the Cache Invalidator with Distributed JMS Caching**

For items where cache-mode="distributedJMS", the GSAInvalidatorService is used to send cache invalidation events via JMS. To do this, the service sends a JMS event of class
atg.adapter.gsa.invalidator.MultiTypeInvalidationMessage, which invalidates a set of items or item types for a given repository. When a transaction is committed, this event is used to invalidate the caches of all items modified in the transaction that use the distributedJMS cache mode.

Because transactions can be arbitrarily large, it is necessary to specify the maximum size of a MultiTypeInvalidationMessage event. The GSAInvalidatorService component includes a property called maxItemsPerEvent, which specifies the maximum number of repository items that one MultiTypeInvalidationMessage can invalidate. The default value for this property is 200. If the number of items to invalidate after a transaction exceeds this threshold, the message invalidates the caches of the updated item types, rather than invalidating caches of individual items. This mechanism keeps the message from growing too large, because it needs only contain information about the item types to invalidate, rather than a list of individual items.

There is no command-line interface for sending MultiTypeInvalidationMessage events. These events are used only for distributed JMS caching.
11 Developing and Testing an SQL Repository

The XML document type definition for the SQL repository includes operation tags whose primary purpose is to help you develop, test, and debug your SQL repository template. You can use these tags to modify your repository’s database to perform the following tasks:

- Adding Items
- Updating Items
- Removing Items
- Querying Items
- Importing and Exporting Items and DDLs

These tags are used by the startSQLRepository script, which is described in this chapter.

To use these developmental tags:

1. Go to the repository’s page in the Administration Interface. For example, for the SQL Profile Repository, go to:

   `hostname:8830/nucleus/atg/userprofiling/ProfileAdapterRepository`

2. In the Run XML Operation Tags on the Repository text area, enter the developmental tags and click Enter.

You can also run the startSQLRepository script from a command line. Create an XML repository definition file and pass it to the startSQLRepository script with appropriate arguments. See the startSQLRepository section in this chapter for more information.

Note: If you add or remove an item descriptor in your repository definition file, you must close and reassemble, redeploy, and restart your application, which restarts the ACC and DAF. Otherwise, errors may result. For example, if you remove an item descriptor, the item descriptor still appears as an option in the ACC query editor (List items of type...) and might cause errors if selected. For instructions on assembling applications, see the ATG Programming Guide.
Adding Items

You can use an XML template to add repository items. Use an `<add-item>` tag for each repository item you want to add. Each `<add-item>` tag must include an `item-descriptor` attribute to specify the name of the item descriptor to which this repository item should belong. You can nest `<set-property>` tags within the `<add-item>` tag to set property values of the new repository item. Any properties you do not set have the default property value for that item descriptor.

For example, the following tags add to the database an instance of `users` with id = 1. It sets the `username` property to Marty.

```xml
<add-item item-descriptor="users" id="1">
    <set-property name="username" value="Marty"/>
</add-item>
```

Note that `<add-item>` tags are processed one at a time. They cannot make forward references to other items and no attempt is made to satisfy database integrity constraints (beyond that automatically done with the cascade operator). Use the `<import-items>` tag if you want to load in items with forward references.

Note also that if you specify the ID of an existing repository item, you update that item, overwriting the values of the existing item with the values you specify in the `<add-item>` tag. Any `add` or `remove` attributes in a `<set-property>` tag within an `<add-item>` tag are ignored.

Adding Items with Composite IDs

If your repository uses composite repository item IDs, you can specify the ID with its encoded form, or in brackets as comma-delimited ID elements. For example, if an ID is composed of the string elements Massachusetts, USA, and Earth and the separator character is the default, (colon), you can specify the ID in one of these forms:

```xml
<add-item item-descriptor="states" id="Massachusetts:USA:Earth">
    <set-property name="capital" value="Boston"/>
</add-item>
```

```xml
<add-item item-descriptor="states" id=[Massachusetts,USA,Earth]>
    <set-property name="capital" value="Boston"/>
</add-item>
```

Adding Items without Specifying IDs

When you add a repository item with the `<add-item>` tag, you can use the `tag` attribute in place of the `id` attribute. If you use a `tag` attribute, the SQL repository chooses a unique repository ID for the item using the `IdGenerator` and associates that tag with that ID. You can then refer to that particular tag name within that XML file with a `tag` attribute. Alternatively, refer to the tag with this special syntax:
$tag: <name>$

The $tag: <name>$ syntax can be used only in:

- the value attribute or body of a <set-value> tag
- the query attribute or the body of a <query-items> tag

The template parser substitutes the ID of the item you created with that tag.

For example, you might want to add an item, one of whose properties is another repository item—for example, a book item, where the book item has an author property which is itself a repository item. If you do not want to supply the repository ID for the author repository item, you can use a tag attribute placeholder like this:

```xml
<add-item item-descriptor="author" tag="AUTHORID_TAG">
  <set-property name="authorName" value="Arthur Ransome"/>
</add-item>
<add-item item-descriptor="book">
  <set-property name="title" value="Swallows & Amazons"/>
  <set-property name="author" value="$tag:AUTHORID_TAG$"/>
</add-item>
```

Adding Items to Multi-Item Properties

If you add items that are themselves properties of other repository items, make sure the item is added before the item that refers to it. This is necessary because a new repository item cannot make forward references to another repository item that is not yet defined.

For example, suppose you have a user item type with a dependents property that refers to a separate dependent item type. Add the dependent items before you add a user item that refers to those dependent items, as in this example:

```xml
<add-item item-descriptor="dependent" id="1234">
  <set-property name="firstName" value="JoeBob"/>
</add-item>
<add-item item-descriptor="dependent" id="1235">
  <set-property name="firstName" value="Mikey"/>
</add-item>
<add-item item-descriptor="user" id="1">
  <set-property name="login" value="toml"/>
  <set-property name="firstName" value="Tom"/>
  <set-property name="dependents" value="1234,1235"/>
</add-item>
```
Updating Items

You can update repository items with the `<update-item>` tag. The `<update-item>` tag encloses one or more `<set-property>` tags that specify the properties and values being set. Each `<update-item>` tag must include an `item-descriptor` attribute to specify the name of the item descriptor of the repository item being removed. You can also use the `skip-update` attribute to set properties in the item, but avoid the update item call until the transaction is committed.

For example, the following element changes the value of the `dependents` property of the user with `id` of 1:

```xml
<update-item item-descriptor="user" id="1" skip-update="true">
  <set-property name="dependents" value="1414,1732"/>
</update-item>
```

You can use the `add` or `remove` attributes to add or remove values from multi-item properties without overwriting the whole property value. For example, to add another value to the `dependents` property:

```xml
<update-item item-descriptor="user" id="1" skip-update="true">
  <set-property name="dependents" value="1799" add="true"/>
</update-item>
```

Removing Items

You can remove items from the repository with the `<remove-item>` tag. Each `<remove-item>` tag must include an `item-descriptor` attribute to specify the name of the item descriptor of the repository item being removed.

For example, the following tag removes a repository item that uses the item descriptor `users` and whose repository ID is 1:

```xml
<remove-item item-descriptor="users" id="1"/>
```

Removing References to Items

When you remove an item, you generally also need to remove references to the item. The `atg.repository.RepositoryUtils` class includes two methods that are useful in this context.

The `removeReferencesToItem` method removes any references to a given item from other items in its repository. This method can only remove references in queryable properties. You can invoke the `removeReferencesToItem` method by setting `remove-references-to="true"` in a `<remove-item>` tag.
The changes to the data caused by the `removeReferencesToItem` method depend on the reference type. For example, you might delete an item of type X, where type Y references X. Item descriptor Y might reference item descriptor X in three ways:

<table>
<thead>
<tr>
<th>If Y references X in this way…</th>
<th>Delete the reference to X as follows…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y has a non-required property whose item-type is X</td>
<td>Set the reference property to null and update item of type Y, essentially nulling the foreign key</td>
</tr>
<tr>
<td>Y has a required property whose item-type is X</td>
<td>Delete the item of type Y, because the foreign key cannot be set to null.</td>
</tr>
<tr>
<td>Y has a multi-valued property whose component-item-type is X</td>
<td>Remove the element in the multi-valued property in Y that refers to the item of type X. This deletes the one-to-many or many-to-many row that represents the Y to X reference. The item of type Y is not deleted.</td>
</tr>
</tbody>
</table>

Data in an auxiliary table is always deleted by a `<remove-item>` tag regardless of the `remove-references-to` attribute because it is not considered a reference.

The `anyReferencesToItem` method queries whether any cross-references to a repository item exist within the repository that contains that item. It uses the same logic as the `removeReferencesToItem` method to determine whether references exist. The `anyReferencesToItem` method can only detect references through queryable properties.

Calling these methods generates multiple repository queries per call, one for each property descriptor that might refer to the item. For example, if the item's type is `contact-info`, one query is performed for each property descriptor whose type is `contact-info`, or any multi-valued type that might contain a list of items of type `contact-info`. The queries each fetch at most one item from the repository, so the effect on the repository's cache should be minimal.

### Querying Items

You can perform queries against the repository through the `<query-items>` tag. The query itself may be specified as a query attribute of the `<query-items>` tag or as PCDATA between the opening and closing tags. The query uses the Repository Query Language (RQL) described in the Repository Query Language section of the Repository Queries chapter.

For example, the following tag queries the database for any repository items whose item descriptor is named `users` and whose `username` property is `Marty`:

```
<query-items item-descriptor="users">username='Marty'</query-items>
```

Queries can be used in this way to preload repository caches. See Preloading Caches in the SQL Repository Caching chapter.
Importing and Exporting Items and DDLs

Some operations tags let you import items from another repository or export items. You can also print out the DDLs used in setting up the tables corresponding to the repository template. See the descriptions of the following tags in the SQL Repository Definition Tag Reference:

- remove-all-items
- export-items
- import-items
- print-ddl

You can also use the startSQLRepository script to export, import, and print repository items, as described in the next section.

startSQLRepository

You can use the utility program startSQLRepository in order to read a repository definition from an XML file or DOM. startSQLRepository can perform these tasks:

- Verify the XML is correctly formed and complies with the DTD.
- Parse and process optional operation tags like <add-item>, <remove-item>, and <query-items>. These tags provide a means for adding, removing, updating items in your SQL repository.
- Generate SQL statements that are required to create the appropriate database table structure when you use the -outputSQL flag.
- Return results of <query-items> and <print-item> requests in the form of <add-item> tags. This lets you easily copy and paste the results into another XML template, so you can add the items to another repository.
- Import and export items and item descriptors.

Note: When running startSQLRepository on a third-party application server, you must configure the server to use an ATG data source and transaction manager, not your native application server’s data source and transaction manager.

Requirements

The following requirements apply if the template contains <table> tags:

- The database accessed by your repository is running.
- The database contains the appropriate tables.
- You have appropriate database access to perform import and create database operations.
The following requirements apply to ensure that the import operation reserves all the IDs it encounters for the repository items that it creates in the target database.

- Repository IDs in the source repository do not collide with repository IDs in the target repository.
- The source and target databases contain IdSpaces, where IdSpaces in both share the same name.
- The name of the IdSpaces used by each item descriptor is the same in the source and target repositories for export and import operations, respectively.

For more information about IdSpaces, see the Core Dynamo Services chapter of the ATG Programming Guide.

**Syntax**

`startSQLRepository` uses the following syntax:

```
startSQLRepository arguments xml-file[ ...]
```

You can supply multiple XML files to `startSQLRepository`, and it processes them in the order specified. For example, you can pass your full repository definition file together with a test file that uses the test operation tags to manipulate repository items, as shown in SQL Repository Test Example.

The following example loads an XML template whose configuration path name is `/atg/test.xml` in a repository with a Nucleus address of `/atg/userprofiling/ProfileAdapterRepository`:

```
startSQLRepository -m DPS
    -repository /atg/userprofiling/ProfileAdapterRepository /atg/test.xml
```

The XML template file name is located in the application’s configuration path. For example, you can reference a file in the `localconfig` directory as `/file-name`. You can also use a file with the same name as your repository’s existing definition file and omit the file name argument from the `startSQLRepository` command. The `startSQLRepository` script uses XML file combination to combine all files with the same name into a single repository definition. See the Nucleus: Organizing JavaBean Components chapter of the ATG Programming Guide.

For example, you can use `startSQLRepository` to print all profiles in the Profile repository by including the following file in:

```
<ATG10dir>/home/localconfig/atg/userprofiling/userProfile.xml
```

```
<gsa-template>
    <print-item item-descriptor="user"/>
</gsa-template>
```

You can use the `startSQLRepository` script together with the test operation tags described earlier in this chapter to quickly test a query, or add, update, remove, or print an item.
### General Arguments

The `startSQLRepository` scripts take the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| `-m startup-module`     | Lists a module to load which contains the target repositories. Supply multiple `-m` options in order to start more than one module. For example:  
  startSQLRepository -m moduleA -m moduleB  
  This argument must precede all others, including `-import`.                              |
| `-s server-name`        | The ATG instance on which to run this script. Use this argument when you have multiple servers running on your machine.  
  This argument must precede all others except `-m`.                                               |
| `-database vendor`      | Customizes the DDL for the SQL variant used by the specified vendor’s database software, where `vendor` can be one of the following values:  
  db2  
  microsoft  
  oracle  
  solid  |
| `-debug`                | Outputs additional logging information. This option is equivalent to setting the `loggingDebug` property to `true` for the repository. |
| `-encoding encoding`    | If the content you are exporting contains non-ASCII characters, use this option to specify an encoding such as 8859_1 or SJIS in which to save your content. |
| `-export "item-type[,...]" file` | Exports items of one or more item descriptors to an XML repository definition file, where `file` is relative to the `<ATG10dir>/home` directory. |
| `-export all file`      | Exports all items of all item descriptors in this repository to a file, where `file` is relative to the `<ATG10dir>/home` directory. |
| `-exportRepositories repository-path[,...] file` | To export data from more than one repository into the same file, use this option. This might be preferable to the `-export` option if repositories are linked, as it prevents duplicating linked item descriptors in multiple files. |
| `-exportRepositories all file` | Exports all repositories to one file, where `file` is relative to the `<ATG10dir>/home` directory. |
### Argument Purpose

<table>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-import input-file</code></td>
<td>The XML file or DOM that contains the repository definition to import into the target repository, where input-file is a file created from running <code>startSQLRepository</code> with <code>-export</code> or <code>-export all</code>. The path of <code>input-file</code> can be absolute or relative to the current directory.</td>
</tr>
<tr>
<td><code>-noTransaction</code></td>
<td>If you use this argument, this operation is not wrapped in a transaction. Using a transaction for large operations can run into database limitations on transaction sizes and numbers of permitted row-level locks.</td>
</tr>
<tr>
<td><code>-output file</code></td>
<td>Sends all output from <code>&lt;print-item&gt;</code> and <code>&lt;query-item&gt;</code> tags to the specified file, instead of standard output.</td>
</tr>
<tr>
<td><code>-output SQL</code></td>
<td>Outputs a DDL (SQL) file for the XML templates in the repository to standard output.</td>
</tr>
<tr>
<td><code>-outputSQLFile file</code></td>
<td>Outputs a DDL (SQL) file for the XML templates in the repository to the specified file.</td>
</tr>
<tr>
<td><code>-repository path</code></td>
<td>The Nucleus path of the repository. For example:</td>
</tr>
<tr>
<td></td>
<td><code>-repository</code> /atg/dynamo/service/jdbc/SQLRepository</td>
</tr>
<tr>
<td></td>
<td>If you run <code>startSQLRepository</code> with the DPS module or a module that requires the DPS module, you can omit this argument, and the script uses the first repository registered in the component/atg/registry/ContentRepositories.</td>
</tr>
<tr>
<td></td>
<td>If you use <code>startSQLRepository</code> to import assets into a versioned repository, this argument is required if the input file specifies the target repository with add-item and update-item tags.</td>
</tr>
<tr>
<td><code>-skipReferences</code></td>
<td>By default, when you use one of the export arguments, all referenced item descriptors are automatically added to the list of item descriptors to be exported. If you use the <code>-skipReferences</code> argument, referenced item descriptors are added only if you affirmatively include them.</td>
</tr>
<tr>
<td><code>-verboseSQL</code></td>
<td>Outputs additional logging information, equivalent to setting <code>loggingSQLInfo</code> and <code>loggingSQLDebug</code> properties to <code>true</code> for the repository's JTDataSource. All SQL emitted is logged.</td>
</tr>
</tbody>
</table>
Exporting Repository Data

You can use `startSQLRepository` to export data from a database to an XML file, which you can later import to another database. For example, you can export Pioneer Cycling data from a SOLID database to a neutral file form and import it into another database, such as Oracle or Microsoft SQLServer, or from one SOLID database to another.

You can export all repositories, or individual repositories from the specified modules. In both cases, first make sure the source database is running.

**Exporting all repositories**

To export the data from all SQL repositories registered in the `/atg/registry/ContentRepositories` component, use the following syntax:

```
bin/startSQLRepository -m module -exportRepositories all output-file
```

The location of the resulting file is relative to the `<ATG10dir>/home` directory.

**Exporting individual repositories**

To export the data from individual SQL repositories, use the following syntax:

```
bin/startSQLRepository -m module -export all output-file -repository repository-path
```

For example, the following command exports the Product Catalog from a Pioneer Cycling store database to `products.xml`:

```
bin/startSQLRepository -m PioneerCycling -export all products.xml -repository /atg/commerce/catalog/ProductCatalog
```

*Note:* when binary data in repository items is exported, it is represented in base64 encoding.

Importing Repository Data

After you use `startSQLRepository` to export repository data to an XML file, you can use the XML file to transfer this data to another database:

1. Add a JDBC driver for your database and configure the JDBC connection pool. For more information, see the ATG Installation and Configuration Guide.

2. Run `startSQLRepository` to import the contents of the XML file to the destination database with the following syntax:

```
startSQLRepository -m module -import input-file -repository repository-path
```

3. For example, given the earlier example, you can import the content from the Pioneer Cycling `products.xml` file as follows:

```
startSQLRepository -m PioneerCycling -import products.xml -repository /atg/commerce/catalog/ProductCatalog
```
Importing to a Versioned Repository

You can use `startSQLRepository` to import repository data into versioned repositories. Before you run this script, format the repository data in an XML file that adheres to the SQL repository definition file syntax.

**Note:** When running `startSQLRepository` on a third-party application server, configure the server to use an ATG data source and transaction manager, not your native application server's data source and transaction manager.

Use the following syntax in order to import repository assets to a versioned repository:

```
startSQLRepository [-m startup-module]... [-s server-name]
    -import input-file -repository path
    {project-spec | workspace-spec} -comment
    [arguments]
```

**Importing to a project or workspace**

The import operation must be directed to a project or to a workspace, as follows:

- `project name [-workflow name] -user username -comment text`
- `workspace name -comment text`

**Note:** After deployment targets are initialized, use `-project` with `startSQLRepository` instead of `-workspace`. When `-project` is used, assets are imported into a new project with the default or specified workflow. Users can then access this project and perform the tasks associated with its workflow.

**Versioning arguments**

The following arguments are specific to the versioning system.
### Argument Description

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-project name</code></td>
<td>The name of the project to create and use for the import. After running <code>startSQLRepository</code> with this argument, the imported assets must be checked in manually through the ATG Business Control Center. This option is available only if the Publishing module is running. You must specify this option or <code>-workspace</code>.</td>
</tr>
<tr>
<td><code>-workspace name</code></td>
<td>Specifies the workspace to use during the import operation, where <code>name</code> is a user-defined string with no embedded spaces and is unique among all workspace names. Use <code>-workspace</code> only during the initial import to the target repository, before you initialize any target sites. The workspace is the area in the VersionManager where the import takes place. If the specified workspace does not exist, the system creates it. Importing into a workspace requires you to supply a checkin comment through <code>-comment</code>. You must specify this option or <code>-project</code>.</td>
</tr>
<tr>
<td><code>-user username</code></td>
<td>The user who performs the import. If a secured repository is accessed for the import, this user’s authorization is checked. This argument is required when the <code>-project</code> argument is supplied, so the user can be identified as the project creator.</td>
</tr>
<tr>
<td><code>-comment text</code></td>
<td>Comment to use for each item when checking in imported data. This comment is stored in each item’s version history. It should not contain spaces.</td>
</tr>
</tbody>
</table>
SQL Repository Test Example

The following is a simple example of how you can create a test repository definition file that defines item descriptors and also uses `<add-item>`, `<remove-item>`, and `<query-items>` tags to manipulate repository items.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- This is a simple xml template demonstrating add-item, remove-item, and query-items tags. --
<!DOCTYPE gsa-template SYSTEM "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Test 1</name>
    <author>Marty</author>
    <version>1</version>
  </header>
  <!-- This defines the item-descriptor -->
  <item-descriptor name="users" default="true">
    <table name="users" id-column-names="id" type="primary">
      <property name="prop1"/>
    </table>
  </item-descriptor>
  <!-- This removes from the database any of 'users' with id = 1 -->
  <remove-item item-descriptor="users" id="1"/>
  <!-- This adds to the database an instance of 'users' with id = 1 -->
  <add-item item-descriptor="users" id="1">
    <set-property name="prop1" value="Marty"/>
  </add-item>
  <!-- This queries the database for any of 'users' with prop1 = "Marty" -->
  <query-items item-descriptor="users">
    prop1="Marty"
  </query-items>
</gsa-template>
```
Using Operation Tags in the Repository Administration Interface

You can also use the operation tags described in this section in the Component Browser of the HTML Administration Interface. Open the HTML Component Browser page for a Repository component. For example, there is an SQL Repository component with a Component Browser URL of:

http://hostname:port/nucleus/atg/dynamo/service/jdbc/SQLRepository

In this URL, hostname represents the name of the machine that runs your application server and port represents the port number that your application server uses to listen for HTTP requests. To find your default port, see the ATG Installation and Configuration Guide.

In the text field, you can enter any XML operations tags against the current repository as if they were commands. Click Enter, and the page displays the output obtained by running the startSQLRepository script against the repository.

Debug Levels

The SQL Repository component has a debugLevel property you can use to adjust the debug log entries. This property is an integer from 0 to 15, with 15 resulting in the greatest frequency of debug log entries.

The default level is 5; this typically is the level desired when contacting ATG support to diagnose problems. Level 5 is the lowest level at which SQL debugging statements are issued.

If loggingDebug is set to true on the SQL Repository component, the debug level must be equal to or greater than 6 in order to set loggingDebug of the transaction manager also to true, as Transaction Manager debugging is often needed in conjunction with SQL debugging. Even at level 0, some debug statements are issued.

You can also get debugging messages for an individual item descriptor or property. Turn on logging debug in the Dynamo Administration Interface page for the Repository component.

You can also turn on debug messages by including a loggingDebug attribute tag in the repository definition for that item descriptor or property. For example:

```
<item-descriptor name="user" ...>
    <attribute name="loggingDebug" value="true" /></attribute>
    ...<property ... /></property>
</item-descriptor>
```
Modifying a Repository Definition

In the course of developing your site or after your site has gone live, you may want to modify your repository schema, adding or removing repository item properties or item descriptors, or altering your database schema. Modifying the repository schema is much like setting it up to begin with:

1. Modify your database, running the appropriate DDLs to make any necessary changes.
2. Edit your repository definition file.
3. Restart your ATG application.
12 SQL Repository Reference

This chapter includes reference information about the SQL repository:

- SQL Repository Definition Tag Reference
- DTD for SQL Repository Definition Files
- Sample SQL Repository Definition Files
- Configuring the SQL Repository Component

SQL Repository Definition Tag Reference

This section describes `gsa-template` elements as they are defined in `gsa_1.0.dtd`. The complete DTD for SQL repository definition files can be found later in this chapter.

A number of tags can be grouped into two categories, and are discussed in detail elsewhere in this guide:

- Named query tags
- Development operation tags

Named Query Tags

The following tags are used to define named queries:

```xml
<named-query>
<rql-query>
<rql>
<sql-query>
<sql>
<input-parameter-types>
<returns>
<dependencies>
```

For more information, see Named Queries in the SQL Repository Queries chapter.

Development Operation Tags

The following tags are used primarily during development, testing, and debugging a repository; they are not typically used in production environments:
For more information, see the Developing and Testing an SQL Repository chapter.

<!DOCTYPE>

All SQL repository templates start with a DOCTYPE declaration that references this document type definition (DTD) file:

gsa_1.0.dtd

This DTD is installed within the <ATG10dir>/DAS/lib/classes.jar archive, but can be referenced with this URL:

http://www.atg.com/dtds/gsa/gsa_1.0.dtd

For example:

<!DOCTYPE gsa-template
     PUBLIC "+//Art Technology Group, Inc.//DTD General SQL Adapter//EN"
     "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">

If your SQL repository definition is comprised of multiple files through XML file combination, include the DOCTYPE declaration only in the file that is first in the application’s configuration path. For more information about XML file combination, see the ATG Programming Guide.

<gsa-template>

<!ELEMENT gsa-template (header?,
   (item-descriptor | add-item | update-item | print-item | remove-item |
   transaction | development-line | query-items | remove-all-items |
   export-items | import-items | print-ddl | dump-caches | load-items)*)>

The <gsa-template> tag is the top-level tag in a repository definition file.
<header>

<!ELEMENT header (name?, author*, version?, description?)>

Parent: <gsa-template>

The <header> tag provides information that can help you manage the creation and modification of repository definition files.

For example:

```xml
<header>
  <name>Catalog Template</name>
  <author>Herman Melville</author>
  <author>Emily Dickinson</author>
  <version>$Id: catalog.xml,v 1.10 2000/12/24 03:34:26 hm Exp $</version>
  <description>Template for the store catalog</description>
</header>
```

</header>

<item-descriptor>

<!ELEMENT item-descriptor ((property | table | attribute | named-query)*, rql-filter?, (property | table | attribute | named-query)*)>

Parent: <gsa-template>

The SQL repository template contains one <item-descriptor> tag for each set of repository items that share the same attributes.

The following sections describe <item-descriptor> attributes:

- General Attributes
- Content Item Attributes
### General Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>The name of this item descriptor, unique within the repository (required). This property is case-insensitive—for example, you cannot set the name property for two item descriptors in the same repository to gender and GENDER. To reference an item descriptor by multiple names, set the itemDescriptorAliases property of the Repository component.</td>
</tr>
<tr>
<td><strong>cache-mode</strong></td>
<td>The caching mode for this item descriptor, one of the following: disabled, simple (default), locked, distributed, distributedJMS, distributedHybrid. Caching can also be disabled for individual properties by setting their cache-mode attribute. See the SQL Repository Caching chapter.</td>
</tr>
<tr>
<td><strong>copy-from</strong></td>
<td>The name of the item descriptor whose properties are inherited by this item descriptor. See Item Descriptor Inheritance.</td>
</tr>
<tr>
<td><strong>default</strong></td>
<td>Boolean, specifies whether this is the repository’s default item descriptor. The default item descriptor is used for new repository items if no item descriptor is explicitly specified. If no item descriptor is designated as the default, the first item descriptor in the repository definition file is the default. Default: false</td>
</tr>
<tr>
<td><strong>description</strong></td>
<td>Optionally describes this item descriptor. Default: value of name</td>
</tr>
<tr>
<td><strong>description-resource</strong></td>
<td>If a resource bundle is specified for this property with the tag &lt;attribute name=resourceBundle&gt;, this attribute specifies the resource bundle key to the item descriptor’s description. See Localizing SQL Repository Definitions.</td>
</tr>
<tr>
<td><strong>display-name</strong></td>
<td>The name of the item descriptor as displayed in the ATG Control Center interface. If no display-name is specified, the name attribute is used.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>display-name-resource</td>
<td>If a resource bundle is specified for this property with the tag <code>&lt;attribute name=resourceBundle&gt;</code>, this attribute specifies the resource bundle key to the item descriptor's display name. See <a href="#">Localizing SQL Repository Definitions</a>.</td>
</tr>
<tr>
<td>display-property</td>
<td>Specifies a property of this item descriptor that is used to represent items of this type in a user interface. For example, a profile item descriptor might set <code>display-property</code> to <code>login</code>. Then, each repository item is represented using the value of the item's <code>login</code> property.</td>
</tr>
<tr>
<td>expert</td>
<td>Boolean, where <code>true</code> specifies to display this item descriptor only to expert users. Default: false</td>
</tr>
<tr>
<td>hidden</td>
<td>Boolean, where <code>true</code> suppresses display of this item types in the ATG Control Center. Default: false</td>
</tr>
<tr>
<td>id-separator</td>
<td>A character used to separate elements of a multi-column repository ID when the ID is string encoded. Default: colon (<code>:</code>)</td>
</tr>
<tr>
<td>id-space-names</td>
<td>The name of the ID space to use for this item descriptor. The default settings are as follows:</td>
</tr>
<tr>
<td></td>
<td>- Item descriptor with a single-column repository ID: Item descriptor name.</td>
</tr>
<tr>
<td></td>
<td>- Item descriptor with a multi-column repository ID: The name of the primary table and the names of the ID column in that table.</td>
</tr>
<tr>
<td></td>
<td>For more information about ID space names and how they affect the IDs of newly generated items, see <a href="#">IdSpaces and the id Property</a> in this manual; and the <a href="#">ATG Programming Guide</a>, the Core Dynamo Services chapter.</td>
</tr>
<tr>
<td>item-cache-size</td>
<td>The maximum number of items of this item descriptor that the item cache can store. When the number of items requested exceeds this number, the least recently accessed item is removed from the cache. See the <a href="#">SQL Repository Caching</a> chapter. Default: 1000</td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item-cache-timeout</td>
<td>The time in milliseconds that an item cache entry can remain unused before its content becomes stale. After turning stale, the item cache entry is reloaded from the database the next time it is accessed. See Cache Timeout for more information. Default: 0 (items remain in the cache indefinitely until otherwise invalidated)</td>
</tr>
<tr>
<td>item-expire-timeout</td>
<td>The maximum time in milliseconds that an entry can remain in the item cache before it is refreshed. See Cache Timeout for more information. Default: 0 (items remain in the cache indefinitely until otherwise invalidated)</td>
</tr>
<tr>
<td>query-cache-size</td>
<td>The maximum number of queries of this item descriptor to store in the query cache. When the number of queries issued against this item descriptor exceeds this number, the least recently used query is removed from the cache. See the SQL Repository Caching chapter. Default: 0 (disables the query cache)</td>
</tr>
<tr>
<td>query-expire-timeout</td>
<td>The maximum time in milliseconds that an entry can remain in the query cache before it is refreshed. See Cache Timeout for more information. Default: 0 (items remain in the cache indefinitely until otherwise invalidated)</td>
</tr>
<tr>
<td>sub-type-property</td>
<td>The name of a property in this item descriptor that specifies the names of its child item descriptors. See Item Descriptor Inheritance.</td>
</tr>
<tr>
<td>sub-type-value</td>
<td>Set to a value that is defined in the parent item descriptor's sub-type-property, which enables inheritance from that parent. See Item Descriptor Inheritance.</td>
</tr>
<tr>
<td>super-type</td>
<td>The name of this item descriptor’s parent. See Item Descriptor Inheritance.</td>
</tr>
<tr>
<td>text-search-properties</td>
<td>A comma-separated list of properties to search if a text search query does not explicitly specify any properties. See Text Search Queries in the SQL Repository Queries chapter.</td>
</tr>
<tr>
<td>versionable</td>
<td>Used only in versioned repositories, specifies whether items of this type should be versioned. Use this attribute to override the setting in the repository property versionItemsByDefault. For more information about this attribute, and versioned repositories in general, see the ATG Content Administration Programming Guide.</td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version-property</td>
<td>A integer property whose value is used as a version control mechanism for items of this type. The value of version-property is incremented each time the item is updated.</td>
</tr>
<tr>
<td>XML:id</td>
<td>Typically used for XML file combination, where elements with the same ID are regarded as the same element.</td>
</tr>
</tbody>
</table>

### Content Item Attributes

The following `<item-descriptor>` attributes are used in content repositories. A content repository includes one item descriptor that manages the folder hierarchy, and one or more item descriptors that define content items. A content item has a property that specifies the item’s folder parent, and a property that is used to store or reference the content data itself. The content data property is usually a `java.io.File`, `String`, or `byte[]` data type. Items in the content item descriptor implement the `ContentRepositoryItem` interface. Items in the folder item descriptor implement the `FolderItem` interface, as well as the `MutableRepositoryItem` interface. For more detail, see the SQL Content Repositories chapter.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>content</td>
<td>Boolean, specifies whether items of this type are content items. If set to <code>true</code>, settings are also required for the following attributes:</td>
</tr>
<tr>
<td></td>
<td>- folder-id-property</td>
</tr>
<tr>
<td></td>
<td>- content-property</td>
</tr>
<tr>
<td></td>
<td>- One or more of: content-name-property, content-path-property, and use-id-for-path</td>
</tr>
<tr>
<td></td>
<td>Default: <code>false</code></td>
</tr>
<tr>
<td>content-checksum-property</td>
<td>Specifies a numeric property in this item descriptor that holds the checksum for a content item descriptor.</td>
</tr>
<tr>
<td></td>
<td>For example, the PublishingFileRepository automatically updates this property when an item's content property changes.</td>
</tr>
<tr>
<td>content-length-property</td>
<td>A property in this item descriptor that contains the number of bytes in the content. This property is used by the method <code>ContentRepositoryItem.getContentLength()</code>.</td>
</tr>
<tr>
<td>content-name-property</td>
<td>A property in this item descriptor that defines the name of this content item or folder item in the folder hierarchy. Unlike content-path-property, the value of this attribute should not include any path separator characters or the names of any parent folders.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>content-path-property</td>
<td>A property in this item descriptor that defines the absolute path name of this item in the folder hierarchy. The setting in <code>content-path-property</code> should include a leading path separator character.</td>
</tr>
<tr>
<td>content-property</td>
<td>Required if <code>content</code> is set to true, the property that contains the content of the content items. The data type of the specified property must be one of the following:</td>
</tr>
<tr>
<td></td>
<td>- File</td>
</tr>
<tr>
<td></td>
<td>- byte[]</td>
</tr>
<tr>
<td></td>
<td>- String</td>
</tr>
<tr>
<td></td>
<td>Content items of type <code>String</code> or <code>byte[]</code> store their data in the database; content items of type <code>File</code> store their data in the file system with the <code>FilePropertyDescriptor</code>.</td>
</tr>
<tr>
<td>folder</td>
<td>Boolean, specifies whether items of this type are folder items. Only one item descriptor in a repository can set this property to true.</td>
</tr>
<tr>
<td></td>
<td>If set to true, settings are also required for the following attributes:</td>
</tr>
<tr>
<td></td>
<td>- folder-id-property</td>
</tr>
<tr>
<td></td>
<td>- content-property</td>
</tr>
<tr>
<td></td>
<td>- One or more of: <code>content-name-property</code>, <code>content-path-property</code>, and <code>use-id-for-path</code></td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>folder-id-property</td>
<td>A property in this item descriptor that specifies the ID of the folder containing this folder or content item. This property must be set for all item descriptors of content and folder items.</td>
</tr>
<tr>
<td>last-modified-property</td>
<td>A property in this item descriptor that contains the time when item content was last modified. The property’s data type must be <code>date</code> or <code>timestamp</code>. This property is used by the method <code>ContentRepositoryItem.getContentViewLastModified()</code>.</td>
</tr>
<tr>
<td>use-id-for-path</td>
<td>Boolean, specifies whether the repository ID for items of this type is the item’s relative path name in the folder hierarchy. Use this attribute if the column used to store the <code>content-path-property</code> is the primary key for the table containing the item.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
</tbody>
</table>
<property>

<!ELEMENT property (derivation?, (option | attribute)*)>

Parent: <item descriptor>, <table>

A <property> tag can be a child of the <item-descriptor> tag or a <table> tag:

- If a child of an <item-descriptor> tag, <property> defines a transient property of the repository item. Because such a transient property is not associated with any database table, it is not stored when the repository item is updated in the database. Transient properties are readable and writable, but are not queryable. See the Transient Properties section of this chapter.

- If a child of a <table> tag, <property> defines a persistent property in a repository item. A <property> tag that is a direct child of an <item-descriptor> tag defines a transient characteristic of a repository item. Because such a transient property is not associated with any database table, it is not stored when the repository item is updated in the database.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The property name (required)</td>
</tr>
<tr>
<td>cache-mode</td>
<td>The caching mode for this property, one of the following: disabled inherit</td>
</tr>
<tr>
<td></td>
<td>A property's caching mode supersedes the item descriptor's caching mode. To restore the default caching mode, set cache-mode to inherit. See SQL Repository Caching.</td>
</tr>
<tr>
<td>cascade</td>
<td>One or more of the following, separated by commas: insert update delete</td>
</tr>
<tr>
<td></td>
<td>See Cascading Data Relationships.</td>
</tr>
<tr>
<td>category</td>
<td>Specifies a category that this property shares with other item properties. Item properties that belong to the same category can be grouped together in a user interface, rather than in alphabetical order according to their display-name attributes. See Grouping and Sorting Properties in the SQL Repository Item Properties chapter.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>category-resource</td>
<td>If a resource bundle is specified for this property with the tag <code>&lt;attribute name=resourceBundle&gt;</code>, this attribute specifies the resource bundle key to the property’s category. See Localizing SQL Repository Definitions.</td>
</tr>
<tr>
<td>column-names</td>
<td>The column name or names in the SQL database. Default: value of name</td>
</tr>
<tr>
<td>component-data-type</td>
<td>If data-type is set to an array, list, set or map of primitive values, this attribute specifies the primitive data type. The data-type can be any valid value other than array, list, set or map. Every element that the property references must be of this data type.</td>
</tr>
<tr>
<td>component-item-type</td>
<td>The name of another item descriptor referenced by this property. If data-type is set to array, list, set, or map, this attribute specifies the type of items that are referenced. All referenced items must be of the same base type.</td>
</tr>
<tr>
<td>data-type</td>
<td>Required unless item-type or property-type is set, one of the following: string, int, byte, array, big string, short, binary, set, enumerated, long, date, list, boolean, float, timestamp, map. See Data-type Mappings: Java and SQL in this section for information about how these values map to Java and SQL data types.</td>
</tr>
<tr>
<td>default</td>
<td>A default value for the property if none is supplied when the repository item is created. A default value cannot be set for multi-valued properties.</td>
</tr>
<tr>
<td>description</td>
<td>Optionally describes this property. Default: value of name</td>
</tr>
<tr>
<td>description-resource</td>
<td>If a resource bundle is specified for this property with the tag <code>&lt;attribute name=resourceBundle&gt;</code>, this attribute specifies the resource bundle key to the property’s description. See Localizing SQL Repository Definitions.</td>
</tr>
<tr>
<td>display-name</td>
<td>Optional, used to identify the property in the user interface. Default: value of name</td>
</tr>
<tr>
<td>display-name-resource</td>
<td>If a resource bundle is specified for this property with the tag <code>&lt;attribute name=resourceBundle&gt;</code>, this attribute specifies the resource bundle key to the property’s display name. See Localizing SQL Repository Definitions.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>editor-class</td>
<td>The Java class name of a PropertyEditor to use for this property. See the JavaBeans specification for a description of PropertyEditors.</td>
</tr>
<tr>
<td>expert</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>group</td>
<td>Specifies a group shared with other properties so they can be loaded in the same SELECT statement. The default group name is the table name.</td>
</tr>
<tr>
<td></td>
<td>You can set the group for a property to add or remove properties from these default groups. This gives you a simple way to optimize the SQL generated by the repository.</td>
</tr>
<tr>
<td>hidden</td>
<td>Boolean, if true, suppresses display in the ATG Control Center.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>item-type</td>
<td>The name of another item descriptor.</td>
</tr>
<tr>
<td></td>
<td>If the value of this property is another repository item, specifies the item descriptor type of that repository item. Required if the data-type or property-type attribute is not specified.</td>
</tr>
<tr>
<td>property-type</td>
<td>The Java class of a user-defined property. See User-Defined Property Types.</td>
</tr>
<tr>
<td>queryable</td>
<td>Boolean, can be true for transient properties only if the entire item descriptor is also transient. See Transient Properties in this chapter.</td>
</tr>
<tr>
<td></td>
<td>Default: true</td>
</tr>
<tr>
<td>readable</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>Default: true</td>
</tr>
<tr>
<td>repository</td>
<td>The Nucleus address of another repository, specifies that this property’s value refers to one or more repository items in the specified repository. If you specify a relative path, it is relative to this repository. See Linking between Repositories.</td>
</tr>
<tr>
<td>required</td>
<td>Boolean, must be set to true if the corresponding database column is defined as NOT NULL.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>sql-type</td>
<td>The SQL type of the corresponding column if it is different from the default type for the data-type, as specified under Data Type Mappings: Java and SQL.</td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>writable</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>Default: true</td>
</tr>
<tr>
<td>xml:id</td>
<td>Typically used for XML file combination, where elements with the same ID are regarded as the same element.</td>
</tr>
</tbody>
</table>

### Data Type Settings

The `data-type` attribute in a `<property>` tag defines the data type of a repository item property. A data type can be a primitive type or refer to an item descriptor type. If you want to define a property that refers to another item, use the `item-type` attribute to refer to that item’s item descriptor.

For multi-valued types, set `data-type` to `array`, `list`, `set`, or `map`. If the elements referenced by this property are primitives or user-defined property types, set their data type with the `component-data-type` attribute. Note that the SQL repository does not support multi-valued properties that reference binary type elements. If a multi-valued property references repository items, specify their item type with the property's `component-item-type` attribute. For user-defined properties, use the `property-type` attribute to specify the Java class of the property's type.

### Data Type Mappings: Java and SQL

The following table shows how the `data-type` attribute names for the primitive types correspond to Java object types and SQL data types. Some SQL data types vary according to your SQL implementation. You can explicitly specify a SQL data type mapping by setting the `sql-type` attribute.

<table>
<thead>
<tr>
<th>data-type value</th>
<th>Java object type</th>
<th>Recommended SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>xxx[]</td>
<td>none</td>
</tr>
<tr>
<td>big string</td>
<td>String</td>
<td>LONG VARCHAR, CLOB TEXT (MS)</td>
</tr>
<tr>
<td>binary</td>
<td>byte[]</td>
<td>BINARY, VARBINARY, IMAGE (MS) LONG RAW, BLOB (Oracle) BLOB (DB2)</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
<td>NUMERIC(1) TINYINT (MS)</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
<td>INTEGER</td>
</tr>
<tr>
<td>date</td>
<td>java.util.Date</td>
<td>DATETIME (MS) DATE (DB2, Oracle)</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
<td>DOUBLE (DB2, MS) NUMBER (Oracle)</td>
</tr>
<tr>
<td>enumerated</td>
<td>String</td>
<td>INTEGER</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
<td>FLOAT (DB2, MS) NUMBER (Oracle)</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>INTEGER</td>
</tr>
<tr>
<td>list</td>
<td>java.util.List</td>
<td>none</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
<td>NUMERIC (19) BIGINT (DB2, MS)</td>
</tr>
<tr>
<td>map</td>
<td>java.util.Map</td>
<td>none</td>
</tr>
<tr>
<td>set</td>
<td>java.util.Set</td>
<td>none</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
<td>INTEGER SMALLINT (DB2, MS)</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
<td>VARCHAR VARCHAR, CLOB (Oracle)</td>
</tr>
<tr>
<td>timestamp</td>
<td>java.sql.Timestamp</td>
<td>DATETIME (MS) DATE (Oracle 8i) TIMESTAMP (DB2, Oracle 9i)</td>
</tr>
</tbody>
</table>

**CLOB and BLOB constraints**

If you plan to use BLOBs (Binary Large Objects) or CLOBs (Character Large Objects), be sure that your database and JDBC driver work with the data and queries you plan to use. Comparison queries (=, ! =, <, <=, >, >=) do not work with BLOBs or CLOBs. Also, Oracle versions before 9.2 do not support pattern-match queries (CONTAINS, STARTS_WITH, ENDS_WITH) against CLOBs.

**<derivation>**

```xml
<!ELEMENT derivation (expression*) >
```

**Parent:** `<property>`

The `<derivation>` tag is used for derived properties. For detailed information on usage, see Derived Properties in the SQLRepository Data Models chapter.
### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>The derivation method to use for the derivation logic, set to one of the following:</td>
</tr>
<tr>
<td></td>
<td>- firstNonNull</td>
</tr>
<tr>
<td></td>
<td>- firstWithAttribute</td>
</tr>
<tr>
<td></td>
<td>- firstWithLocale</td>
</tr>
<tr>
<td></td>
<td>- alias</td>
</tr>
<tr>
<td></td>
<td>- union</td>
</tr>
<tr>
<td></td>
<td>- collectiveUnion</td>
</tr>
<tr>
<td></td>
<td>Default: firstNonNull</td>
</tr>
<tr>
<td>user-method</td>
<td>A user-defined derivation method to use.</td>
</tr>
<tr>
<td>override-property</td>
<td>The name of a property that, when set explicitly, overrides the derived property value that is otherwise used.</td>
</tr>
</tbody>
</table>

#### <option>

```xml
<!ELEMENT option EMPTY>
```

**Parent:** `<property>`

If a property's `data-type` property is set to `enumerated`, use `<option>` tags to indicate the possible values of the enumerated properties. For example:

```xml
<property name="gender" data-type="enumerated">
  <option value="male" code="0"/>
  <option value="female" code="1"/>
</property>
```

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value of the enumerated option.</td>
</tr>
<tr>
<td>code</td>
<td>The integer code that represents the enumerated option in the database.</td>
</tr>
</tbody>
</table>

If no code is specified, an appropriate code is generated by the SQL repository. The value of the `code` attribute is a sequential integer, with the first option beginning at 0.
If a resource bundle is specified for this property with the tag
\texttt{<attribute name=resourceBundle>}, this attribute specifies the resource bundle
key to the property option’s the string value. See \textit{Localizing SQL Repository
Definitions}.

Typically used for XML file combination, where elements with the same ID are
regarded as the same element.

\texttt{<attribute>}

\texttt{<!ELEMENT attribute EMPTY>}

\textbf{Parent:} \texttt{<item-descriptor>,<property>,<table>}

A Java Beans\texttt{PropertyDescriptor} can store an arbitrary set of name/value pairs called \textit{feature
descriptor attributes}. You can use the \texttt{<attribute>} tag in the SQL repository as a child of a \texttt{<property>}
or an \texttt{<item-descriptor>} tag to supply parameters that affect the behavior of properties or item types
in your repository definition.

The \texttt{<attribute>} tag is an empty tag that defines the parent’s feature descriptor value or values. This tag
lets you associate arbitrary name/\texttt{string} value pairs with any property or item type. The name/\texttt{string} value pairs
are added to the property descriptor via the \texttt{set Value} method of \texttt{java.beans.FeatureDescriptor},
and can later be used by the application. For example:

\begin{verbatim}
<property name="employeeNumber" data-type="string">
  <attribute name="PCCExpert" value="true" data-type="boolean"/>
</property>
\end{verbatim}

See \textit{User-Defined Property Types} for more information.

You can refer to values of Nucleus components with the \texttt{bean} attribute of the \texttt{<attribute>} tag. For example:

\begin{verbatim}
<attribute name="documentRootPath" bean="/atg/demo/QuincyFunds/repositories/FeaturesDataStore.relativePathPrefix"/>
\end{verbatim}

Attribute tags must be empty and have no child tags.

\textit{Attributes}
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>The name of the name/value pair. You can specify any name here and it is added to the list of feature descriptor attributes for your property.</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>The value of the name/value pair. The data type of this value is defined by the <code>data-type</code> attribute supplied to this tag. If no <code>data-type</code> attribute is provided, the value of the attribute is a string.</td>
</tr>
<tr>
<td><strong>data-type</strong></td>
<td>The primitive data-type of the value, one of the following:</td>
</tr>
<tr>
<td></td>
<td><code>string</code>* <code>int</code></td>
</tr>
<tr>
<td></td>
<td><code>byte</code> <code>short</code></td>
</tr>
<tr>
<td></td>
<td><code>date</code> <code>long</code></td>
</tr>
<tr>
<td></td>
<td><code>timestamp</code> <code>float</code></td>
</tr>
<tr>
<td></td>
<td><code>double</code></td>
</tr>
<tr>
<td>*default</td>
<td></td>
</tr>
<tr>
<td><strong>bean</strong></td>
<td>The name of a Nucleus component or property that is the value of the attribute. If a relative address is specified, the address is relative to the <code>Repository</code> component. See the Assigning FeatureDescriptorValues with the <code>&lt;attribute&gt;</code> Tag section in this chapter.</td>
</tr>
<tr>
<td><strong>Xml:id</strong></td>
<td>Typically used for XML file combination, where elements with the same ID are regarded as the same element.</td>
</tr>
</tbody>
</table>

#### `<table>`

```
<!ELEMENT table (property | attribute)*>
```

**Parent:** `<item-descriptor>`

The `<table>` tag specifies an SQL database table that store properties of repository items defined by this item descriptor.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>The table’s database name.</td>
</tr>
<tr>
<td><strong>id-column-names</strong></td>
<td>The name or names of the database columns that correspond to the repository ID.</td>
</tr>
<tr>
<td><strong>multi-column-name</strong></td>
<td>For multi-valued properties of type <code>array</code>, <code>list</code> or <code>map</code>, specifies which column to use to sort <code>array</code> or <code>list</code> elements, or <code>map</code> keys.</td>
</tr>
</tbody>
</table>
The table’s type, one of the following:

- primary
- auxiliary (default)
- multi

Typically used only in versioned repositories, this attribute is set to an integer between 1-9, which specifies the relationship of this table to other tables in a many-to-many relationship. In a two-sided many-to-many relationship, the table with the ‘second’ side should set this attribute to 2; the table with the ‘first’ side should set this attribute to 1.

In a versioned repository, set this attribute to 1 for the table that contains the asset_version column; set it to 2 for the table that contains the sec_asset_version table.

Default: 1

Typically used for XML file combination, where elements with the same ID are regarded as the same element.

**<expression>**

```xml
<!ELEMENT derivation (expression*)>
```

**Parent:** `<derivation>`

The `<expression>` tag encloses a repository item property name. One or more `<expression>` tags provide the sources of a derived property.

For detailed information on usage, see Derived Properties in the SQLRepository Data Models chapter.

**<rql-filter>**

```xml
<!ELEMENT rql-filter (rql, param*)>
```

**Parent:** `<item-descriptor>`

The `<rql-filter>` tag can be used to define a filter for database read operations. The `<rql-filter>` tag encloses a Repository Query Language (RQL) string that defines the filter query. See the Repository Filtering section in this chapter.
<named-query>

<!ELEMENT named-query (rql-query | sql-query)>

Parent: item-descriptor

<rql-query>

<!ELEMENT rql-filter (rql,param*)>

Parent: <named-query>

<rql>

<!ELEMENT rql (#PCDATA)>

Parent: <rql-query>,<rql-filter>

<param>

<!ELEMENT param EMPTY>

Parent: <rql-query>,<rql-filter>

<sql-query>

<!ELEMENT sql-query (query-name, sql, returns?, input-parameter-types?, dependencies?)>

Parent: <named-query>

This element defines a specific SQL statement to be used in the named query.

<sql>

<!ELEMENT sql (#PCDATA)>

Parent: <sql-query>

The body of this tag specifies the SQL string to be used in the named query.
For stored procedures, use the appropriate stored procedure invocation syntax along with the `stored-procedure` attribute in the `<sql>` tag:

```
<sql stored-procedure="true">
```

### `<input-parameter-types>`

```
<!ELEMENT input-parameter-types (#PCDATA)>
```

**Parent:** `<sql-query>`

The `<input-parameter-types>` element is a comma-separated list of class names that any parameters in the query must be an instance of. There must be as many class names as parameters.

### `<returns>`

```
<!ELEMENT returns (#PCDATA)>
```

**Parent:** `<sql-query>`

The body of this optional tag specifies a comma-separated list of Repository property names that are returned by this query.

### `<dependencies>`

```
<!ELEMENT dependencies (#PCDATA)>
```

**Parent:** `<sql-query>`

If any properties specified by the body of the `<dependencies>` tag are changed, this query is flushed from the query cache.

### `<transaction>`

```
<!ELEMENT transaction (add-item | update-item | print-item | remove-item | 
  transaction | query-items | remove-all-items | 
  export-items | load-items | rollback-transaction) *> 
```

**Parent:** `<gsa-template>,<transaction>`

You can use a `<transaction>` tag to group a set of test operation tags. A `<transaction>` tag takes no attributes. If a `<transaction>` tag appears inside of another transaction, the outer transaction is suspended while the inner transaction executes, and resumes when it ends.
<add-item> tags in this element are processed one at a time. They cannot make forward references to other items and no attempt is made to satisfy database integrity constraints (beyond that automatically done with the cascade operator). Use the <import-items> tag to load items with forward references.

By default, all test operation tags are enclosed in a single transaction. But to avoid database deadlocks, you should place all test operation tags inside <transaction> tags. For example, given this pattern:

```xml
<add-item item-descriptor="foo" id="1"/>
<transaction>
  <print-item item-descriptor="foo" id="1"/>
</transaction>
```

the <print-item> tag cannot find item 1 because that item is not yet committed. Also, you can run into deadlocks with this pattern if you try to access or modify items that may be locked by operations in the outer tag. Instead, use a pattern like this:

```xml
<transaction>
  <add-item item-descriptor="foo" id="1"/>
</transaction>
<transaction>
  <print-item item-descriptor="foo" id="1"/>
</transaction>
<rollback-transaction>
```

### <rollback-transaction>

```xml
<!ELEMENT rollback-transaction EMPTY>
```

**Parent:** <gsa-template>, <transaction>

The <rollback-transaction> tag is used only in the <transaction> test operation tag, to mark the transaction as rollback only. It must be empty and has no child tags or attributes.

### <add-item>

```xml
<!ELEMENT add-item (set-property*)>
```

**Parent:** <gsa-template>, <transaction>, <import-items>

**Attributes**
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item-descriptor</td>
<td>The name of the item descriptor to use when adding items to the repository (required)</td>
</tr>
<tr>
<td>id</td>
<td>The RepositoryId to use for the added item. The value must be unique among all items.</td>
</tr>
<tr>
<td>repository</td>
<td>The Nucleus address of the repository where the item is to be added, optional if the item is added to the base repository specified in the startSQLRepository command.</td>
</tr>
<tr>
<td>on-commit</td>
<td>Boolean. If set to <code>true</code>, indicates to add the item only after the transaction is committed.</td>
</tr>
<tr>
<td>skip-add</td>
<td>Boolean. If set to <code>true</code>, indicates not to add the item when the transaction is committed. Use this attribute to create transient items.</td>
</tr>
<tr>
<td>tag</td>
<td>Use this to add a new item with a guaranteed unique RepositoryId. You can refer to this item with this <code>tag</code> attribute in <code>print-item</code> and <code>update-item</code> tags within the same XML file. This is useful for writing test scripts that are run over and over again on the same database, each time operating on different items.</td>
</tr>
</tbody>
</table>

### `<update-item>`

```xml
<!ELEMENT update-item (set-property*)>
```

**Parent:** `<gsa-template>`, `<transaction>`

See [Updating Items](#) in the chapter [Developing and Testing an SQL Repository](#).

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item-descriptor</td>
<td>The name of the item descriptor to use when updating items (required)</td>
</tr>
<tr>
<td>id</td>
<td>Specifies a repository ID to use for this item. You must specify id or tag.</td>
</tr>
<tr>
<td>tag</td>
<td>If you added your item with an <code>&lt;add-item&gt;</code> tag using the <code>tag</code> attribute, an <code>&lt;update-item&gt;</code> tag in the same XML file can refer to that item with the <code>tag</code>.</td>
</tr>
<tr>
<td>skip-update</td>
<td>Boolean. If set to <code>true</code>, specifies to update the item with property changes only when the transaction is committed. Default: <code>false</code></td>
</tr>
</tbody>
</table>
<remove-item>

<!ELEMENT remove-item EMPTY>

The <remove-item> tag is a procedural tag for removing items from the repository.

Parent: <gsa-template>, <transaction>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item-descriptor</td>
<td>The item descriptor to use when removing an item (required)</td>
</tr>
<tr>
<td>id</td>
<td>RepositoryId of the item to remove (required)</td>
</tr>
<tr>
<td>tag</td>
<td>If you added your item with an &lt;add-item&gt; tag using the tag attribute, a &lt;remove-item&gt; tag in the same XML file can refer to that item with the tag.</td>
</tr>
<tr>
<td>remove-references-to</td>
<td>Boolean. If true, items that reference the item to remove are removed also.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
</tbody>
</table>

<remove-all-items>

The <remove-all-items> tag is a procedural tag for removing all items in the repository. This tag is enabled only if the system property atg.allowRemoveAllItems is set on application startup. You can set this property by adding -Datg.allowRemoveAllItems to the JAVA_ARGS in your <ATG10dir>/home/local/config/environment.bat or environment.sh file

<query-items>

<!ELEMENT query-items (#PCDATA)>

Parent: <gsa-template>, <transaction>

This tag performs queries against the repository. For detailed information, see Querying Items in the Developing and Testing an SQL Repository chapter.

This tag can also be used for loading caches. See Preloading Caches in the SQL Repository Caching chapter.
Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item-descriptor</td>
<td>The item descriptor to use when querying items in the repository (required)</td>
</tr>
<tr>
<td>query</td>
<td>Contains the RQL query to issue against item-descriptor. You can also specify the query in the tag body.</td>
</tr>
<tr>
<td>id-only</td>
<td>Boolean. If true, logs only the repository ID of the items returned by the query. Default: false</td>
</tr>
<tr>
<td>quiet</td>
<td>Boolean. If true, eliminates log messages for each item returned. Default: false</td>
</tr>
<tr>
<td>print-content</td>
<td>Boolean. If true, prints the content property of the repository items returned by the query. Default: false</td>
</tr>
</tbody>
</table>

<print-item>

<!ELEMENT print-item EMPTY>

Parent: <gsa-template>, <transaction>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item-descriptor</td>
<td>The item descriptor to use when printing an item (required). If you omit the id, tag, and path attributes, all items in this item descriptor are printed.</td>
</tr>
<tr>
<td>folder</td>
<td>The path name of the folder to print. When a folder is printed, each of its children is displayed, using the display-property attribute.</td>
</tr>
<tr>
<td>id</td>
<td>The ID to use for this item. If you do not set id, tag, or path all items in the descriptor are printed. Optional RepositoryId of item</td>
</tr>
<tr>
<td>path</td>
<td>Specifies an item or folder to print. When a folder is printed, each of its children is displayed, with the display-property attribute. Path name of item or folder</td>
</tr>
</tbody>
</table>
If you add your item with an `<add-item>` tag with the `tag` attribute, you can refer to that item in the same XML file with the `tag` attribute in the `<print-item>` tag.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>If you add your item with an <code>&lt;add-item&gt;</code> tag with the <code>tag</code> attribute, you can refer to that item in the same XML file with the <code>tag</code> attribute in the <code>&lt;print-item&gt;</code> tag.</td>
</tr>
<tr>
<td>print-content</td>
<td>Boolean. If set to <code>true</code>, prints the item’s entire content and its properties. Default: <code>false</code></td>
</tr>
</tbody>
</table>

The `<set-property>` tag is used only in the `<add-item>` and `<update-item>` test operation tags.

Parent: `<gsa-template>`, `<transaction>`

**Property-setting Syntax**

You specify to set properties as follows:

- To set the value of an Array, List, or Set property, use a comma-separated list of values:

  ```xml
  <set-property name="interests" value="fishing, fussing, wrassling"/>
  ```

- To set the value of a Map property, use a comma-separated list of `key=value` pairs:

  ```xml
  <set-property name="homes" value="Jefferson=Monticello, Jackson=Hermitage, Madison=Montpelier"/>
  ```

- To add or remove a value to a multi-valued property, use the Boolean `add` or `remove` attributes. For example, to add a value to the preceding example:

  ```xml
  <set-property name="homes" value="Buchanan=Wheatland" add="true"/>
  ```

- To set the value of a property that refers to another repository item, use the ID of the other repository item:

  ```xml
  <set-property name="bestBuddy" value="10022349_5"/>
  ```

- To set a property to null, use this form:

  ```xml
  <set-property name="foo" value="__NULL__"/>
  ```

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the property to set (required)</td>
</tr>
<tr>
<td>Value</td>
<td>The value to assign to the property (required)</td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Boolean. If true, add this value to a multi-valued property.</td>
</tr>
<tr>
<td>Remove</td>
<td>Boolean. If true, remove this value from a multi-valued property.</td>
</tr>
</tbody>
</table>

### <import-items>

```xml
<!ELEMENT import-items (add-item)*>  
```

**Parent:** `<gsa-template>`, `<transaction>`

The `<import-items>` tag is a procedural tag that can be used to add items to a repository in a more complex way than is possible with `<add-item>` tags in a `<transaction>` tag.

As child elements of `<import-items>`, `<add-item>` tags are processed differently than as children of a `<transaction>` tag in that they can have forward references. When the template is parsed, the parser makes three passes through the `<add-item>` tags in an `<import-items>` tag. On the first pass, the items are created. On the second pass, it sets required properties and properties that do not reference other items, then calls add-item. On the final pass, it sets any remaining properties and calls update-item if necessary.

### <export-items>

```xml
<!ELEMENT export-items EMPTY>  
```

**Parent:** `<gsa-template>`, `<transaction>`

The `<export-items>` tag is a procedural tag for exporting the data required to recreate one or more item descriptors. The data is exported as XML to standard output. Using this tag is similar to running the `startSQLRepository` script with the `-export` argument. See the `startSQLRepository` section in the Developing and Testing an SQL Repository chapter.

**Attributes**

<table>
<thead>
<tr>
<th>item-descriptors</th>
<th>Specifies a comma-separated list of one or more item descriptor names. For example:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>&lt;export-items item-descriptors=&quot;authors,books&quot;/&gt;</code></td>
</tr>
<tr>
<td></td>
<td>If none are specified, all item descriptors are exported.</td>
</tr>
</tbody>
</table>
By default, when you use `<export-items>` tag, all referenced item descriptors are automatically added to the list of item descriptors to be exported. If you use the `skip-references="true"` attribute, referenced item descriptors are added only if you affirmatively include them.

### `<load-items>`

```xml
<!ELEMENT load-items (#PCDATA)>
```

**Parent:** `<gsa-template>, <transaction>`

The `<load-items>` body is a comma-separated list of the repository IDs of the items that should be loaded into the item cache. Loading an item cache can improve performance, as it otherwise can take some time for the normal run of queries to fill the caches.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>item-descriptor</td>
<td>The item descriptor whose item cache should be loaded (required)</td>
</tr>
<tr>
<td>properties</td>
<td>A list of properties to cache. If no properties are specified, no properties of the items are cached.</td>
</tr>
<tr>
<td>load-all-items</td>
<td>Boolean. If set to <code>true</code>, the <code>&lt;load-items&gt;</code> tag loads all items for the given item descriptor, ignoring the list of repository IDs in the body of the tag. Default: <code>false</code></td>
</tr>
<tr>
<td>quiet</td>
<td>If this attribute is set to <code>true</code>, the <code>&lt;load-items&gt;</code> tag produces no output. Default: <code>false</code></td>
</tr>
</tbody>
</table>

### `<dump-caches>`

```xml
<!ELEMENT dump-caches EMPTY>
```

**Parent:** `<gsa-template>, <transaction>`

The `<dump-caches>` tag can be used to print out the contents of the item cache for one or more item descriptors.
**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dump-type</td>
<td>Set to one of the following:</td>
</tr>
<tr>
<td></td>
<td>- debug: Cached items are logged.</td>
</tr>
<tr>
<td></td>
<td>- queries: Creates a log entry consisting of the <code>&lt;load-items&gt;</code> tag that is used to reload the cache.</td>
</tr>
<tr>
<td></td>
<td>- both: Combines the output of debug and queries.</td>
</tr>
<tr>
<td>item-descriptors</td>
<td>A comma-separated list of one or more item descriptor names. If no item descriptors are specified, all item descriptor caches are exported.</td>
</tr>
</tbody>
</table>

For example, given this `<dump-caches>` tag:

```
<dump-caches item-descriptors="product" dump-type="queries"/>
```

The following output might be logged, if there are four product items in the cache:

```
============= START BUFFER PRECACHE =============
<load-items item-descriptor="product">
  prod100003, prod100002, prod100001, prod100001
</load-items>
============= END BUFFER PRECACHE =============
```

**<print-ddl>**

```
<!ELEMENT dump-caches EMPTY>
```

**Parent:** `<gsa-template>`, `<transaction>`

This tag prints the DDLs that are used and exports the data to standard output. Using this tag is similar to running the `startSQLRepository` script with the `-outputSQL` or `-outputSQLFile <file>` argument. See the `startSQLRepository` section in the Developing and Testing an SQL Repository chapter.

**Attributes**
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database-name</td>
<td>Specifies the database vendor so you can generate SQL appropriate for your</td>
</tr>
<tr>
<td></td>
<td>production database software; one of the following:</td>
</tr>
<tr>
<td></td>
<td>db2</td>
</tr>
<tr>
<td></td>
<td>microsoft</td>
</tr>
<tr>
<td></td>
<td>oracle</td>
</tr>
<tr>
<td></td>
<td>solid</td>
</tr>
</tbody>
</table>

### DTD for SQL Repository Definition Files

The DTD for SQL repository definition files is installed in the `<ATG10dir>/DAS/lib/classes.jar` archive. It can also be referenced with this URL:

http://www.atg.com/dtds/gsa/gsa_1.0.dtd

```xml
<!ENTITY % flag "(true | false)">
<!ELEMENT gsa-template (header?,
  item-descriptor | add-item | update-item | print-item | remove-item |
  transaction | development-line | query-items | remove-all-items |
  export-items | import-items | print-ddl | dump-caches | load-items)*>

<!ELEMENT header (name?, author*, version?, description?)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT author (#PCDATA)>

<!ELEMENT version (#PCDATA)>
```
<!ELEMENT description (#PCDATA)>

<!ENTITY % cache-mode "(disabled | simple | locked | distributed | distributedJMS | distributedHybrid )">

<!ENTITY % property-cache-mode "(disabled | inherit)">

<!ELEMENT item-descriptor ((property | table | attribute | named-query)*, rql-filter?, (property | table | attribute | named-query)*)>

<!ATTLIST item-descriptor
xml:id ID #IMPLIED
name CDATA #REQUIRED
display-name CDATA #IMPLIED
display-name-resource CDATA #IMPLIED
default %flag; "false"
super-type CDATA #IMPLIED
sub-type-property CDATA #IMPLIED
sub-type-value CDATA #IMPLIED
copy-from CDATA #IMPLIED
content %flag; "false"
folder %flag; "false"
use-id-for-path %flag; "false"
content-name-property CDATA #IMPLIED
content-path-property CDATA #IMPLIED
content-property CDATA #IMPLIED
content-length-property CDATA #IMPLIED
content-checksum-property CDATA #IMPLIED
folder-id-property CDATA #IMPLIED
last-modified-property CDATA #IMPLIED
display-property CDATA #IMPLIED
version-property CDATA #IMPLIED
hidden %flag; "false"
expert %flag; "false"
writeable %flag; "true"
description CDATA #IMPLIED
description-resource CDATA #IMPLIED
cache-mode %cache-mode; "simple"
id-space-name CDATA #IMPLIED
id-space-names CDATA #IMPLIED
text-search-properties CDATA #IMPLIED
item-cache-size CDATA #IMPLIED
item-cache-timeout CDATA #IMPLIED
item-expire-timeout CDATA #IMPLIED>
<!ENTITY %flag "false" "true">

<!-- Property tag - defines one property descriptor for an item descriptor -->
<!ELEMENT property (derivation?, (option | attribute)*)>
<!ATTLIST property
  xml:id ID #IMPLIED
  name CDATA #REQUIRED
  column-name CDATA #IMPLIED
  column-names CDATA #IMPLIED
  property-type CDATA #IMPLIED
  data-type CDATA #IMPLIED
  data-types CDATA #IMPLIED
  item-type CDATA #IMPLIED
  sql-type CDATA #IMPLIED
  sql-types CDATA #IMPLIED
  component-item-type CDATA #IMPLIED
  component-data-type CDATA #IMPLIED
  display-name CDATA #IMPLIED
  display-name-resource CDATA #IMPLIED
  description CDATA #IMPLIED
  description-resource CDATA #IMPLIED
  required %flag; "false"
  readable %flag; "true"
  writable %flag; "true"
  queryable %flag; "true"
  default CDATA #IMPLIED
  hidden %flag; "false"
  expert %flag; "false"
  editor-class CDATA #IMPLIED
  category CDATA #IMPLIED
  category-resource CDATA #IMPLIED
  cascade CDATA #IMPLIED
  repository CDATA #IMPLIED
  cache-mode %property-cache-mode; "inherit"
  group CDATA #IMPLIED
>

<!-- Derived properties have an associated derivation which specifies how the derived property values are derived -->
<!ELEMENT derivation (expression*)>
<!ATTLIST derivation
  method CDATA #IMPLIED
  user-method CDATA #IMPLIED
  override-property CDATA #IMPLIED
>
A derived property expression, when evaluated specifies a value used in deriving a derived property value.

```xml
<!ELEMENT expression (#PCDATA)>
```

Defines a table for an item descriptor.

```xml
<!ELEMENT table (property | attribute)*>
<!ATTLIST table
  xml:id          ID      #IMPLIED
  name  CDATA       #REQUIRED
  multi-column-name   CDATA         #IMPLIED
  type                (primary|auxiliary|multi)     "auxiliary"
  id-column-name      CDATA          #IMPLIED
  id-column-names     CDATA          #IMPLIED
  shared-table-sequence   (1|2|3|4|5|6|7|8|9)     "1"
>```

Options are possible values for enumerated attributes.

```xml
<!ELEMENT option EMPTY>
<!ATTLIST option
  xml:id          ID      #IMPLIED
  value         CDATA           #IMPLIED
  resource      CDATA           #IMPLIED
  bean          CDATA           #IMPLIED
  code          CDATA           #IMPLIED>
```

The attribute tag is used to specify the list of feature descriptor values.

```xml
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
  xml:id          ID      #IMPLIED
  name         CDATA           #REQUIRED
  value        CDATA           #IMPLIED
  bean         CDATA           #IMPLIED
  data-type    CDATA           #IMPLIED>
```

This tag specifies an RQL statement to be used as a filter for an item descriptor.

```xml
<!ELEMENT rql-filter (rql,param*)>
```

RQL query string itself.

```xml
<!ELEMENT rql (#PCDATA)>
```

RQL query parameters.

```xml
<!ELEMENT param EMPTY>
<!ATTLIST param
  name               CDATA       #IMPLIED
  value              CDATA       #IMPLIED
  bean               CDATA       #IMPLIED>
<!-- The named-query element. This specifies an association between a user-defined name and a Query representation -->
<!ELEMENT named-query (rql-query | sql-query)>

<!-- The rql-query element. Identifies an association between a user-defined name and an RQL query string, that can later be retrieved by name from the corresponding repository view that this tag is found under -->
<!ELEMENT rql-query (query-name, rql)>

<!-- The sql-query element. Identifies an association between a user-defined name and an SQL query string, that can later be retrieved by name from the corresponding repository view that this tag is found under -->
<!ELEMENT sql-query (query-name, sql, returns?, input-parameter-types?, dependencies?)>

<!ELEMENT sql (#PCDATA)>
<!ATTLIST sql
  stored-procedure %flag; #IMPLIED
>
<!ELEMENT returns (#PCDATA)>
<!ELEMENT input-parameter-types (#PCDATA)>
<!ELEMENT dependencies (#PCDATA)>

<!-- The query-name element, which indicates the user-defined name of a named query instance -->
<!ELEMENT query-name (#PCDATA)>

<!-- The transaction element. It surrounds the operation elements add-item, print-item etc. Note that add-item tags in this element are processed one at a time. They cannot make forward references to other items and no attempt is made to satisfy database integrity constraints (beyond that automatically done with the cascade operator) Use the import-items tag if you want to load items with forward references. -->
<!ELEMENT transaction (add-item | update-item | print-item | remove-item | transaction | query-items | remove-all-items | export-items | load-items | rollback-transaction)?>

<!-- The development-line element. It surrounds the operation elements add-item, print-item etc. Note that add-item tags in this element are processed one at a time. They cannot make forward references to other items and no attempt is made to satisfy database integrity constraints (beyond that automatically done with the cascade operator) Use the import-items tag if you want to load items with forward references. -->
<!ELEMENT development-line (add-item | update-item | print-item | remove-item | transaction | query-items | remove-all-items | export-items | load-items)*>

<!ATTLIST development-line
The import-items element. This tag only contains add-item tags. These
tags can contain forward references. The tags are processed in three
passes - pass one creates all items. Pass two, sets required properties
and optional properties which do not refer to other items. Pass three
sets the remaining properties and updates the item.

<!ELEMENT import-items (add-item)*>

<!-- Procedural tags for adding and modifying items -->
<!ELEMENT add-item (set-property*)>
<!ATTLIST add-item
  item-descriptor CDATA #REQUIRED
  id              CDATA #IMPLIED
  tag             CDATA #IMPLIED
  on-commit       CDATA #IMPLIED
  skip-add        CDATA #IMPLIED
  repository      CDATA #IMPLIED
  no-checkin      %flag; "false"

<!-- Procedural tags for adding and modifying items -->
<!ELEMENT update-item (set-property*)>
<!ATTLIST update-item
  item-descriptor CDATA #REQUIRED
  id              CDATA #IMPLIED
  tag             CDATA #IMPLIED
  skip-update     CDATA #IMPLIED

<!-- Procedural tag for removing an item -->
<!ELEMENT remove-item EMPTY>
<!ATTLIST remove-item
  item-descriptor CDATA #REQUIRED
  id              CDATA #IMPLIED
  tag             CDATA #IMPLIED
  remove-references-to %flag; "false"

<!-- Procedural tag for removing all items. Only enabled if the system
property atg.allowRemoveAllItems is set on startup -->
<!ELEMENT remove-all-items EMPTY>
<!ATTLIST remove-all-items
  item-descriptor CDATA #REQUIRED
<!-- Procedural tag for exporting the data required to recreate one or more
item-descriptors. The item-descriptors attribute specifies a comma
separated list of one or more item descriptor names. If none are
specified, all item-descriptors are exported -->
<!ELEMENT export-items EMPTY>
<!ATTLIST export-items
  item-descriptors CDATA #IMPLIED
  skip-references %flag; "false"
>
<!-- Procedural tag for querying and printing an item -->
<!ELEMENT query-items (#PCDATA)>
<!ATTLIST query-items
  item-descriptor CDATA #REQUIRED
  query CDATA #IMPLIED
  print-content CDATA #IMPLIED
  quiet %flag; "false"
  id-only %flag; "false"
>
<!-- Procedural tag for caching a list of items -->
<!ELEMENT load-items (#PCDATA)>
<!ATTLIST load-items
  item-descriptor CDATA #REQUIRED
  properties CDATA #IMPLIED
  load-all-items %flag; "false"
  quiet %flag; "false"
>
<!-- Procedural tag for printing an item -->
<!ELEMENT print-item EMPTY>
<!ATTLIST print-item
  item-descriptor CDATA #IMPLIED
  path CDATA #IMPLIED
  folder CDATA #IMPLIED
  id CDATA #IMPLIED
  tag CDATA #IMPLIED
  print-content CDATA #IMPLIED
>
<!-- Sets a property value. Used only in the add-item and update-item tags -->
<!ELEMENT set-property (#PCDATA)>
<!ATTLIST set-property
  name CDATA #REQUIRED
  value CDATA #IMPLIED
  add %flag; "false"
  remove %flag; "false"
>
<!-- Sets a property value. Used only in the add-item and update-item tags -->

Sample SQL Repository Definition Files

This section includes a number of simple examples of SQL repository definition files and the corresponding SQL statements to create the tables described by the definition files. These examples demonstrate a variety of data relationship mappings:

- **Simple One-to-One** maps a repository item to a single table row. It includes just a primary table, with no joins with other tables. This is the simplest case.
- **One-to-One with Auxiliary Table** maps a repository item to a primary table and an auxiliary table (a one-to-one relationship). Each user has a job title and function.
- **One-to-Many with an Array** maps a repository item to a primary table and a multi-value table with an array property. This demonstrates a one-to-many relationship. The multi table, named `subjects_tbl`, contains a list of a user’s favorite subjects (simple strings). When using an array or list type property, the multi table requires a multi-column-name attribute (in this example, `seq_num`) to ensure that the ordering of the multi-values are maintained.
- **One-to-Many with a Set** maps a repository item to a primary table and a multi-value table with a set type property. This is another example of one-to-many relationship. Because a set is used, a multi-column-name attribute is not required.
- **One-to-Many with a Map** maps a repository item to a primary table and a multi-value table with a map property. When using a map type property, the multi table requires a multi-column-name attribute (in this example, `card_key`). This column contains keys that uniquely identify each of the multi-values. For example, each user has many
credit cards; the keys are strings that identify each of the user’s cards (like business card, frequent flyer card, personal card.

- **One-to-Many Mapping to Other Repository Items** maps a one-to-many relationship. It defines two item types, user and address. Each user can have many addresses.

- **Ordered One-to-Many** demonstrates an ordered one-to-many relationship with a list type property. It defines two item types, author and book. Each author can have many books, and the order of the books is considered significant.

- **Many-to-Many** maps a many-to-many relationship. It defines two item types, user and address. Each user can have many addresses. Many users may live at the same address.

- **Multi-Column Repository IDs** demonstrates the use of composite repository IDs.

**Simple One-to-One**

This example maps a repository item to a single table row. It includes just a primary table, with no joins with other tables.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template
  PUBLIC "-//Art Technology Group, Inc.//DTD Dynamo Security//EN"
  "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Repository Example Version A</name>
    <author>Pat Durante</author>
    <description>
      This template maps a repository item to a single table row. Just a primary table...no joins with other tables. Simplest case.
    </description>
  </header>

  <item-descriptor name="user" default="true">
    <table name="usr_tbl" type="primary" id-column-names="id">
      <property name="id" data-type="string"/>
      <property name="name" column-names="nam_col" data-type="string"/>
      <property name="age" column-names="age_col" data-type="int"/>
    </table>
  </item-descriptor>
</gsa-template>
```
SQL Statements

drop table usr_tbl;

CREATE TABLE usr_tbl |
  id                      VARCHAR(32)     not null,  
  nam_col                 VARCHAR(32)     null,  
  age_col                 INTEGER null,  
  primary key(id) |
|

One-to-One with Auxiliary Table

This example maps a repository item to a primary table and an auxiliary table (a one-to-one relationship).

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template PUBLIC "-//Art Technology Group, Inc.//DTD Dynamo Security//EN" "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Repository Example Version B</name>
    <author>Pat Durante</author>
    <description>
      This template maps a repository item to a primary table and an auxiliary table (a one-to-one relationship.) Each user has a job title and function.
    </description>
  </header>
  <item-descriptor name="user" default="true">
    <table name="usr_tbl" type="primary" id-column-names="id">
      <property name="id" data-type="string"/>
      <property name="name" column-names="nam_col" data-type="string"/>
      <property name="age" column-names="age_col" data-type="int"/>
    </table>
    <table name="job_tbl" type="auxiliary" id-column-names="id">
      <property name="function"/>
      <property name="title"/>
    </table>
  </item-descriptor>
</gsa-template>
SQL Statements

```sql
DROP TABLE usr_tbl;
DROP TABLE job_tbl;

CREATE TABLE usr_tbl (
  id          VARCHAR(32)     NOT NULL,
  name_col    VARCHAR(32)     NULL,
  age_col     INTEGER         NULL,
  PRIMARY KEY (id)
);

CREATE TABLE job_tbl (
  id          VARCHAR(32)     NOT NULL REFERENCES usr_tbl(id),
  function    VARCHAR(32)     NULL,
  title       VARCHAR(32)     NULL,
  PRIMARY KEY (id)
);
```

One-to-Many with an Array

This example maps a repository item to a primary table and a multi-value table with an array property. This demonstrates a one-to-many relationship.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template PUBLIC "-//Art Technology Group, Inc.//DTD Dynamo Security//EN"
  "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Repository Example Version C</name>
    <author>Pat Durante</author>
    <description>
      This template maps a repository item to a primary table and a multi-value table using an array property. A one-to-many relationship. The "multi" table contains a list of a user's favorite subjects (simple strings). When using an "array" property, the "multi" table requires a "multi-column-name" (e.g., seq_num) to ensure that the ordering of the multi-values are maintained.
    </description>
  </header>

  <item-descriptor name="user" default="true">
    <table name="usr_tbl" type="primary" id-column-names="id">
      <property name="id" data-type="string"/>
    </table>
    <table name="job_tbl" type="multi" id-column-names="id">
      <property name="function" data-type="string"/>
      <property name="title" data-type="string"/>
    </table>
  </item-descriptor>
</gsa-template>
```
<table>
  <tashington)
  <table name="subjects_tbl" type="multi" id-column-names="id"
    multi-column-name="seq_num">
    <property name="favoriteSubjects" column-names="subject" data-type="array"
      component-data-type="string"/>
  </table>
</item-descriptor>
</gsa-template>

SQL Statements

drop table usr_tbl;
drop table subjects_tbl;

CREATE TABLE usr_tbl (
  id                      VARCHAR(32)     not null,
  nam_col                 VARCHAR(32)     null,
  age_col                 INTEGER null,
  primary key(id)
);

CREATE TABLE subjects_tbl (
  id                      VARCHAR(32)     not null references usr_tbl(id),
  seq_num                 INTEGER not null,
  subject                 VARCHAR(32)     null,
  primary key(id, seq_num)
);

One-to-Many with a Set

This example maps a repository item to a primary table and a multi-value table with a set type property.

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template
  PUBLIC "-//Art Technology Group, Inc.//DTD Dynamo Security//EN"
  "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Repository Example Version D</name>
    <author>Pat Durante</author>
    <description>
      This template maps a repository item to a primary table and a multi-value table using a set property.
A one-to-many relationship. Since we are using a "set", we are not required to use a "multi-column-name" attribute. Demonstrates that D5 repositories (unlike D4.5) do not require a "seq_num" column.

SQL Statements

```
drop table subjects_tbl;
drop table usr_tbl;

CREATE TABLE usr_tbl (  
id                      VARCHAR(32)     not null,
nam_col                 VARCHAR(32)     null,
age_col                 INTEGER null,
primary key(id)
);

CREATE TABLE subjects_tbl (  
id                      VARCHAR(32)     not null references usr_tbl(id),
subject                 VARCHAR(32)     not null,
primary key(id, subject)
);
```

One-to-Many with a Map

This example maps a repository item to a primary table and a multi-value table with a map property.
This template maps a repository item to a primary table and a multi-value table using a map property. A one-to-many relationship. When using a "map" property, the "multi" table requires a "multi-column-name" (e.g., card_key). This column will contain keys that uniquely identify each of the multi-values (for example, each user has many credit cards... the keys are strings that identify each of the user's cards (like business card, frequent flyer card, personal card.)

```
CREATE TABLE usr_tbl (
    id                VARCHAR(32)     not null,
    nam_col           VARCHAR(32)     null,
    age_col           INTEGER     null,
    primary key(id)
);```

```
CREATE TABLE credit_card_tbl (id, card_key)
```

```
DROP TABLE credit_card_tbl;
DROP TABLE usr_tbl;
```
CREATE TABLE credit_card_tbl (
    id                VARCHAR(32)     not null references usr_tbl(id),
    card_key          VARCHAR(32)     not null,
    card_num          VARCHAR(32)     null,
    primary key(id, card_key)
) ;

CREATE INDEX credit_card_tbl_idx ON credit_card_tbl(id);

One-to-Many Mapping to Other Repository Items

This example maps out a one-to-many relationship between user items and address items. It demonstrates the use of the component-item-type attribute, which allows one repository item to contain other repository items. Each user item can contain many address items, such as home address, shipping address, business address.

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template
PUBLIC "-//Art Technology Group, Inc.//DTD Dynamo Security//EN"
"http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Repository Mapping Example Version F</name>
    <author>Ben Erwin</author>
    <description>This template maps out a one-to-many relationship between user items and address items. It demonstrates the use of the component-item-type attribute (which allows one repository item to contain other repository items.) Each user item will contain many address items (home address, business address, etc.)</description>
  </header>
  <item-descriptor name="address">
    <table name="addr_tbl" type="primary" id-column-name="addr_id">
      <property name="user" column-name="user_id" item-type="user"/>
      <property name="street" data-type="string"/>
      <property name="city" data-type="string"/>
    </table>
  </item-descriptor>
  <item-descriptor name="user" default="true">
    <table name="usr_tbl" type="primary" id-column-name="id">
      <property name="id" data-type="string"/>
      <property name="name" column-name="nam_col" data-type="string"/>
      <property name="age" column-name="age_col" data-type="string"/>
    </table>
  </item-descriptor>
</gsa-template>
SQL Statements

CREATE TABLE usr_tbl |
id VARCHAR(32) not null,
  nam_col VARCHAR(32) null,
  age_col VARCHAR(32) null,
  primary key(id)
|

CREATE TABLE addr_tbl |
  addr_id VARCHAR(32) not null,
  user_id VARCHAR(32) null references usr_tbl(id),
  street VARCHAR(32) null,
  city VARCHAR(32) null,
  primary key(addr_id)
|

Ordered One-to-Many

Another data model you can use in the SQL Repository is an ordered one-to-many relationship. Suppose you have an author item descriptor and you want to model each author’s books in the order they were published. Your SQL repository definition file can define two item descriptors that look something like this:

<i策 item-descriptor name="author">
  <table name="author" type="primary" id-column-name="author_id">
    |
  </table>
  <table name="book" type="multi" id-column-name="author_id"
    multi-column-name="sequence_num">
    <property name="books" data-type="list" component-item-type="book"
      column-name="book_id"/>
  </table>
</i策 item-descriptor>
<i策 item-descriptor name="book">
  <table name="book" type="primary" id-column-name="book_id">
    <property name="author" item-type="author" column-name="author_id"/>
    <property name="seq" data-type="int" column-name="sequence_num"/>
  </table>
</i策 item-descriptor>
Note some limitations for this data model:

- You must use the `List` data type to represent the ordered "many" side of the relationship.
- The `sequence_num` and `author_id` columns in the `book` table cannot be specified as not null, as the SQL Repository tries to set these fields to null when items in the List are removed.
- The `book` item descriptor needs to define a property to point to the `sequence_num` field, like this:

  ```xml
  <property name="seq" data-type="int" column-name="sequence_num"/>
  ```

### SQL Statements

```sql
CREATE TABLE author (
    author_id VARCHAR(32) not null,
    primary key(author_id)
);

CREATE TABLE book (
    book_id VARCHAR(32) not null,
    sequence_num INTEGER,
    author_id VARCHAR(32) references author(author_id),
    primary key(book_id)
);
```

### Many-to-Many

This example maps out a many-to-many relationship. It defines two item types, `user` and `address`. Each user can have many addresses. Many users may live at the same address.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template
 PUBLIC "-//Art Technology Group, Inc.//DTD Dynamo Security//EN"
 "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">

<gsa-template>
  <header>
    <name>People Repository Version H</name>
    <author>Pat Durante</author>
    <description>
      This template maps out a many-to-many relationship
      between user items and address items. Each user can
      have many addresses. Many users may live at the
      same address.
    </description>
  </header>
```

224

12 - SQL Repository Reference
<item-descriptor name="address">
  <table name="addr_tbl" type="primary" id-column-names="address_id">
    <property name="street" data-type="string"/>
    <property name="city" data-type="string"/>
  </table>
</item-descriptor>

<table name="user_address_tbl" type="multi" id-column-names="addr_id">
  <property name="users" column-names="user_id" data-type="set" component-item-type="user"/>
</table>
</item-descriptor>

=item-descriptor name="user" default="true">
  <table name="usr_tbl" type="primary" id-column-names="id">
    <property name="id" data-type="string"/>
    <property name="name" column-names="nam_col" data-type="string"/>
    <property name="age" column-names="age_col" data-type="int"/>
  </table>
  <table name="user_address_tbl" type="multi" id-column-names="user_id">
    <property name="addresses" column-names="addr_id" data-type="set" component-item-type="address"/>
  </table>
</item-descriptor>
</gsa-template>

**SQL Statements**

drop table addr_tbl;
drop table user_address_tbl;
drop table usr_tbl;

CREATE TABLE addr_tbl (  
  address_id VARCHAR(32)  not null,
  street VARCHAR(32)  null,
  city VARCHAR(32)  null,
  primary key(addr_id)
);
CREATE TABLE user_address_tbl (  
  addr_id VARCHAR(32)  not null references addr_tbl(address_id),
  user_id VARCHAR(32)  not null references usr_tbl(id),
  primary key(addr_id, user_id)
);

CREATE INDEX user_address_tbl_user_idx ON user_address_tbl(user_id);

CREATE TABLE usr_tbl |
Multi-Column Repository IDs
This example demonstrates the use of multi-column or composite repository IDs.

```sql
id VARCHAR(32) not null,
nam_col VARCHAR(32) null,
age_col INTEGER null,
primary key(id)
);
```

---

Multi-Column Repository IDs
This example demonstrates the use of multi-column or composite repository IDs.

```xml
<item-descriptor name="typeX" id-separator=":">
  <table name="TYPEX" type="primary" id-column-names="TYPEX_ID">
    <property name="id" column-names="TYPEX_ID" data-type="string" />
    <property name="name" column-names="NAME" data-type="string" />
  </table>
  <table name="TYPEXY" type="multi" id-column-names="TYPEX_ID">
    <property name="typeXYs" component-item-type="typeXY"
      column-names="TYPEX_ID,TYPEY_ID" data-type="set" />
  </table>
</item-descriptor>

<item-descriptor name="typeY" id-separator=":">
  <table name="TYPEY" type="primary" id-column-names="TYPEY_ID">
    <property name="id" column-names="TYPEY_ID" data-type="string" />
    <property name="name" column-names="NAME" data-type="string" />
  </table>
</item-descriptor>

<item-descriptor name="typeZ" id-separator=":">
  <table name="TYPEZ" type="primary" id-column-names="TYPEZ_ID">
    <property name="id" column-names="TYPEZ_ID" data-type="string" />
    <property name="name" column-names="NAME" data-type="string" />
  </table>
</item-descriptor>

<item-descriptor name="typeXY" id-separator=":">
  <table name="TYPEXY" type="primary" id-column-names="TYPEX_ID,TYPEY_ID">
    <property name="id" column-names="TYPEX_ID,TYPEY_ID" data-types="string,string" />
    <property name="name" column-names="NAME" data-type="string" />
    <property name="x" column-names="TYPEX_ID" item-type="typeX" />
    <property name="y" column-names="TYPEY_ID" item-type="typeY" />
  </table>
  <table name="TYPEXYZ" type="multi" id-column-names="TYPEX_ID,TYPEY_ID">
    <property name="typeXYZs" component-item-type="typeXYZ"
      column-names="TYPEX_ID,TYPEY_ID,TYPEZ_ID" data-type="set" />
  </table>
</item-descriptor>
```
<item-descriptor name="typeXYZ" id-separator=":">
<table name="TYPEXYZ" type="primary"
    id-column-names="TYPEX_ID,TYPEY_ID,TYPEZ_ID">
    <property name="id" column-names="TYPEX_ID,TYPEY_ID,TYPEZ_ID"
data-types="string,string,string" />
    <property name="name" column-names="NAME" data-type="string" />
    <property name="x" column-names="TYPEX_ID" item-type="typeX" />
    <property name="y" column-names="TYPEY_ID" item-type="typeY" />
    <property name="z" column-names="TYPEZ_ID" item-type="typeZ" />
    <property name="xy" column-names="TYPEX_ID,TYPEY_ID" item-type="typeXY" />
</table>
</item-descriptor>

SQL Statements

drop table TYPEXYZ;
drop table TYPEXY;
drop table TYPEZ;
drop table TYPEY;
drop table TYPEX;

CREATE TABLE TYPEX (  
    TYPEX_ID          VARCHAR(32)     not null,
    NAME               VARCHAR(32)     null,
    primary key(TYPEX_ID)
);

CREATE TABLE TYPEY (  
    TYPEY_ID          VARCHAR(32)     not null,
    NAME               VARCHAR(32)     null,
    primary key(TYPEY_ID)
);

CREATE TABLE TYPEZ (  
    TYPEZ_ID          VARCHAR(32)     not null,
    NAME               VARCHAR(32)     null,
    primary key(TYPEZ_ID)
);

CREATE TABLE TYPEXY (  
    TYPEX_ID                VARCHAR(32)     not null,
    TYPEY_ID                VARCHAR(32)     not null,
    NAME                    VARCHAR(32)     null,
    primary key(TYPEX_ID, TYPEY_ID),
    foreign key (TYPEX_ID) references TYPEX(TYPEX_ID),
    foreign key (TYPEY_ID) references TYPEY(TYPEY_ID)
);
Configuring the SQL Repository Component

Each SQL repository is a component of class `atg.adapter.gsa.GSARepository`. This class implements `atg.repository.MutableRepository`, `atg.repository.content.ContentRepository` and extends `atg.repository.RepositoryImpl`. The ATG platform includes a sample SQL Repository component with a Nucleus address of `/atg/dynamo/service/jdbc/SQLRepository`. You can use this component, or create your own. An ATG instance can have any number of SQL Repository components running at the same time.

Registering a Content Repository

Content repositories must be added to the list of repositories in the `initialRepositories` property of the `/atg/registry/ContentRepositories` component. This also causes the new repository to show up in the Content window of the ATG Control Center. To cause a repository to appear instead in the Portal or the Commerce window of the ACC, edit the Repository Editor’s definition in the `/atg/devtools/admins.xml` file. This XML file should be placed in the application’s configuration path `atg/devtools`. In the `/atg/devtools/admins.xml` file, set `task` to the ACC task area where you want the repository to appear. For example:

```xml
<custom-admin id="CustomProductCatalog">
  <display-name>My Product Catalog</display-name>
  <task>commerce</task>
  ...
</custom-admin>
```

The repository is displayed in the ATG Control Center under the name specified by the `<display-name>` tag. The repository’s `repositoryName` property must match the value specified by the `<repository-name>` tag in the `/atg/devtools/admins.xml` file. For example:

```xml
<default-admin id="StandardProductCatalog" xml-combine="replace">
  <display-name>Catalog Elements (En)</display-name>
  <task>commerce</task>
```

CREATE TABLE TYPEXYZ(
  TYPEX_ID          VARCHAR(32)     not null,
  TYPEY_ID          VARCHAR(32)     not null,
  TYPEZ_ID          VARCHAR(32)     not null,
  NAME              VARCHAR(32)     null,
  primary key(TYPEX_ID, TYPEY_ID,TYPEZ_ID),
  foreign key (TYPEX_ID) references TYPEX(TYPEX_ID),
  foreign key (TYPEY_ID) references TYPEY(TYPEY_ID),
  foreign key (TYPEZ_ID) references TYPEZ(TYPEZ_ID)
);
<repository-name>ProductCatalog</repository-name>
<folder-view>true</folder-view>
<create-bean-displays>
...
</create-bean-displays>
<standard-bean-displays>
...
</standard-bean-displays>
</default-admin>

### SQL Repository Component Properties

An SQL Repository component is derived from the class `atg.adapter.gsa.GSARepository` and includes the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allowNullValues</td>
<td>Boolean, specifies whether to allow null values in multi-valued properties:</td>
</tr>
<tr>
<td></td>
<td>false (default): null values are automatically removed from multi-valued properties; an attempt to add null value to this property yields an exception.</td>
</tr>
<tr>
<td></td>
<td>true: null values can be set in multi-valued properties.</td>
</tr>
<tr>
<td></td>
<td>You can also enable null values for individual multi-valued properties by setting the <code>allowNullValues</code> attribute to <code>true</code> in its <code>&lt;property&gt;</code> tag.</td>
</tr>
<tr>
<td>autoCommitInitialization</td>
<td>If <code>setAutoCommit</code> and <code>localTransactionModelInitialization</code> are both true, JDBC connections are explicitly set with the value of <code>autoCommitInitialization</code>. Otherwise JDBC connections are left as is.</td>
</tr>
<tr>
<td></td>
<td>Default: true</td>
</tr>
<tr>
<td>autoUpdateSubscribers</td>
<td>For distributed TCP caching mode, should ATG automatically populate the <code>das_gsa_subscriber</code> database table?</td>
</tr>
<tr>
<td></td>
<td>Default: true</td>
</tr>
<tr>
<td>cacheRestoreFile</td>
<td>If <code>restoreCacheOnRestart</code> is true, an XML file used to reload item caches on restart is written to this location.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>cacheSwitchHot</code></td>
<td>If a target site is configured for switch deployment, specifies whether to pre-populate repository caches before the data source is switched. If this property is set to <code>true</code>, the repository prepopulates an on-deck set of caches with data from the next <code>DataSource</code> before the switch occurs.</td>
</tr>
<tr>
<td></td>
<td>Default: <code>false</code></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This property must be set to <code>false</code> if you use Content Administration for deployment. For information about optimizing switch deployment caching, see the ATG Content Administration Programming Guide.</td>
</tr>
<tr>
<td><code>checkTables</code></td>
<td>If set to <code>true</code>, the <code>GSARepository</code> verifies each database table with a simple SQL query at application startup. To skip the validity check and achieve faster startups, set this to <code>false</code>.</td>
</tr>
<tr>
<td></td>
<td>Default: <code>false</code></td>
</tr>
<tr>
<td><code>databaseName</code></td>
<td>This property is used by the <code>startSQLRepository</code> script. Do not change its value.</td>
</tr>
<tr>
<td><code>databaseTableInfo</code></td>
<td>This property is used by the <code>startSQLRepository</code> script. Do not change its value.</td>
</tr>
<tr>
<td><code>dataSource</code></td>
<td>This refers to a <code>DataSource</code> (<code>javax.sql.DataSource</code>) to use for obtaining connections. DataSources should typically implement resource pooling for best performance.</td>
</tr>
<tr>
<td></td>
<td>This property is typically set as follows:</td>
</tr>
<tr>
<td></td>
<td><code>/atg/dynamo/service/jdbc/pool-name</code></td>
</tr>
<tr>
<td><code>debugLevel</code></td>
<td>An integer value that indicates the detail of debugging messages printed out when the Repository's <code>loggingDebug</code> property is set to <code>true</code>. Higher values generate more messages. The range is from 0-15.</td>
</tr>
<tr>
<td></td>
<td>You can also set the debug level for an individual item descriptor or property in the Dynamo Administration Interface or with the <code>loggingDebug</code> attribute tag. See Debug Levels in the Developing and Testing an SQL Repository chapter.</td>
</tr>
<tr>
<td></td>
<td>Default: 5</td>
</tr>
<tr>
<td><code>definitionFiles</code></td>
<td>The location of the repository definition XML files, specified as an absolute name on the application configuration path. ATG uses XML file combination to collate multiple definition files into a single repository definition.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>disableItemCachesAtStartup</td>
<td>If true the repository disables all item caches when it starts up. This overrides all item cache size settings in the definition file. The caches can still be turned on later programmatically. This is mostly for debugging.</td>
</tr>
<tr>
<td>disableQueryCachesAtStartup</td>
<td>If true the repository disables all query caches when it starts up. This overrides all query cache size settings in the definition file. The caches can still be turned on later programmatically. This is mostly for debugging.</td>
</tr>
<tr>
<td>enforceRequiredProperties</td>
<td>If true, the repository checks to make sure all required properties are present when adding repository items and forbids the setting of a required property to null.</td>
</tr>
<tr>
<td>escapeWildcards</td>
<td>The characters % and _ are typically treated as wildcards in database queries. If this property is set to true, the GSARepository uses an escape character before % and _ in all pattern-match queries. The one exception is when a pattern-match query is used to simulate a text search query, as in that case, wildcards should be allowed to be passed through. The escape character is specified by the wildcardEscapeCharacter property and the default value is \.</td>
</tr>
<tr>
<td>eventServer</td>
<td>The event server component that handles cache invalidation messages for item descriptors that use distributed TCP caching mode.</td>
</tr>
<tr>
<td>groupContainer</td>
<td>If you want to define profile groups or content groups, set this to the RepositoryGroups component. See the ATG Personalization Programming Guide for more information about profile groups and content groups.</td>
</tr>
<tr>
<td>idGenerator</td>
<td>An IdGenerator to use for generating unique IDs for items.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| itemDescriptorAliases        | A map that you can use to allow one item descriptor to be accessed by more than one name. You configure it as a `Map` that maps the alias to the existing item descriptor name that is its equivalent. For example, this setting allows the name `All Profiles` to be used to refer to the item descriptor named `user`:  
  `itemDescriptorAliases=All Profiles=user`                                                                                                                      |
| loadItemBatchSize            | The maximum number of items to load from the database at one time. This property is consulted by `getItems()` and the hot cache switching logic.  
  Default: 200                                                                                                                                                    |
| localeSensitiveSorting      | If `true`, sorted query results are sorted in a locale sensitive manner. More specifically, String values are compared using `java.text.Collator`. Because most databases cannot handle sorting with multiple locales, setting this option to `true` also means that the repository performs all sorting in memory. If `false`, database sorting (via `ORDER BY`) is used where applicable and Strings are compared using `String.compareTo()`. If database sorting is adequate for your purposes, leaving this property set to `false` provides better performance.  
  Default: `false`                                                                                                                                              |
| localTransactionModelInitializaton | If `true`, use local transaction mode for initializing the service. Some database/JDBC driver combinations require this mode for JDBC meta-data queries when the `GSARepository` initializes. If `false`, a `TransactionDemarcation` with mode `REQUIRED` is used.  
  Default: `true`                                                                                                                                              |
| lockManager                  | A `ClientLockManager` to use for locked mode caching. See the `SQL Repository Caching` chapter.                                                                                                          |
| metaDataSchemaPattern        | The name of the database account that was used to create the tables that underlie the repository. See `Table Ownership Issues`.  
  Default: `DYNAMO`                                                                                                                                                |
<p>| metaDataCatalogName          | The name of a metadata catalog. See <code>Table Ownership Issues</code> in the <code>SQL Repository Queries</code> chapter.                                                                                                  |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>outerJoinSupport</code></td>
<td>Configures the syntax to use for outer joins. See Outer Joins in the SQL Repository Queries chapter in the SQL Repository Queries chapter for valid settings.</td>
</tr>
<tr>
<td><code>pathSeparator</code></td>
<td>Change this property only if paths in your content folders use a separator different than the default <code>/</code> (forward slash).</td>
</tr>
<tr>
<td><code>prohibitCollectionDuplicates</code></td>
<td>Boolean, specifies whether this repository allows duplicate values in ordered multi-valued properties of type List and Array.</td>
</tr>
<tr>
<td></td>
<td>For example, if this property is set to <code>true</code>, you cannot set duplicate values in the String list property <code>myList</code>. Thus, attempts to update the datastore with the following additions will yield an error on the third duplicate item:</td>
</tr>
<tr>
<td></td>
<td><code>myList.add(&quot;one&quot;);</code> \n<code>myList.add(&quot;two&quot;);</code> \n<code>myList.add(&quot;one&quot;);</code></td>
</tr>
<tr>
<td></td>
<td>This setting can be overriden by individual properties (see Prohibiting Duplicate Values in the chapter SQL Repository Item Properties).</td>
</tr>
<tr>
<td></td>
<td>Default: <code>false</code></td>
</tr>
<tr>
<td><code>repositoryName</code></td>
<td>The repository name.</td>
</tr>
<tr>
<td><code>restoreCacheOnRestart</code></td>
<td>If <code>true</code>, the repository automatically dumps the contents of its item caches when it is stopped and reloads the same items into the caches when it is started again. Tags that reload the caches are written into the file specified by the <code>cacheRestoreFile</code> property.</td>
</tr>
<tr>
<td></td>
<td>Default: <code>false</code></td>
</tr>
<tr>
<td><code>safeSQLTypes</code></td>
<td>A comma-separated list of SQL types for which the repository always uses the default JDBC type. You can set this property to string values of SQL types like <code>varchar</code>, or to the corresponding integer values specified in the class <code>java.sql.Types</code> (for example, <code>-4</code>).</td>
</tr>
<tr>
<td></td>
<td>Default: <code>null</code></td>
</tr>
<tr>
<td><code>selectiveCacheInvalidationEnabled</code></td>
<td>Boolean, if set to <code>true</code> enables selective cache invalidation. For more information, see the ATG Content Administration Programming Guide.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setAutoCommit</td>
<td>If true, the Repository calls Connection.setAutoCommit() as needed. If false, the repository does not call that API. Some JDBC drivers, due to bugs, may cause errors in the GSARespository.initialize() method unless this property is set to false. If you need to set it to false, set autoCommitInitialization or localTransactionModeInitialization to false. Default: false</td>
</tr>
<tr>
<td>simulateTextSearchQueries</td>
<td>If true, substitute pattern match queries for text search queries. This setting is not supported for production ATG applications. See Text Search Queries. Default: false</td>
</tr>
<tr>
<td>SQLLowerFunction</td>
<td>The name of the SQL function to use to lower-case an expression. This is used for case-insensitive querying. If this property is null, no attempt is made to lower-case database expressions. Default: lower</td>
</tr>
<tr>
<td>storeTransientItems</td>
<td>If true, the getItem method returns items that are cached, but not yet added. Default: true</td>
</tr>
<tr>
<td>subscriberRepository</td>
<td>If you use distributed TCP caching mode, ATG maintains an item descriptor for the das_gsa_subscriber table. This property specifies which repository that item descriptor belongs to. By default, this item descriptor is in the /atg/dynamo/service/jdbc/SQLRepository repository. If for any reason you desire to use a different repository instance, you must make sure that each repository that uses distributed TCP caching mode has the same value for its subscriberRepository property. Default: /atg/dynamo/service/jdbc/SQLRepository</td>
</tr>
<tr>
<td>synchronousInvalidationEvents</td>
<td>For distributed TCP caching mode, should invalidation events be sent asynchronously, for better performance, or synchronously, to avoid a slight window of stale cache? Default: false</td>
</tr>
<tr>
<td>tablePrefix</td>
<td>A string that is prepended to the table name when inserts or updates are made. See Table Ownership Issues.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>transactionManager</td>
<td>A TransactionManager to use for all transactions. All code in the same server typically use the same TransactionManager.</td>
</tr>
<tr>
<td></td>
<td>Default: /atg/dynamo/transaction/TransactionManager</td>
</tr>
<tr>
<td>updateSchemaInfoCache</td>
<td>If true, the Repository creates files that store the SQL type for each column in the database schema.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>useCacheForDelete</td>
<td>If true, the Repository tries to optimize certain SQL delete operations based on the values in the cache. For certain usage patterns, such as when there are many multi-valued properties, setting this to true can result in a significant performance gain. Set this property to true only when (a) you define a version property for each item descriptor or (b) you use locked caching mode. Setting this property causes it to be set in each of the item descriptors defined in the Repository.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>userPropertyDescriptors</td>
<td>The Java class names of user defined property descriptors that should be loaded for this repository. User defined property descriptors register themselves in a static system-wide table. This property enables you to ensure that these classes are loaded before the repository loads any XML definitions that might refer to them.</td>
</tr>
<tr>
<td>useSetAsciiStream</td>
<td>If useSetAsciiStream is set to true, the SQL repository always uses setAsciiStream() instead of setString() in prepared statements. You can useSetAsciiStream instead of useSetUnicodeStream, but you lose the ability to handle internationalized values in the database.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>useSetBinaryStream</td>
<td>If useSetBinaryStream is set to true, the SQL repository always uses setBinaryStream() instead of setBytes() in prepared statements. The setBinaryStream() is required for large byte arrays in some JDBC drivers.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>useSetObject</td>
<td>If useSetObject is set to true, the SQL repository always uses setObject() instead of setInt(), setFloat(), setDouble(), or setString() in prepared statements.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>useSetUnicodeStream</td>
<td>If useSetUnicodeStream is set to true, the SQL repository always uses setUnicodeStream() instead of setString() in prepared statements. The setUnicodeStream() method is required for large Strings in some JDBC drivers. Setting useSetUnicodeStream to true is recommended if you use Oracle with internationalized content, but is not recommended if you do not have internationalized content in your database. Note that if you use MS SQL Server, you must set useSetUnicodeStream to false.</td>
</tr>
<tr>
<td>Default: true</td>
<td></td>
</tr>
<tr>
<td>useTransactionsForCachedReads</td>
<td>By default, the SQL repository does not use transactions when reading from the cache. This improves performance. To disable this optimization, set this property to true.</td>
</tr>
<tr>
<td>Default: false</td>
<td></td>
</tr>
<tr>
<td>wildcardEscapeCharacter</td>
<td>This character is used in queries to escape characters that are otherwise treated as wildcards. See the description of the escapeWildcards property.</td>
</tr>
<tr>
<td>Default: \</td>
<td></td>
</tr>
<tr>
<td>XMLToDomParser</td>
<td>The parser used to parse the XML definition file. This value is read-only</td>
</tr>
<tr>
<td>Default: atg.xml.tools.XMLToDomParser</td>
<td></td>
</tr>
<tr>
<td>XMLToolsFactory</td>
<td>An XMLToolsFactory to use in parsing XML templates.</td>
</tr>
<tr>
<td>Default: /atg/dynamo/service/xml/XMLToolsFactory</td>
<td></td>
</tr>
</tbody>
</table>
13 SQL Content Repositories

A content repository comprises repository items that correspond to documents maintained in a hierarchical name space. A content repository typically serves as a source of content items to display to a user, directly or as an element in a page.

An SQL repository implemented through the Generic SQL Adapter connector can act as a content repository, storing content items that are displayed in pages. Because the GSARepository class implements two interfaces—atg.repository.Repository and atg.repository.content.ContentRepository—and a repository can contain multiple repository item types, a single repository can contain both content repository items (arranged in a hierarchical structure with folders that can contain repository items and other folders) and non-content repository items (arranged in a flat structure).

You can use a content repository to serve targeted content, as described in the Creating Rules for Targeting Content and Setting Up Targeting Services chapters of the ATG Personalization Programming Guide. A product catalog in a commerce application is also typically a content repository, as described in the Using and Extending the Default Catalog chapter of the ATG Commerce Programming Guide.

Note that the essential feature of a content repository is that it represents a hierarchical structure of folders and repository items, like a directory structure. The repository items themselves do not necessarily represent content that is displayed in a Web application, although in most cases they do. What is significant is whether the repository items are maintained in a hierarchical structure.

You can define one or more item descriptors in an SQL repository to be a content item descriptor that defines a type of ContentRepositoryItem. When you retrieve one of these items by calling any Repository methods, the repository item implements the atg.repository.content.ContentRepositoryItem interface. You can have other item descriptors in the same repository that do not implement this interface and do not define content items.

The Repository Loader is a utility that handles the work of creating and updating content repository items from documents on your file system. The repository template can be configured so the loader assigns the values of your content repository item’s properties from selected portions of these documents while still allowing access to the entire document. These properties include metadata about the document file such as its length and the time it was last modified. The Repository Loader can be configured to periodically scan the file system and synchronize it with the repository representation, adding, updating and deleting content repository items as necessary. See the Repository Loader chapter for more information.
Setting Up an SQL Content Repository

You can think of a content repository item as consisting of content and metadata. For example, if your content repository includes repository items that are news stories, the metadata might include a story’s byline, dateline, length, and keywords, while the content includes the text of the story itself. You can adopt one of two basic architectural styles when you set up an SQL content repository:

- You can store both the content and the metadata in your SQL database. A content repository item includes a property whose value was the content.
- You can store the metadata in the database, and the content in your file system. In this case, the metadata includes properties that indicate how to look up the content in the file system. A content repository item includes a property whose value was a pointer to the content in the file system.

As with other repositories, setting up an SQL content repository involves the following steps:

1. Design the item types you want to include in your content repository. For each type of repository item, decide what sorts of properties you want to have available to you for searching and targeting content in the repository.
2. Set up an SQL database containing content repository items, to act as the data store of the repository.
3. Create a repository definition. This is an XML file that describes the repository’s item descriptors and property descriptors, and defines the relationship among these items and the rows and tables of the database. See Creating an SQL Content Repository Definition.
4. Configure an SQL Repository component that interacts with the data store you set up in step 2 to create, modify, and retrieve repository items. See Configuring an SQL Content Repository.

A repository that contains content items must include item descriptors flagged with the folder and content attributes of the <item-descriptor> tag in the SQL repository definition.

Creating an SQL Content Repository Definition

An SQL content repository is an implementation of the Generic SQL Adapter. Its repository definition follows the SQL repository definition file syntax described in the SQL Repository Definition Tag Reference in the SQL Repository Reference chapter. In particular, an SQL content repository is characterized by the <item-descriptor> attributes described in the Content Item Attributes section.

Note the following points that are particular to SQL content repositories:

- A repository that contains content items must include one item descriptor flagged with the folder and one or more item descriptors flagged with the content attributes of the <item-descriptor> tag in the SQL repository definition. See Folder and Content Item Descriptors.
The folder item descriptor and the content item descriptors must define properties that define the pathname of each folder and content item. These properties are used to retrieve the content item and identify the place of each folder or content item in the content repository’s hierarchical namespace. See Path and Item ID Attributes.

Folder and Content Item Descriptors

An SQL content repository must contain:

- One item descriptor that defines repository items that act as folders (the folder item descriptor)
- One or more item descriptors that define content repository items (the content item descriptors).

Items defined by the content item descriptor implement the atg.repository.content.ContentRepositoryItem interface. Items defined by the folder item descriptor implement the atg.repository.content.FolderItem interface, as well as the atg.repository.MutableRepositoryItem interface.

Path and Item ID Attributes

The folder and content item descriptors must define properties that represent the name or path of the items. These properties must be mapped directly to columns of the database so queries can be performed against them.

You can use one of three different techniques to specify how path information is stored in the database:

- use-id-for-path
- content-name-property
- content-path-property

Regardless of how you store path information in the database, you can get the path of an item with this method in the atg.repository.content.FolderItem interface (which is extended by the ContentRepositoryItem interfaces):

```java
public String getItemPath()
```

This method returns the path of this item, represented as a relative path from the repository’s root folder.

**use-id-for-path**

This is the simplest mechanism. In this mode, the relative path name of the item is used as the ID of the repository item. Your database must then include an ID column that is a string large enough to hold the entire relative path name of each folder item and content item. Put this ID column in your primary table and set the id-column-name attribute of the primary table to point to this column. You then set use-id-for-path="true" for that item descriptor. For example:
The use-id-for-path mode may not work if you have an existing database schema that does not follow this format. This approach also might not work if path names in your repository are longer than the size of varchar you can efficiently store and query against in your database. Some databases impose a 255-character limit on the size of queryable columns. This may be too small to hold the entire path for some content repositories.

Note that even though you put the entire path name in the property designated by the id-column-names attribute, you still need to use the folder-id-property attribute to designate a property that holds the name of the parent folder of the item. In the preceding example, the folder-id property holds the name of the folder.

content-name-property

You can set the item descriptor’s content-name-property attribute. In this case, you can store just the name of the repository item itself (rather than the entire path name) in one property of the repository item and use the content-name-property to designate the name of this property. The content-name-property specifies the property representing the name of the folder item or content item, while the folder-id-property specifies the property representing the parent folder of the folder item or content item. From these two pieces of information let you compute the path for a given item by walking up the content hierarchy.

The operation of computing the path for this item is more expensive in this mode, because you query up the folder hierarchy to compute the path for an item rather than get the path from a single property. However, this mode can overcome the problem of the size limitation on queryable columns. Now the column size for the content name limits the size of each individual component of the file name, not the size of the entire path.

A folder-id-property is required for all content repositories, whichever method they use to specify how path information is stored in the database. The data type of the folder-id-property can be data-type=’string’ (or whatever type you define your repository IDs to be), or you can specify that its item-type is the name of the folder item descriptor. This enables you to conveniently access folder information from the item itself. For example:
Because this `content-name` property is not guaranteed to be unique across the repository, you have a separate column for the ID of this repository.

**content-path-property**

You might be unable to use the repository item's path as its repository ID. If that is the case, perhaps due to references to these rows from other tables, and if you can store the entire path name of the item as a column in your table, you can use a third alternative. In this mode, you can set the `content-path-property` to refer to a property that holds the path name of the item. You then use a separate property and column in your table to refer to the ID for this item. For example:

```xml
<item-descriptor name="folder" folder="true"
    content-path-property="pathname">
    <table name="folder" id-column-names="id">
        <property name="id" data-type="long"/>
        <property name="pathname" data-type="string"/>
    </table>
</item-descriptor>
```

---

**Defining Content Item Descriptors**

The `<item-descriptor>` tags in a repository definition file can include a set of attributes that are specific to content repositories. These attributes are described in the Content Item Attributes section in the SQL Repository Reference chapter. These `<item-descriptor>` tag attributes are:

- `content`
- `folder`
- `content-name-property`
- `content-path-property`
- `use-id-for-path`
- `folder-id-property`
- `content-property`
- `content-length-property`
- `last-modified-property`
Content Attributes and Properties

A content item descriptor must define a `content-property` property and optionally can define a `lastModified` or `length` property. The `content-property` attribute specifies the name of an item descriptor property that is used to store or reference the content data itself. The content property is usually a `java.io.File`, `String` or a `byte[]` data type.

These properties can be implemented as user-defined properties so they can be computed at run time. This approach enables them to be taken from the file system, not stored in the database. For example:

```xml
<item-descriptor name="files" content-length-property="length" last-modified-property="lastModified" content-property="data">
  <table name="media_files" type="auxiliary" id-column-names="media_id">
    <property name="length" data-type="long" column-names="length"/>
    <property name="lastModified" data-type="timestamp" column-names="last_modified"/>
    <property name="data" data-type="binary" column-names="data"/>
  </table>
</item-descriptor>
```

You configure your content item descriptors by naming the properties to use to retrieve each of these values. This is done with the following attributes in the `<item-descriptor>` tag:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>content-path-property</td>
<td>Specifies the ID of the folder containing this folder or content item.</td>
</tr>
<tr>
<td>content-name-property</td>
<td>Refers to the name of this folder or content item (not the full path name)</td>
</tr>
<tr>
<td>content-property</td>
<td>For content item descriptors, this is the name of the property that holds the content itself.</td>
</tr>
<tr>
<td>last-modified-property</td>
<td>For content item descriptors, this optionally is used to specify a property that can be used to retrieve the last modified time for that piece of content.</td>
</tr>
</tbody>
</table>

Storing Content on a File System

If you want to keep the repository content on a file system rather than in your database, you can use a property descriptor, `atg.repository.FilePropertyDescriptor`, as the property type of your `content-property`. The `FilePropertyDescriptor` is a simple read-only property descriptor that takes a path name and converts it to a `java.io.File` object. Use a `pathPrefix` attribute in the property definition to specify the parent directory of the content.

In the following example, the `articleText` property is defined with the `atg.repository.FilePropertyDescriptor` property type. The `pathPrefix` is defined as `/tmp`: 
Content Repository Example

This section demonstrates the design and configuration of a simple repository. For another example, see the Repository Loader Example in the Repository Loader chapter. In this example, the repository supports a Web site that talks about books. Each book has an author, a title, a cover illustration, and a descriptive text.

The book as a business entity has a corresponding repository item type in this example. Its attributes are maintained as properties of the book item type. The book item type is defined in the example repository definition file with this item descriptor element:

```xml
<item-descriptor name="book" display-property="title" content="true" content-property="bookcover_image" content-path-property="filePath" folder-id-property="parentFolder">
    <table name="book_info" id-column-name="id" type="primary">
        <property name="filePath" data-type="big string"/>
        <property name="parentFolder" item-type="book_folder"/>
        <property name="title" data-type="big string"/>
        <property name="author" data-type="big string"/>
    </table>
    <property name="bookcover_image" property-type="atg.repository.FilePropertyDescriptor"/>
</item-descriptor>
```
**Book Item Type Properties**

The book item descriptor contains the following properties:

- **author** and **title** are strings, with one-to-one relationships between books and authors and books and titles. More complex relationships are possible, of course.

- **bookcover_image** is defined outside the `<table>` tag, which indicates that the property is transient and maintained outside the database. The property is defined as a file type, maintained on the file system, as follows:
  
  ```
  property-type="atg.repository.FilePropertyDescriptor"
  ```

  As described in the Storing Content on a File System section, this property type indicates that the repository should use the property's path name and convert it to a `java.io.File` object.

**Locating the Content with Path and Folder Properties**

In order to keep a hierarchical directory structure, define a folder item type named `book_folder`:

```
<item-descriptor name="book_folder" display-property="folderPath" folder="true" content-path-property="folderPath" folder-id-property="parentFolder">
  <table name="book_folder" id-column-name="id" type="primary">
    <property name="parentFolder" item-type="book_folder"/>
    <property name="folderPath" data-type="big string"/>
  </table>
</item-descriptor>
```

This item type is specified to be a folder with the attribute `<folder="true">`. The `<folder-id-property>` attribute in the item-descriptor tag indicates that this item stores its parent folder ID in the database with a property named `parentFolder`.

The book and the `book_folder` item types store their paths in the database with the `content-path-property` attribute. The `content-path-property` attribute indicates the property of this item that defines the absolute path name of this item in the folder hierarchy. In this example, the path is stored in the property named `folderPath`. If the example has especially deep hierarchies, resulting in excessively long path names, it might instead store just the name of this item with the `content-name-property` attribute, and have the repository calculate the item's absolute path by determining its parent folders, with the property indicated by the `folder-id-property` attribute.

**Book Example Repository Definition File**

The complete example repository definition file used in this example is as follows:
Book Example SQL Table Creation Statements

The following SQL creates the tables used by the book example content repository:

```sql
-- drop table book_folder;
-- drop table book_info;
CREATE TABLE book_folder ( id, parentFolder, folderPath )
CREATE TABLE book_info ( id, parentFolder, folderPath, title, author, bookcover_image )
```
Adding Content to the Content Repository

You can add items to the content repository with the tags described in the Developing and Testing an SQL Repository chapter. For a more scalable method of adding items to the repository, see the Repository Loader chapter.

For example, the following three add-item tags:

1. Create a book_folder item named /.
   
   ```xml
   <add-item item-descriptor="book_folder" id="folder:/">
     <set-property name="folderPath" value="/"/>
   </add-item>
   ```

2. Create a subfolder of / named foo, with a path of /foo.
   
   ```xml
   <add-item item-descriptor="book_folder" id="folder:/foo">
     <set-property name="parentFolder" value="/"/>
     <set-property name="folderPath" value="/foo"/>
   </add-item>
   ```

3. Create a book item titled bar, with a path of /foo/bar.
   
   ```xml
   <add-item item-descriptor="content" id="content:/foo/bar">
     <set-property name="parentFolder" value="/foo"/>
     <set-property name="filePath" value="/foo/bar"/>
     <set-property name="title" value="bar"/>
   </add-item>
   ```

Accessing Items in the Content Repository

After you have set up a content repository, you can use it to serve targeted content, as described in the Creating Rules for Targeting Content and Setting Up Targeting Services chapters of the ATG Personalization Programming Guide. You can also search for text in content items as described in the Text Search Queries section of the SQL Repository Queries chapter.
You can also get repository items programmatically, given a repository ID:

```java
// repository id of the item we want to get
String id = "1001";

// name of item descriptor describing the type of item we want
String descriptorName = "book";

// get the item from the repository
RepositoryItem item = pRepository.getItem(id, descriptorName);

// make sure we have an item
if (item == null)
{
    pln("Item not found, descriptor=" + descriptorName + ", id=" + id);
    return;
}

// get the author property of the item
String author = (String)item.getPropertyValue("author");
```

**Configuring an SQL Content Repository**

The Repository component for an SQL content repository is a standard SQL Repository component of class `atg.adapter.gsa.GSARepository`. The Configuring the SQL Repository Component section of the SQL Repository Reference chapter describes the properties you can configure.
14 Repository Loader

For many development environments, it often makes sense to create the content of repository items directly on a file system, then load those items into an ATG repository. The ATG Repository Loader provides a flexible way to take files that are stored in a file system, convert them into repository items, and load the items into the repository.

The Repository Loader can load into an SQL repository HTML files, XML files, and binary media such as image or audio files. The Repository Loader can transform files into XML files, then transform the XML files into repository items.

You can configure the Repository Loader to perform the load operation in two ways:

- Scan the file system and identify the files to load, then load the files on command or on a specified schedule.
- Submit a manifest file to the Repository Loader that specifies the files to load.

Scanning the file system requires less administrative effort; using a manifest file incurs less system overhead. The number of files to scan and load generally determines which option is best.

This chapter includes the following sections:

- Repository Loader Architecture
- Repository Loader Components
- Repository Loader Administration
- RLClient
- Importing Versioned Repository Data
- Repository Loader Example

Repository Loader Architecture

The Repository Loader is implemented by the following components:

- FileSystemMonitorScheduler sets up a schedule to start the FileSystemMonitorService, and whether to scan recursively.
- FileSystemMonitorService specifies which files to scan for upload to the repository.
- **LoaderManager** accepts repository loader jobs. It maintains a queue of pending jobs, and uploads the job for type mapping processing.

- **TypeMapper and TypeMappings** map file types to specific content handlers.

- **ContentHandlers** transform the file into a repository item and invoke a back end system to perform add, update, and remove operations on the repository.

**Repository Loader interfaces**

ATG provides two interfaces to invoke and manage the Repository Loader:

- **Repository Loader Administration** is a Web-based interface that lets you start and manage repository load jobs. With this interface, you interactively identify the directories whose contents you wish to load into the repository.

- **RLClient** is a command line RMI utility; it requires a manifest file that identifies the files to load into the repository.

A fully-configured Repository Loader setup can be represented as follows:
Repository Loader Components

The following sections describe Repository Loader components in detail—their interfaces and classes, and properties.

**FileSystemMonitorScheduler**

You can configure a FileSystemMonitorScheduler component to initiate scheduled file system scans by the FileSystemMonitorService. If enabled, this component initiates the scan according to the specified schedule. If set to false, the load process must be initiated by the Repository Loader Administration or the RLClient.

**FileSystemMonitorScheduler properties**

The FileSystemMonitorScheduler has the following configurable properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>A Boolean property, determines whether the scheduler is enabled.</td>
</tr>
<tr>
<td>fileSystemManager</td>
<td>Specifies the implementation of interface fileSystemManager.</td>
</tr>
<tr>
<td>lastScannedStorage</td>
<td>Specifies the URI of a file that stores the time of the last file scan:</td>
</tr>
<tr>
<td>recursiveScan</td>
<td>A Boolean property, specifies whether the FileSystemMonitorService recursively scans the supplied paths.</td>
</tr>
<tr>
<td>schedule</td>
<td>The schedule for scanning the file system. For example:</td>
</tr>
<tr>
<td>scheduler</td>
<td>The Nucleus address of the Scheduler component, which initiates the file scan. If set to null, no rescanning occurs.</td>
</tr>
</tbody>
</table>

For information about valid formats for this property, see the Core Dynamo Services chapter of the ATG Programming Guide.

Default:

- **enabled**: false
- **fileSystemManager**: FileSystemMonitorService
- **lastScannedStorage**: {serverHomeDirResource?resourceURI=data/rl/FileSystemMonitorLastScan.txt}
- **recursiveScan**: true
- **schedule**: every 2 hours in 15 minutes
- **scheduler**: /atg/dynamo/service/Scheduler
FileSystemMonitorService

The FileSystemMonitorService class implements two interfaces:

- `atg.repository.loader.FileSystemMonitor`
- `atg.repository.loader.FileSystemManager`

A component of the FileSystemMonitorService class is configured to scan a specified file directory and identify the files to upload. A FileSystemMonitorService can be invoked by the FileSystemMonitorScheduler or the Repository Loader Administration.

You can configure a FileSystemMonitorService so it uploads only files that meet one or more of the following criteria:

- Reside in a specified folder and, optionally, its subfolders
- Modified since the time the file system was last scanned
- Named with specific file extensions

After the FileSystemMonitorService identifies the files to upload, it supplies these files to a LoaderManager component as an array of files, or in the form of a manifest file if the component’s `createManifestMode` property is set to true.

**FileSystemMonitorService properties**

A FileSystemMonitorService component has the following configurable properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createManifestMode</code></td>
<td>A Boolean property, set to true to indicate that a large file system is being processed. The FileSystemMonitorService passes files to the LoaderManager in the form of a manifest file, rather than as an array of files. You can supply the name of the manifest file by setting the <code>manifestFile</code> property. Default: false</td>
</tr>
<tr>
<td><code>escapeManifestUsingCDATASection</code></td>
<td>A Boolean property, set to true if <code>createManifestMode</code> property is set to true and file system names contain special characters such as ampersand (&amp;) that must be escaped in the generated manifest’s XML. Default: false</td>
</tr>
<tr>
<td><code>filters</code></td>
<td>A String array of file extension strings to use as filter criteria when gathering updates. Only files whose file extensions match one of the strings in this property are scanned. For example: <code>filters = .html, .htm, .wml</code></td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>includeFolders</code></td>
<td>A Boolean property, specifies whether to include the content folders in the scan results. Set to <code>true</code> in order for folders to be created as folder repository items in a content repository. Default: <code>true</code></td>
</tr>
<tr>
<td><code>lastScanned</code></td>
<td>A timestamp property, specifies the last time the FileSystemMonitorService executed. Default: 0 (load all files)</td>
</tr>
<tr>
<td><code>loaderManager</code></td>
<td>The Nucleus address of the LoaderManager component.</td>
</tr>
<tr>
<td><code>manifestFile</code></td>
<td>The manifest file name to use if <code>createManifestMode</code> is set to true. Default: <code>RLxxxx.xml</code></td>
</tr>
<tr>
<td><code>rootPath</code></td>
<td>The root path to monitor. All files to scan must be in or under this path.</td>
</tr>
<tr>
<td><code>typeMapper</code></td>
<td>The Nucleus address of the TypeMapper component. This setting has precedence over the LoaderManager configuration's defaultTypeMapper setting. If a manifest file specifies its own TypeMappings, those have precedence.</td>
</tr>
</tbody>
</table>

### LoaderManager

The `LoaderManagerImpl` class implements the interface `atg.repository.loader.LoaderManager`. A LoaderManager component of type `LoaderManagerImpl` accepts Repository Loader jobs. Repository jobs can be initiated from two sources:

- `FileSystemMonitorService`
- `RLClient` (Repository Loader RMI client)

In both cases, the LoaderManager component passes the files to a TypeMapper component, which determines how to process them. Alternatively, the TypeMapping can be specified directly or in a manifest file.

### Setting the repository path separator on Windows

Before submitting an import job to the LoaderManager on Windows, the target repository's `pathSeparator` property must be set to backslash (`\`). After the import completes, reset this property to forward slash (`/`).

**Note:** If you set this `pathSeparator` directly in a properties file rather than in the ACC, use this format:
**LoaderManager Properties**

A LoaderManager component has the following configurable properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cancelledJobsFifoSize</code></td>
<td>The number of cancelled jobs that can be kept in the queue and are viewable in the Repository Loader Administration. Default: 0</td>
</tr>
<tr>
<td><code>completedJobsFifoSize</code></td>
<td>The number of completed jobs that should be kept in the queue. Default: 20</td>
</tr>
<tr>
<td><code>defaultBatchSize</code></td>
<td>The number of files to handle in a single transaction. This value can be overridden by a batch size argument provided in the <code>load()</code> and <code>remove()</code> methods. A batch size of -1 means to handle the entire job in one transaction. A batch size of 0 or 1 means to treat each file as a separate transaction. Default: -1</td>
</tr>
<tr>
<td><code>defaultTypeMapper</code></td>
<td>The default TypeMapper that the LoaderManager uses if no TypeMapper is provided in the LoaderManager methods <code>load()</code> or <code>processManifest()</code>, or by the FileSystemMonitorService. If a manifest file specifies its own TypeMappings, those have precedence. Type: <code>atg.repository.loader.TypeMapper</code> Default: <code>/atg/dynamo/service/loader/FileTypeMapper</code></td>
</tr>
<tr>
<td><code>jobEventListeners</code></td>
<td>An array of components that listen to <code>JobEvents</code>. Type: <code>atg.repository.loader.JobEventListener[]</code></td>
</tr>
<tr>
<td><code>jobIdGenerator</code></td>
<td>An <code>IdGenerator</code> component that creates job IDs. Type: <code>atg.service.idgen.IdGenerator</code> Default: <code>/atg/dynamo/service/FileIdGenerator</code></td>
</tr>
<tr>
<td><code>jobIdPrefix</code></td>
<td>A string to prepend to <code>jobIds</code>. Default: <code>RLJob</code></td>
</tr>
<tr>
<td><code>jobIdSpace</code></td>
<td>The name of the <code>IdSpace</code> used to generate <code>jobIds</code>. Default: <code>RLModuleJobIdSpace</code></td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jobQueueSize</code></td>
<td>The number of threads used in the jobs queue</td>
</tr>
<tr>
<td></td>
<td>Default: 1</td>
</tr>
<tr>
<td><code>loaderErrorEventListens</code></td>
<td>An array of components that listen to error Loader Events</td>
</tr>
<tr>
<td></td>
<td>Type: <code>atg.repository.loader.LoaderErrorEventListener[]</code></td>
</tr>
<tr>
<td></td>
<td>Default: null</td>
</tr>
<tr>
<td><code>loaderEventListens</code></td>
<td>An array of components that listen to add, update and remove Loader Events</td>
</tr>
<tr>
<td></td>
<td>Type: <code>atg.repository.loader.LoaderEventListener[]</code></td>
</tr>
<tr>
<td><code>suspendedJobsFifoSize</code></td>
<td>The number of suspended jobs that should be kept in the queue</td>
</tr>
<tr>
<td></td>
<td>Default: 10</td>
</tr>
<tr>
<td><code>suspendFailedJobs</code></td>
<td>A Boolean property, specifies whether to suspend or cancel failed jobs</td>
</tr>
<tr>
<td></td>
<td>Default: <code>true</code></td>
</tr>
</tbody>
</table>

#### Error policies

The LoaderManager uses a configurable error handling policy, defined by the Repository Loader's ErrorPolicy component. Each job processed by the LoaderManager might contain files that cause exceptions to be thrown. The LoaderManager consults the ErrorPolicy component to determine how to handle exceptions. All exceptions are logged. The Repository Loader sets a success state for each job and each batch. ErrorPolicy methods also determine how to proceed after encountering a problem while processing a job:

<table>
<thead>
<tr>
<th>ErrorPolicy method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>checkIsExceptionFatal</code></td>
<td><code>true</code>: The exception terminated a job.</td>
</tr>
<tr>
<td></td>
<td><code>false</code>: Invoked <code>checkRequiresNewTransaction</code> method.</td>
</tr>
<tr>
<td><code>checkRequiresNewTransaction</code></td>
<td><code>true</code>: The exception required a new transaction in order to continue.</td>
</tr>
<tr>
<td><code>checkEndTransactionWithBatchState</code></td>
<td><code>true</code>: Transaction demarcations should end using the value of the job's <code>batchFailed</code> property. If a batch fails, all subsequent batches in the job are rolled back, whether or not they contain errors.</td>
</tr>
</tbody>
</table>
TypeMapper and TypeMappings

Your file system might have different file types, where each type has different requirements for conversion into a repository item. The TypeMapper component determines which of the configured set of TypeMapping components is appropriate for a given file. Each TypeMapping specifies a content handler component. The file is routed to the appropriate ContentHandler for its type.

TypeMapper component

A TypeMapper component can be created from one of two classes:

- `atg.repository.loader.ExtFilterTypeMapper` specifies one or more file extensions in its `extensions` property. Source file extensions are thereby mapped to the appropriate TypeMappings.
- `atg.repository.loader.DirFilterTypeMapper` specifies one or more source file parent directories in its `directories` property. These directories determine the location of source file type mappings.

In both cases, the `extensions` and `directories` are array properties whose elements map to the corresponding elements in the `typeMappings` property.

In the following example, a TypeMapper configuration maps five file extensions to five TypeMappings. Thus, files with extension `.cfo` map to TypeMapping `CatalogFolderTypeMapping`, `.ctg` maps to `CatalogTypeMapping`, and so on:

```
$class=atg.repository.loader.ExtFilterTypeMapper
$scope=global

$extensions+=.cfo,.ctg,.cat,.prd,.sku
$typeMappings+=
   CatalogFolderTypeMapping,
   CatalogTypeMapping,
   CategoryTypeMapping,
   ProductTypeMapping,
   SkuTypeMapping
```

TypeMapper properties

A TypeMapper component is configured with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>folderTypeMapping</code></td>
<td>The Nucleus address of the TypeMapping that handles folder item descriptors. This property is required if any source files are content item types.</td>
</tr>
<tr>
<td><code>typeMappings</code></td>
<td>An array of Nucleus addresses of the TypeMappings used by this TypeMapper component.</td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extensions</td>
<td>An array of file extensions, specified in TypeMappers of class <code>atg.repository.loader.ExtFilterTypeMapper</code>.</td>
</tr>
<tr>
<td>directories</td>
<td>An array of directories, provided in TypeMappers of class <code>atg.repository.loader.ExtFilterTypeMapper</code>.</td>
</tr>
</tbody>
</table>

### TypeMapping properties

The Repository Loader includes a `TypeMappingImpl` implementation of the `TypeMapping` interface. You can configure an instance of `TypeMappingImpl` for a given item descriptor with the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contentIsXML</td>
<td>A boolean property, specifies whether files assigned to this mapping contain XML content.</td>
</tr>
<tr>
<td>parseContent</td>
<td>A boolean property, specifies whether to parse the content of files assigned to this mapping for property values.</td>
</tr>
<tr>
<td>itemDescriptorName</td>
<td>The name of the item descriptor handled by this TypeMapping.</td>
</tr>
<tr>
<td>contentHandler</td>
<td>The Nucleus address of the ContentHandler component that handles content for this mapping.</td>
</tr>
<tr>
<td>contentRootPathProvider</td>
<td>The Nucleus address of the ContentRootPathProvider component used by this mapping, if any.</td>
</tr>
<tr>
<td>encodingTyper</td>
<td>The Nucleus address of the PageEncodingTyper used by this mapping, if any.</td>
</tr>
<tr>
<td>pathPropertyName</td>
<td>If the item descriptor is not a content item descriptor, set this property to the name of a repository item property to hold the file path of the items.</td>
</tr>
<tr>
<td>repository</td>
<td>The Nucleus address of the MutableRepository that contains the item descriptor handled by this TypeMapping.</td>
</tr>
</tbody>
</table>
Property | Description
---|---
updatePropertyConfiguration | Specifies how this mapping uses ID and path properties to create, update, and remove items, one of the following:

- CONTENT_ITEM_DESCRIPTOR_ID_AND_PATH_PROP
- CONTENT_DEFINED_ID_AND_NO_PATH_PROP
- CONTENT_DEFINED_ID_AND_NAMED_PATH_PROP
- GENERATED_ID_AND_NO_PATH_PROP
- GENERATED_ID_AND_NAMED_PATH_PROP
- ID_EQUALS_FILE_PATH

For information about these settings, see Setting Repository IDs.

Setting Repository IDs

When the Repository Loader creates a repository item from a file, it must assign that item a repository item ID. A TypeMapping’s updatePropertyConfiguration property points to an enumeration (of class UpdatePropertyConfiguration) that describes how repository item IDs and path properties are used by the repository, and the TypeMapping used to set the repository item ID and locate repository items for update.

The enumeration can be set to one of the following values:

- CONTENT_ITEM_DESCRIPTOR_ID_AND_PATH_PROP
  
  For content item descriptor types only, use descriptor metadata to determine which properties should be used to assign the repository item ID and path properties. IDs can still be assigned from content data.

  If you use this value, make sure the source files do not include an ID tag, or set their TypeMapping’s parseContent property to false.

- CONTENT_DEFINED_ID_AND_NO_PATH_PROP
  
  The repository item ID property is set as part of the file parsing process. Because the file content uniquely and persistently defines the repository item’s ID, no path property needs to be assigned.

- CONTENT_DEFINED_ID_AND_NAMED_PATH_PROP
  
  The repository item ID property is set as part of the file parsing process. Set the repository item property specified by the TypeMapping’s pathPropertyName property with the file’s path.

- GENERATED_ID_AND_NO_PATH_PROP
  
  Use a value generated by the IdGenerator for the repository item ID, using the IdGenerator specified by the TypeMapping’s idGenerator property. If no idGenerator is specified, errors result. No path property is set. As a consequence, files assigned to this mapping cannot be updated or removed with the Repository Loader.

- GENERATED_ID_AND_NAMED_PATH_PROP
Use a value generated by the `IdGenerator` for the repository item ID, using the `IdGenerator` specified by the `TypeMapping`'s `idGenerator` property. If no `IdGenerator` is specified, errors result. Set the repository item property specified by the `TypeMapping`'s `pathPropertyName` property with the file's path.

`ID_EQUALS_FILE_PATH`

For non-content item descriptors. Use the file's path as both its repository item ID and its path value. If you use this value, make sure the source files do not include an ID tag, or set their `TypeMapping`'s `parseContent` property to `false`.

**ContentHandlers**

Files are routed to ContentHandler components according to the mapping of file types and item descriptors, as established by TypeMapping components. Each ContentHandler transforms files into repository items and invokes a back end system to perform add, update, and remove operations on the repository.

The ATG platform provides one ContentHandler class:

`atg.repository.loader.Xml2RepositoryContentHandler`

This class transforms source files into XML files that conform to the provided `xml2repository schemas`, then transforms the XML files into repository items.

**ContentHandler properties**

A ContentHandler component has the following configurable properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addService</td>
<td>The component that handles repository add operations.</td>
</tr>
<tr>
<td>removeService</td>
<td>The component that handles repository remove operations.</td>
</tr>
<tr>
<td>updateService</td>
<td>The component that handles repository update operations.</td>
</tr>
<tr>
<td>enableTransforms</td>
<td>Boolean, specifies whether an <code>xmlTransformer component</code> is enabled.</td>
</tr>
<tr>
<td>transformFilter</td>
<td>The <code>FileMappingFilter</code> that determines whether to run a file through the <code>XMLTransformer component</code>.</td>
</tr>
<tr>
<td>exceptionOnZeroRemove</td>
<td>Boolean, specifies whether to throw an exception if a remove operation removed no repository items.</td>
</tr>
</tbody>
</table>
| idPropertyNameForQueries| The name of the property to use for queries where the `TypeMapping`'s `updatePropertyConfiguration` property is set to one of the following:  
`CONTENT_DEFINED_ID_AND_NO_PATH_PROP`  
`CONTENT_DEFINED_ID_AND_NAMED_PATH_PROP` |
In order to add repository items, set the property `updateService.addWhenNoMatchedItems` to `true`; this enables execution of the component specified by the ContentHandler's `addService` property.

**XMLTransformer component**

The XMLTransformer component has a `stylesheets` property, which is set to one or more style sheet files; these enable transformation of XML files at load time into the required format. Depending on the number and complexity of the stylesheets, this can be a resource-intensive operation.

An XMLTransformer instance is found in Nucleus at:

```
/atg/dynamo/service/loader/typemapping/SampleXMLTransformer
```

### Repository Loader Administration

The Repository Loader includes a Web application comprised of administration JSPs and form handlers, where you create, delete, and monitor Repository Loader jobs. To use the Repository Loader Administration, include the `RL` module among the ATG modules that are specified when assembling the application EAR file.

You access the Repository Loader Administration from this URL:

```
http://hostname:port/rl
```

For example, if you use JBoss and the default JBoss port and your browser runs on the same machine as your Web server, use the following URL:

```
http://localhost:8080/rl
```

For details about using the Repository Loader Administration, see the Repository Loader Example section.

### RLClient

RLClient is an RMI client you can use to submit manifests that identify the files to load into the repository. Manifests are processed by the RemoteManifestProcess component `/atg/dynamo/repository/loader/RemoteManifestProcessorService`. You invoke the RLClient with this script:

```
<ATG10dir>/RL/bin/RLClient.{bat|sh}
```

**Note:** Before you run the script, set the `DYNAMO_HOME` variable.

See Repository Loader Manifest for information about the manifest file format.
### Command-line arguments

The following table describes the command-line arguments supplied to the `RLClient` script.

<table>
<thead>
<tr>
<th>Required arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-m manifestFilePath</code></td>
<td>The <code>-m</code> and <code>-mp</code> switches specify the server-side path to the manifest that contains the desired load commands.</td>
</tr>
<tr>
<td><code>-mp manifestFilePath</code></td>
<td>Use the <code>-mp</code> switch if the RLClient and host run on different operating systems.</td>
</tr>
<tr>
<td><code>-h hostname</code></td>
<td>Name of the host where the RemoteManifestProcessor runs.</td>
</tr>
<tr>
<td><code>-auth username:passwd</code></td>
<td>Supplies the username and password. By default, these are tested against the <code>/atg/dynamo/security/AdminUserAuthority</code>. To specify a different user authority, set the <code>userAuthority</code> property in <code>/atg/dynamo/repository/loader/RemoteManifestProcessorService</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-p propertiesFilePath</code></td>
<td>Path to a properties file that contains additional LoaderManager parameters. See Supplemental RLClient Parameters.</td>
</tr>
<tr>
<td><code>-r RMI Port</code></td>
<td>RMI port of the host where the RemoteManifestProcessor runs. If omitted, the default port is 8860.</td>
</tr>
<tr>
<td><code>-s servicename</code></td>
<td>Nucleus address of the RemoteManifestProcessorService component. If omitted, the default value is: <code>/atg/dynamo/repository/loader/RemoteManifestProcessorService</code>.</td>
</tr>
</tbody>
</table>

### Supplemental RLClient Parameters

You can supply RLClient the address of a properties file via its `-p` switch, which provides more LoaderManager parameters. This file can set the following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>batchSize</code></td>
<td>The number of files to process in each transaction. If omitted, RLClient uses the value of the LoaderManager's <code>defaultBatchSize</code> property. This property's default value is -1, which specifies to handle the entire import in one transaction.</td>
</tr>
<tr>
<td><code>numElementsInManifest</code></td>
<td>The total number of files in the manifest</td>
</tr>
</tbody>
</table>
For example:

```
atg.repository.loader.batchSize=2
atg.repository.loader.numElementsInManifest=725
```

**Repository Loader Manifest**

If the content root in the file system for your repository contains a large number of files, the FileSystemMonitorService might require a long time to identify which files and folders to load into the repository, and for the LoaderManager and ContentHandlers to convert the files and folders into repository items. You can reduce processing overhead by creating a Repository Loader manifest file that identifies in advance the files and folders to load.

For example, the following Repository Loader manifest file adds five files:

```
<manifest>
  <add>/main/Dynamo/RL/sample-data/user001.xml</add>
  <add>/main/Dynamo/RL/sample-data/user002.xml</add>
  <add>/main/Dynamo/RL/sample-data/user003.xml</add>
  <add>/main/Dynamo/RL/sample-data/user004.xml</add>
  <add>/main/Dynamo/RL/sample-data/user005.xml</add>
</manifest>
```

Elements in a Repository Loader manifest file are handled in order of appearance, so one element should not depend on a later element. For example, a content repository requires a folder hierarchy; so a content item should not precede the folder that contains it.

**Document Type Definition**

The Repository Loader manifest file is an XML file that conforms to the following DTD:

```
<!DOCTYPE manifest [
<!ELEMENT manifest (add | update | remove)>
<!ATTLIST manifest num-elements CDATA #IMPLIED>
<!ELEMENT add (#PCDATA)>
<!ATTLIST add type-mapping CDATA #IMPLIED>
<!ELEMENT update (#PCDATA)>
<!ATTLIST update type-mapping CDATA #IMPLIED>
]>
```
Manifest File Tags and Attributes

The Repository Loader manifest file uses the following XML elements:

<table>
<thead>
<tr>
<th>Tag/Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;manifest&gt;</code></td>
<td>Wraps the entire manifest.</td>
</tr>
<tr>
<td><code>num-elements</code></td>
<td>An attribute of the <code>&lt;manifest&gt;</code> tag, optionally indicates the total number of <code>add</code>, <code>remove</code>, and <code>update</code> elements in the manifest file.</td>
</tr>
<tr>
<td><code>&lt;add&gt;</code></td>
<td>Contains the path name of the source file or folder. For example: <code>&lt;add&gt;/home/Dynamo/RL/sample-data/user001.xml&lt;/add&gt;</code></td>
</tr>
<tr>
<td><code>&lt;remove&gt;</code></td>
<td><code>&lt;remove&gt;/home/Dynamo/RL/sample-data/user001.xml&lt;/remove&gt;</code></td>
</tr>
<tr>
<td><code>&lt;update&gt;</code></td>
<td><code>&lt;update&gt;/home/Dynamo/RL/sample-data/user002.xml&lt;/update&gt;</code></td>
</tr>
<tr>
<td><code>type-mapping</code></td>
<td>An attribute of an <code>add</code>, <code>remove</code>, or <code>update</code> tag, optionally specifies the TypeMapping for processing this file. The attribute value must be the absolute Nucleus path of a component that implements <code>atg.repository.loader.TypeMapping</code>.</td>
</tr>
</tbody>
</table>

**Note:** if no type-mapping is provided, the Repository Loader uses the TypeMapper that is specified by the `LoaderManager` configuration.

Importing Versioned Repository Data

You can use the Repository Loader to import data into ATG Content Administration versioned repositories. The loader can perform these tasks:

- Import file asset metadata into the PublishingFileRepository and write the file contents to the file system. It can also import file asset data into any custom versioned repositories that store content repository assets.
- Import repository assets into a versioned repository.

The Repository Loader module is included automatically when you use the `Publishing.base` module or any module that requires `Publishing.base`.

This section contains information that is specific to configuring and using the Repository Loader to import assets into Content Administration repositories. It includes these topics:

- Configuring the VersionedLoaderEventListener
Configuring the VersionedLoaderEventListener

In order to import repository assets into versioned repositories, you must configure a VersionedLoaderEventListener, whose tasks include creating and checking in the workspace that is used during the import. You create a VersionedLoaderEventListener component from this class:

```java
atg.epub.loader.VersionedLoaderEventListener
```

**VersionedLoaderEventListener properties**

The following table describes VersionedLoaderEventListener properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appendTimeToProcessName</td>
<td>Set to <code>true</code> in order to append the import time to the name of the project created and used for the import. If set to <code>true</code>, the Repository Loader creates a project for importing file assets, under the name specified by properties <code>processNamePrefix</code> and <code>timeFormat</code>. Default: <code>true</code></td>
</tr>
<tr>
<td>checkinOnCompletion</td>
<td>Set to <code>true</code> if the workspace (and its assets) should be checked in. Otherwise, <code>false</code>. If the <code>workspaceName</code> property is set to the name of an existing workspace—typically, an active project’s workspace—the Repository Loader blocks the check-in no matter how this property is set. If a project’s workspace is checked in while the project itself remains active, various problems result. For example, users cannot complete or delete the project, or view imported assets in the project. Default: <code>true</code></td>
</tr>
<tr>
<td>createProjects</td>
<td>Set to <code>true</code> in order to create a project and import assets into it. If set to <code>true</code>, the Repository Loader creates a project for importing file assets, under the name specified by properties <code>processNamePrefix</code> and <code>timeFormat</code>. Default: <code>false</code></td>
</tr>
<tr>
<td>disallowWorkspaceImportAfterTargetCreation</td>
<td>Specifies whether you can import an asset into an existing workspace specified by <code>workspaceName</code> after deployment target sites are initialized. If target sites are already initialized, this property must be set to <code>false</code> in order to allow import operations to succeed. Default: <code>true</code></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>password</strong></td>
<td>Password of the user to authenticate. Default: <code>admin</code></td>
</tr>
<tr>
<td><strong>personaPrefix</strong></td>
<td>The substring in an ACL that is used to identify the user in the UserAuthority, set as follows:</td>
</tr>
<tr>
<td></td>
<td>- Admin$user$: The user who performs the import is using an ACC account.</td>
</tr>
<tr>
<td></td>
<td>- Profile$login$: The user who performs the import is using an ATG Business Control Center account.</td>
</tr>
<tr>
<td></td>
<td>Default: <code>Admin$user$</code></td>
</tr>
<tr>
<td><strong>processDescription</strong></td>
<td>An arbitrary string. Default: <code>Imported by the RepositoryLoader</code></td>
</tr>
<tr>
<td><strong>processNamePrefix</strong></td>
<td>Together with timeFormat, specifies the name of the project used if createProjects is set to true.</td>
</tr>
<tr>
<td></td>
<td>Default: <code>Content Administration Import</code></td>
</tr>
<tr>
<td><strong>rootPath</strong></td>
<td>The fully qualified path of the parent folder of the top-level folder in the manifest to import, or the folder system to scan. All folders and files to import must be in or under this root folder.</td>
</tr>
<tr>
<td></td>
<td>For example, you might set rootPath to <code>/users/joe/import</code> and import the following files via an automatic import:</td>
</tr>
<tr>
<td></td>
<td><code>/users/joe/import/file1</code></td>
</tr>
<tr>
<td></td>
<td><code>/users/joe/import/dir/file2</code></td>
</tr>
<tr>
<td></td>
<td><code>/users/joe/import/dir/dir2/file3</code></td>
</tr>
<tr>
<td></td>
<td>The directories and files imported into the PublishingFileRepository are as follows (specified from the repository’s root):</td>
</tr>
<tr>
<td></td>
<td><code>/file1</code></td>
</tr>
<tr>
<td></td>
<td><code>/dir/file2</code></td>
</tr>
<tr>
<td></td>
<td><code>/dir/dir2/file3</code></td>
</tr>
<tr>
<td><strong>Note 1:</strong></td>
<td>When performing imports on Windows, use double backslashes (\ \ ) as path separators. For example:</td>
</tr>
<tr>
<td></td>
<td>C:\ATG\ATG10.0.1\MyProductionModule\config</td>
</tr>
<tr>
<td><strong>Note 2:</strong></td>
<td>When importing the initial set of ATG assets from your production module, such as scenarios and slots, the root path is the production module’s config directory, which contains the ATG assets.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| timeFormat    | The format to use when appending the import time to the name of project used for the import. The default format is:  

`MMMMM dd, yyyy hh:mm:ss aaa`

For information on changing the default value, see the API reference for class `java.text.SimpleDateFormat`.                                                                                                      |
|               |                                                                                                                                                                                                                                                                                                                                                     |
| userAuthority | The `userAuthority` that resolves the user specified in the `username` property.                                                                                                                                                                                                        |
|               | Default: `/atg/dynamo/security/UserAuthority`                                                                                                                                                                                                                                         |
| userName      | The username of the user to authenticate.                                                                                                                                                                                                                                             |
|               | Default: `admin`                                                                                                                                                                                                                                                                       |
| workflowName  | Default: `/Common/commonWorkflow.wdl`                                                                                                                                                                                                                                               |
| workspaceName | The name of the workspace to use for the import. If the corresponding workspace does not exist, it is created by the system.  

All workspace names must be unique. The import fails if the name corresponds to a completed project’s workspace (because it is already checked in).  

If `createProjects` is set to `false` (the default), the system creates a workspace for the import. If the `workspaceName` property is not set (the default), the system creates a workspace name from the `IdGenerator`; otherwise, it uses the name specified in this property.  

If `createProjects` is set to `true`, the system ignores this property and creates a project for the import.                                                                                                                                                                                                 |

### Project properties

After you initialize a target site, the Repository Loader must import assets into a project. In order to do so, set the `VersionedLoaderEventListener`'s `createProjects` property to true. When you run the Repository Loader on these assets, it performs these tasks:

- Creates a project, concatenating the project name from the values specified in the `VersionedLoaderEventListener` properties `processNamePrefix` and `timeFormat`.
- Imports the assets into that project.

### User access configuration

The properties `userAuthority`, `personaPrefix`, `userName`, and `password` collectively verify user access to the secured versioned repository where file assets are imported—for example, `/atg/epub/file/SecuredPublishingFileRepository`.
Default configuration

By default, the VersionedLoaderEventListener is configured as follows:

```java
$className=atg.epub.loader.VersionedLoaderEventListener

versionManager=/atg/epub/version/VersionManagerService

rootPath={appModuleResource?moduleId=home&resourceURI=}

createProjects=false

projectNamePrefix=Content Administration Import

appendTimeToProjectName=true

projectDescription=Imported by the RepositoryLoader

timeFormat=MMMMM dd, yyyy hh:mm:ss a

# The activity to associate with a project created by the loader
# activityId=merchandising.manageCommerceAssets

workflowName=/Common/commonWorkflow.wdl

userAuthority=/atg/dynamo/security/UserAuthority

personaPrefix=Admin$user$

userName=admin

password=admin

checkinOnCompletion=true

checkinComment=

idGenerator=/atg/dynamo/service/IdGenerator

workspaceNameIdSpace=repositoryLoaderWkspName

#workspaceName=RepLoader-1
```

Importing Targeters that Reference rules Files

Manually created targeters can store their rule sets in separate .rules files. If so, you must edit each applicable RuleSetService configuration file to specify the virtual file system that stores the .rules files, ConfigFileSystem. This also ensures deployment of the .rules file together with the RuleSetService to the corresponding ConfigFileSystem on the production site.

**Note:** This step is optional if the rule set is stored in the RuleSetService's ruleSet property; this is always true for targeters that you create in the ATG Control Center.

For each targeter that stores its rule set in a separate .rules file, modify the RuleSetService configuration file as follows:

```
rulesFileSystem=/atg/epub/file/ConfigFileSystem
```

For example:
$class\text{=}atg\text{.}\ targeting\text{.}\ RuleSetService$

rulesFileSystem\text{=}/atg/epub/file/ConfigFileSystem
rulesFilePath=targeting/rulesets/NewEnglandSnowboarders.rules
updatesEnabled=true
rulesFileCheckSeconds=0

The property rulesFileSystem specifies the VFS that stores the .rules file, providing the system with a reference to the file via the VFS. This setting ensures the file is exposed properly in the content development environment and is deployed to the correct asset destination, the ConfigFileSystem.

### Configuring TypeMapping Components for the PublishingFileRepository

ATG Content Administration provides a set of TypeMapping components for the default content item descriptors in the PublishingFileRepository, and a PublishingTypeMapper component that defines this array of TypeMapping components. These components are located in:

<ATG10dir>/Publishing/base/config/atg/epub/file/typemappers

You can configure additional TypeMapping components by extending the PublishingFileRepository to support additional content item descriptors (see the section Configure Support for Other File Assets in the ATG Content Administration Programming Guide).

### Repository Loader Example

The Repository Loader module includes a simple example of an SQL repository that uses the Repository Loader. The Repository Loader example is in the <ATG10dir>/RL/Example directory. It loads the files in the FileSystemMonitorService’s root path:

<ATG10dir>/RL/Example/j2ee-apps/example/web-app/public

The SQL repository in this example is a GSARespository component with this Nucleus address:

/atg/rl-example/ExampleRepository

Repository item types in ExampleRepository are specified by this repository definition file:

<ATG10dir>/RL/Example/config/atg/rl-example/exampleRepository.xml.

This XML file defines the following item types in its item descriptors (note how content item types use item descriptor inheritance):
## Item descriptor name | Description
--- | ---
**fileFolder** | A content folder item type.  
**fileAsset** | A content item type. This is the super-type for a series of item types that inherit from the `fileAsset` item type. The type property specifies which subtype (`textFileAsset`, `binaryFileAsset`, `htmlArticle`, `txtPressRelease`, `xmlPressRelease`, `gifImage`, or `jpgImage`) an item belongs to. A `fileAsset` item also defines `lastModified`, `size`, and `parentFolder` properties.
**textFileAsset** | A content item type that inherits from `fileAsset`. It is designed for text files. The text content is stored in the `content` big string property. It has subtypes named `htmlArticle`, `txtPressRelease`, and `xmlPressRelease`.
**binaryFileAsset** | A content item type that inherits from `fileAsset`. The content is stored in the `content` binary property. It has subtypes named `gifImage` and `jpgImage`.
**htmlArticle** | A content item type that inherits from `fileAsset` and from `textFileAsset`. It defines a `published` timestamp property and a `keywords` string property.
**txtPressRelease** | A content item type that inherits from `fileAsset` and from `textFileAsset`.
**xmlPressRelease** | A content item type that inherits from `fileAsset` and from `textFileAsset`.
**gifImage** | A content item type that inherits from `fileAsset` and from `binaryFileAsset`.
**jpgImage** | A content item type that inherits from `fileAsset` and from `binaryFileAsset`.
**address** | A simple non-content item type. Used by the user item type's `addresses` property.
**contact** | A non-content item type. Used by the user item type's `contacts` property.
**phone** | A simple non-content item type. Used by the user item type's `numbers` property.
**user** | A complex non-content item type. The user item type is described in detail in the [User Item Type](#) section.
User Item Type

The user item type demonstrates a variety of data relationships. It shows how an item type can use properties that nest other item types. The user item descriptor is defined as follows:

```xml
<item-descriptor name="user" default="true">
    <table name="rlex_user" type="primary" id-column-name="id">
        <property name="id" data-type="string"/>
        <property name="name" column-name="nam_col" data-type="string"/>
        <property name="age" column-name="age_col" data-type="string"/>
    </table>

    <!-- a set of address items -->
    <table name="rlex_address" type="multi" id-column-name="user_id">
        <property name="addresses" column-name="addr_id" data-type="set" component-item-type="address" cascade="delete,update"/>
    </table>

    <!-- a set of contact items -->
    <table name="rlex_contact" type="multi" id-column-name="user_id">
        <property name="contacts" column-name="con_id" data-type="set" component-item-type="contact" cascade="delete,update"/>
    </table>

    <!-- a map of phone items -->
    <table name="rlex_phone" type="multi" id-column-name="user_id" multi-column-name="kind">
        <property name="numbers" column-name="phone_id" data-type="map" component-item-type="phone" cascade="delete,update"/>
    </table>

    <!-- a one-to-one mapping in an aux table -->
    <table name="rlex_job" type="auxiliary" id-column-names="id">
        <property name="jobtype"/>
        <property name="title"/>
    </table>

    <!-- a multivalue property (array) -->
    <table name="rlex_subjects" type="multi" id-column-names="id" multi-column-name="seq_num">
        <property name="favoriteSubjects" column-names="subject" data-type="array" component-data-type="string"/>
    </table>

    <!-- a multivalue property (list) -->
    <table name="rlex_worst" type="multi" id-column-names="id" multi-column-name="seq_num">
        <property name="worstSubjects" column-names="subject" data-type="list" component-data-type="string"/>
    </table>
</item-descriptor>
```
**Item properties**

The *user* item type defines in its primary table three string properties:

- `id`
- `name`
- `age`

User contact information is defined in three multi-valued properties:

- `addresses`
- `contacts`
- `numbers`

The values of each of these multi-valued properties are other repository items—*address*, *contact*, and *phone*, respectively.

The item type also defines these properties:

- `jobType` and `title` are string properties that use a one-to-one mapping in an auxiliary table.
- `favoriteSubjects`, `worstSubjects`, and `card_num` are multi-valued properties that use a one-to-many mapping in a multi table.

**Item Pathnames**

The Repository Loader example repository uses a *parentFolder* property in each content item type, along with the item’s file name, in order to determine the item’s path. Each TypeMapping component in the example uses this property setting:

```
updatePropertyConfiguration=CONTENT_ITEM_DESCRIPTOR_ID_AND_PATH_PROP
```
Type Mappings and Content Handlers

The Repository Loader example is configured with TypeMapping components for each of its item types. The TypeMapping determines which ContentHandler component processes items. The following table shows which TypeMapping and ContentHandler components are defined in the Nucleus directory `atg/rl-example/ExampleRepository`:

<table>
<thead>
<tr>
<th>Item type</th>
<th>TypeMapping component</th>
<th>ContentHandler component</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileFolder</td>
<td>FolderTypeMapping</td>
<td>ContentHandler</td>
</tr>
<tr>
<td>textFileAsset</td>
<td>UnparsedContentTypeMapping</td>
<td>ContentHandler</td>
</tr>
<tr>
<td>htmlArticle</td>
<td>HtmlArticleTypeMapping</td>
<td>HtmlArticleContentHandler</td>
</tr>
<tr>
<td>txtPressRelease</td>
<td>PressReleaseTXTTypeMapping</td>
<td>ContentHandler</td>
</tr>
<tr>
<td>xmlPressRelease</td>
<td>PressReleaseXMLTypeMapping</td>
<td>ContentHandler</td>
</tr>
<tr>
<td>gifImage</td>
<td>GifImageTypeMapping</td>
<td>ContentHandler</td>
</tr>
<tr>
<td>jpgImage</td>
<td>JpgImageTypeMapping</td>
<td>ContentHandler</td>
</tr>
<tr>
<td>user</td>
<td>UserTypeMapping</td>
<td>UserContentHandler</td>
</tr>
</tbody>
</table>

**Note:** All type mappings that do not require XSL preprocessing use the ContentHandler component.

TypeMapper

The Repository Loader example is configured with a TypeMapper component that maps file extensions to TypeMappings, with the following property configuration:

```properties
extensions=.xml,.txt,.gif,.jpg,.html,.eml

typeMappings=pressReleaseXMLTypeMapping,
pressReleaseTXTTypeMapping,
gifImageTypeMapping,
jpgImageTypeMapping,
htmlArticleTypeMapping,
userTypeMapping
```

The yields the following mappings:

<table>
<thead>
<tr>
<th>This file extension:</th>
<th>Maps to this TypeMapping component:</th>
</tr>
</thead>
<tbody>
<tr>
<td>.xml</td>
<td>pressReleaseXMLTypeMapping</td>
</tr>
</tbody>
</table>
This file extension: | Maps to this TypeMapping component:
---|---
.txt | pressReleaseTXTTypeMapping
.gif | gifImageTypeMapping
.jpg | jpgImageTypeMapping
.htm | htmlArticleTypeMapping
.eml | userTypeMapping

xml2repository Schemas

The Repository Loader uses XML schemas to represent repository items as XML files. You can load the repository using XML files that conform to the schema, or export existing repository items in the form of XML files that can be loaded later.

The Repository Loader example includes an XML schema for each of the repository item types listed below, in the following directory:

<ATG10dir>/RL/Example/repository2xml/schemas

<table>
<thead>
<tr>
<th>Item type</th>
<th>Schema file</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileFolder</td>
<td>RL-ExampleRepository+fileFolder.xsd</td>
</tr>
<tr>
<td>htmlArticle</td>
<td>RL-ExampleRepository+htmlArticle.xsd</td>
</tr>
<tr>
<td>txtPressRelease</td>
<td>RL-ExampleRepository+txtPressRelease.xsd</td>
</tr>
<tr>
<td>xmlPressRelease</td>
<td>RL-ExampleRepository+xmlPressRelease.xsd</td>
</tr>
<tr>
<td>gifImage</td>
<td>RL-ExampleRepository+gifImage.xsd</td>
</tr>
<tr>
<td>jpgImage</td>
<td>RL-ExampleRepository+jpgImage.xsd</td>
</tr>
</tbody>
</table>

Running the Repository Loader Example

You can use the Repository Loader example to perform two tasks:

- Convert files to repository items
- Export repository items to XML

Convert files to repository items

In order to run the Repository Loader example, follow these steps:
1. During application assembly, specify the following module so you can use the Repository Loader example:

```
RL.Example
```

**Note:** To avoid conflicts among database connection configurations, exclude from the application any other ATG demos, examples, or reference application modules that include the Repository Loader.

2. If you are running on Windows, set the `pathSeparator` property of the `/atg/rl-example/ExampleRepository` component to backslash (`\`).

**Note:** if you set this property directly in a properties file rather than in the ACC, use this format:

```
pathSeparator=\\n
```

3. After the application starts, navigate to the Repository Loader Administration, on this page:

```
http://hostname:port/rl
```

   - `hostname`: the application server machine
   - `port`: the port your application server uses to listen for HTTP requests.

For example:

```
http://skua:8180/rl
```


5. In the Create Job page:
   - Set Recurse to Yes
   - Click Add Files

This loads the files from the root path into the ExampleRepository as repository items. The Repository Loader Administration should show the job as completed.

**Export repository items to XML**

You can use the Repository Loader example to evaluate output by exporting a repository item to XML. Navigate to this page:

```
http://hostname:port/rl-example/itemAsXml.jsp
```

This page takes a hard-coded repository item (user001) and outputs it as an XML file.

Examining the XML format generated for a given item descriptor can help you generate compliant XML, or write an XSL stylesheet for the Repository Loader to apply before processing a file’s contents.
15 Repository Web Services

The ATG platform provides infrastructure and tools for accessing ATG repositories with Web services. The Creating Repository Web Services chapter in the ATG Web Services and Integration Framework Guide describes how to create a Web service that exposes a particular repository item descriptor, or an individual repository item property. Repository items can be passed via a Web service in the form of an XML file created with the ATG xml2repository feature, described in the Repository to XML Data Binding chapter of the ATG Web Services and Integration Framework Guide.

This chapter describes three generalized Web services you can use to provide access to ATG repositories:

- GetRepositoryItem Web Service
- PerformRQLQuery Web Service
- PerformRQLCountQuery Web Service

These Web services are packaged in the `<ATG10dir>/DAS/WebServices` directory. For information about how to deploy these and other Web services, see the ATG Web Services and Integration Framework Guide.

Note that the descriptions of these Web services include URLs that begin `http://hostname:port` where `hostname` machine running your application server and `port` is the port your application server uses to listen for HTTP requests. To find the default port, see the ATG Installation and Configuration Guide.

Also to these repository Web services, ATG Personalization and ATG Commerce contain several Web services that provide access to specific ATG repositories and repository item types. A complete list of the Web services included in the ATG platform can be found in the ATG Web Services and Integration Framework Guide.

GetRepositoryItem Web Service

The GetRepositoryItem Web service retrieves an item with the given repository ID and item type from the given repository. The item is returned in XML format. The Web service method calls directly through to the `getRepositoryItem` method of the `atg.repository.RepositoryServices` class, which handles all logic, error checking, and result transformation.
**Web Service Implementation**

<table>
<thead>
<tr>
<th>Web Service URL</th>
<th><code>http://hostname:port/repository/generic/getRepositoryItem/getRepositoryItem</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Service Class Name</td>
<td><code>webservice.GetRepositoryItemsImpl</code></td>
</tr>
<tr>
<td>Nucleus Component</td>
<td><code>/atg/repository/RepositoryServices</code></td>
</tr>
<tr>
<td>Method Name</td>
<td><code>getRepositoryItem</code></td>
</tr>
<tr>
<td>Input Parameters</td>
<td>String <code>pRepositoryPath</code></td>
</tr>
<tr>
<td></td>
<td>the path of the repository component from which to retrieve the item</td>
</tr>
<tr>
<td></td>
<td>String <code>pItemDescriptorName</code></td>
</tr>
<tr>
<td></td>
<td>the item type of the item to retrieve</td>
</tr>
<tr>
<td></td>
<td>String <code>pRepositoryId</code></td>
</tr>
<tr>
<td></td>
<td>the repository ID of the repository item to retrieve</td>
</tr>
<tr>
<td>Output</td>
<td>String <code>the foundRepositoryItem</code> in XML format, or null if no item with that repository ID exists</td>
</tr>
<tr>
<td>Exceptions</td>
<td><code>atg.repository.RepositoryException</code> if a repository error occurs</td>
</tr>
<tr>
<td></td>
<td><code>atg.repository.xml.GetException</code> if an error occurs translating the item into XML</td>
</tr>
</tbody>
</table>

**GetRepositoryItem Exceptions**

Exceptions generated by the `GetRepositoryItem` Web service may occur for the following reasons:

**RepositoryException**

- The `pRepositoryPath` specified by the argument is null, or empty
- The `pRepositoryPath` specified by the argument does not resolve to a component
- The `pRepositoryPath` specified by the argument does not resolve to a Repository
- The call to `Repository.getItem` throws a `RepositoryException`

**GetException**

- The call to `GetService getItemAsXML` throws a `GetException`

**Web Service Generation**

The following parameters are used to generate this Web service, using the `WebServiceGeneratorImpl` class:
### Parameter Table

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NucleusPath</td>
<td>/atg/repository/RepositoryServices</td>
</tr>
<tr>
<td>MethodName</td>
<td>getRepositoryItem</td>
</tr>
<tr>
<td>EarFileName</td>
<td>repositoryWebServices.ear</td>
</tr>
<tr>
<td>AppName</td>
<td>RepositoryWebServices</td>
</tr>
<tr>
<td>AppDescription</td>
<td>A collection of Web services used to make repository calls</td>
</tr>
<tr>
<td>ServletName</td>
<td>getRepositoryItem</td>
</tr>
<tr>
<td>ServletDisplayName</td>
<td>GetRepositoryItem</td>
</tr>
<tr>
<td>ServletDescription</td>
<td>When called, gets a repository using the given arguments</td>
</tr>
<tr>
<td>URLPattern</td>
<td>GetRepositoryItem</td>
</tr>
<tr>
<td>WebURI</td>
<td>generic.war</td>
</tr>
<tr>
<td>ContextPath</td>
<td>repository/generic</td>
</tr>
<tr>
<td>WebAppName</td>
<td>RepositoryServices</td>
</tr>
<tr>
<td>WebAppDescription</td>
<td>A collection of generic repository Web services, where the user must</td>
</tr>
<tr>
<td></td>
<td>provide information about which repository is being acted upon</td>
</tr>
<tr>
<td>Host</td>
<td>null (will be dynamic)</td>
</tr>
<tr>
<td>Port</td>
<td>null (will be dynamic)</td>
</tr>
<tr>
<td>ParameterNames</td>
<td>pRepositoryPath, pItemDescriptorName, pRepositoryId</td>
</tr>
<tr>
<td>UseSession</td>
<td>true</td>
</tr>
</tbody>
</table>

### Web Service Security

By default, the GetRepositoryItem Web service uses a security policy that allows access only by Administrators. You are free to change this to suit your needs, depending on how you expect to use this service.

1. **FunctionalName:** GenericRepositoryUser
2. **SecurityPolicy:** StandardSecurityPolicy
3. **ACL:** Administrators

### PerformRQLQuery Web Service

The PerformRQLQuery Web service executes a given RQL string on the repository specified by the pRepositoryPath argument, and returns items of the type specified by the pItemDescriptorName.
The given RQL string cannot contain parameters, as RQL expects parameters in the form of an array of Objects. The Web service calls through to the `performRQLQuery` method of the `atg.repository.RepositoryServices` class, which handles all logic, error checking, and result transformation.

**Web Service Implementation**

<table>
<thead>
<tr>
<th>Web Service URL</th>
<th><a href="http://hostname:port/repository/generic/performRQLQuery/performRQLQuery">http://hostname:port/repository/generic/performRQLQuery/performRQLQuery</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Service Class Name</td>
<td>webservice.PerformRQLQuerySEIImpl</td>
</tr>
<tr>
<td>Nucleus Component</td>
<td>/atg/repository/RepositoryServices</td>
</tr>
<tr>
<td>Method Name</td>
<td>performRqlQuery</td>
</tr>
</tbody>
</table>
| Input Parameters         | String pRepositoryPath  
The path of the repository component to query against. |
|                         | String pItemDescriptorName  
The item type of the repository items to query against. |
|                         | String pRQLString  
The RQL string to execute. Note that this string cannot contain parameters. |
| Output                   | String[]  
The found RepositoryItems in XML format, or null if no items satisfy the given query. |
| Exceptions               | atg.repository.RepositoryException if a repository error occurs |
|                         | atg.repository.xml.GetException if an error occurs translating the item into XML |

**PerformRQLQuery Exceptions**

Exceptions generated by the `PerformRQLQuery` Web service may occur for the following reasons:

**RepositoryException**

- The `pRepositoryPath` specified by the argument is null, or empty.
- The `pRepositoryPath` specified by the argument does not resolve to a component.
- The `pRepositoryPath` specified by the argument does not resolve to a Repository.
- The `pItemDescriptorName` specified by the argument does not identify an ItemDescriptor of the given repository.
- The `pRQLString` specified by the argument is null, or empty.
- The RQL code throws an exception during parsing or execution.
GetException

The call to `GetService.getItemAsXML` throws a `GetException` for any found items.

**Web Service Generation**

The following parameters are used to generate this Web service, using the `WebServiceGeneratorImpl` class:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NucleusPath</strong></td>
<td>/atg/repository/RepositoryServices</td>
</tr>
<tr>
<td><strong>MethodName</strong></td>
<td>performRQLQuery</td>
</tr>
<tr>
<td><strong>EarFileName</strong></td>
<td>repositoryWebServices.ear</td>
</tr>
<tr>
<td><strong>AppName</strong></td>
<td>RepositoryWebServices</td>
</tr>
<tr>
<td><strong>AppDescription</strong></td>
<td>A collection of Web services used to make repository calls</td>
</tr>
<tr>
<td><strong>ServletName</strong></td>
<td>performRQLQuery</td>
</tr>
<tr>
<td><strong>ServletDisplayName</strong></td>
<td>PerformRQLQuery</td>
</tr>
<tr>
<td><strong>ServletDescription</strong></td>
<td>When called, executes the given query against the specified repository</td>
</tr>
<tr>
<td><strong>URLPattern</strong></td>
<td>PerformRQLQuery</td>
</tr>
<tr>
<td><strong>WebURI</strong></td>
<td>generic.war</td>
</tr>
<tr>
<td><strong>ContextPath</strong></td>
<td>repository/generic</td>
</tr>
<tr>
<td><strong>WebAppDisplayName</strong></td>
<td>RepositoryServices</td>
</tr>
<tr>
<td><strong>WebAppDescription</strong></td>
<td>A collection of generic repository Web services, where the user must provide information about which repository is being acted upon</td>
</tr>
<tr>
<td><strong>Host</strong></td>
<td>null (will be dynamic)</td>
</tr>
<tr>
<td><strong>Port</strong></td>
<td>null (will be dynamic)</td>
</tr>
<tr>
<td><strong>ParameterNames</strong></td>
<td>pRepositoryPath, pItemDescriptorName, pRQLString</td>
</tr>
<tr>
<td><strong>UseSession</strong></td>
<td>true</td>
</tr>
</tbody>
</table>

**PerformRQLCountQuery Web Service**

The `PerformRQLCountQuery` Web service executes a given RQL string on the repository specified by the `pRepositoryPath` argument, and returns the number of items that satisfy that query. The given RQL
string cannot contain parameters, as RQL expects parameters in the form of an array of Objects and a Web service request cannot pass an array of Objects. The Web service calls through to the `performRQLCountQuery` method of the `atg.repository.RepositoryServices` class, which handles all logic and error checking.

### Web Service Implementation

<table>
<thead>
<tr>
<th>Web Service URL</th>
<th><a href="http://hostname:port/repository/generic/performRQLCountQuery/performRQLCountQuery">http://hostname:port/repository/generic/performRQLCountQuery/performRQLCountQuery</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Service Class Name</td>
<td><code>webservice.PerformRQLCountQuerySEIImpl</code></td>
</tr>
<tr>
<td>Nucleus Component</td>
<td><code>/atg/repository/RepositoryServices</code></td>
</tr>
<tr>
<td>Method Name</td>
<td><code>performRQLCntQuery</code></td>
</tr>
<tr>
<td>Input Parameters</td>
<td><code>String pRepositoryPath</code>&lt;br&gt;The path of the repository component to query against.&lt;br&gt;<code>String pItemDescriptorName</code>&lt;br&gt;The item type of the repository items to query against.&lt;br&gt;<code>String pRQLString</code>&lt;br&gt;The RQL string to execute. Note that this string cannot contain parameters.</td>
</tr>
<tr>
<td>Output</td>
<td><code>int</code>&lt;br&gt;The number of RepositoryItems that satisfy the given query.</td>
</tr>
<tr>
<td>Exception</td>
<td><code>atg.repository.RepositoryException</code> if a repository error occurs</td>
</tr>
</tbody>
</table>

### PerformRQLCountQuery Exceptions

Exceptions generated by the `PerformRQLCountQuery` Web service may occur for the following reasons:

**RepositoryException**

- `The pRepositoryPath specified by the argument is null, or empty.`
- `The pRepositoryPath specified by the argument does not resolve to a component.`
- `The pRepositoryPath specified by the argument does not resolve to a Repository.`
- `The pItemDescriptorName specified by the argument does not identify an ItemDescriptor of the given repository.`
- `The pRQLString specified by the argument is null, or empty.`
- `The RQL code throws an exception during parsing or execution.`

### Web Service Generation

The following parameters are used to generate this Web service, using the `WebServiceGeneratorImpl` class:
### Repository Web Service Security

Each repository Web service defines a security function. This function lets you define a security policy that can be applied across many services at once. You can define these functional names and security policy relationships in the Web Service Security Configuration section of the ATG Web Service Administration interface. See Creating Web Services in the ATG Web Services and Integration Framework Guide for more information.

If you want to change any of these functional names in order to change the way different Web services are grouped, you must regenerate the Web services, because the functional name for security policy purposes is hard coded into the generated class. The functional name for each of the three repository Web services included in the ATG platform is `repositoryOperation`. By default, this functional name is mapped to a security policy that allows access only by Administrators. You are free to change this to suit your needs, depending on how you expect to use this service.
16 Composite Repositories

All ATG repositories provide a means for representing information in a data store as Java objects. The composite repository lets you use more than one data store as the source for a single repository. The composite repository consolidates all data sources in a single data model, making the data model flexible enough to support the addition of new data sources. Additionally, the composite repository allows all properties in each composite repository item to be queryable. Thus, from the point of view of your ATG application, the composite repository presents a consistent view of your data, regardless of which underlying data store the data may reside in.

The composite repository is a repository that unifies multiple data sources. Its purpose is to make any number of repositories appear in an ATG application as a single repository. The composite repository defines a mapping between item descriptors and properties as they appear to facilities that use the composite repository and item descriptors and properties of the data models that comprise the composite data model. A composite repository is composed of any number of composite item descriptors. Each item descriptor can draw on different data models from different repositories, and map underlying data model attributes in different ways.

**Use Example**

Suppose you maintain profile data both in an SQL database and an LDAP directory. ATG’s profile repository ships with a user composite item descriptor comprised of just one primary item descriptor and no contributing item descriptors. The primary item descriptor is the user item descriptor. You can add to the composite item descriptor the user item descriptor from the LDAP repository as a contributing item descriptor. If there are any property name collisions between the SQL repository and the LDAP repository, you can resolve them by mapping the properties explicitly to different names in the composite repository configuration. After you’ve done this, your ATG applications can view both LDAP profile information and SQL database profile information as properties of composite items in the composite user item descriptor.

**Primary and Contributing Item Descriptors**

Each composite item descriptor is composed of any number of contributing item descriptors. One of these contributing item descriptors must be designated as the primary item descriptor. The primary item descriptor’s main purpose is to provide the ID space for the composite item descriptor. The composite item descriptor can incorporate any number of contributing item descriptors, which contribute properties to the composite repository items.

Each contributing item has one or more relationships to the primary item. These relationships are defined in the contributing item descriptor. Each relationship defines a unique ID attribute in the primary item descriptor, as well as a unique ID attribute in the contributing item descriptor. The attribute can be the
repository item ID or a unique property. A contributing item is linked to a primary item if the value of its unique ID attribute matches the value of the primary item’s unique ID attribute. If multiple relationships are defined, they are AND’d together.

For example, suppose you have a contributing item descriptor that defines two relationships to the primary item descriptor. One says that a primary item’s firstName property must match the contributing item’s userFirstName property and the other says that the primary item’s lastName property must match the contributing item’s userLastName. These two relationships together mean that a user’s first names and last names must each match for two items to be related. This is useful in situations where no one property uniquely identifies a user. See link-via-property for an example of defining a relationship with two or more properties.

Item Inheritance and Composite Repositories

A composite repository can handle item descriptor inheritance only for its primary item descriptors. For example, suppose you have a user composite item descriptor. Its primary item descriptor is named person and is part of an LDAP repository. The contributing item descriptor is named user and is part of an SQL repository. The user item descriptor has a subtype named broker. The composite items have access to the properties of the person item descriptor and the user item descriptor, but not to properties that exist only in the broker item descriptor.

Transient Properties and Composite Repositories

An LDAP repository does not support transient properties. Therefore, if you want to use transient properties in your composite item descriptor, the transient properties must be derived from an SQL repository or other repository that does support transient properties.

Non-Serializable Items and Composite Repositories

An LDAP repository item is not serializable. Therefore, if you have a property that derives from an LDAP repository item, you should mark the property as not serializable by setting the serialize attribute to false:

```xml
<property name="propName">
  ...
  <attribute name="serialize" value="false"/>
  ...
</property>
```

Property Derivation

The properties in a composite item descriptor are determined as follows:

1. If configured to do so, all properties from the primary and contributing item descriptors are combined into the composite item descriptor, with each property retaining its property name and property type.
2. Any properties marked as excluded are removed from the composite item descriptor. See Excluding Properties.

3. All property mappings are performed. This means that a primary or contributing property that is to be mapped gets renamed in the composite item descriptor. See Property Mappings.

4. If there are any two properties in the composite item descriptor that have the same name, an error results. The composite repository requires that all composite property names map explicitly to only one primary or contributing property.

### Configuring a Composite Repository

1. Design the composite repository. Pick what item types you want to represent in your composite repository's composite item descriptors

2. Specify the primary item descriptor. This is where the composite repository item's repository item IDs come from

3. Specify any contributing item descriptors you need to supplement the primary item descriptor.

4. Resolve any property name collisions between properties in the primary item descriptor and the contributing item descriptors. See Property Mappings.

5. Determine whether you want to use static or dynamic linking for properties whose types are repository items. See Link Methods.

6. Determine what item creation policy you want the composite repository to implement. See Creating Composite and Contributing Items.

7. Determine whether there are any properties in your primary or contributing item descriptors that you want to exclude from the composite item descriptor. See Excluding Properties.

8. Create and configure a CompositeRepository component. See Configuring the Composite Repository Component.

### Property Mappings

The composite repository requires that all composite property names map explicitly to only one primary or contributing property. If primary or contributing item descriptors contain one or more properties with the same name, you must exclude one of the properties (see Excluding Properties) or map it to another name.

You can map a property with the mapped-property-name attribute in an item descriptor's property tag. For example, given two contributing item descriptors, where each has a login property, you can map one of the properties to a different name like this:

```xml
<property name="ldapLogin"... mapped-property-name="login"/>
```
In this example, the name attribute specifies the property name in the composite item descriptor and the mapped-property-name attribute specifies the name of the property in the primary or contributing item descriptor to which this property maps.

Excluding Properties

Sometimes you may not want to expose absolutely every property from the underlying primary and contributing item descriptors in the composite item descriptor. You can configure the item descriptor to exclude those contributing properties that are not desired. You do this by setting a property tag’s exclude attribute to true:

```xml
<property name="password... exclude="true"/>
```

Link Methods

The link-method attribute determines what happens when the composite repository needs to get a property value that belongs to a contributing repository item. For example, a process might call:

```java
CompositeItem.getPropertyValue("ldapFirstName");
```

where ldapFirstName is a property of a contributing repository item in an LDAP repository. The CompositeItem that is being asked for the property needs to look for this contributing item. If it can find it, it retrieves the property value and acts according to the value of the link-method attribute: static or dynamic.

**Static link method**

If link-method is set to static, the contributing item is stored in a member variable of that composite repository item. The next time a property is requested from that same item, it retrieves it from this variable instead of finding it again from the underlying contributing repository. This saves some computational effort and results in faster property retrieval.

If the value of the property or properties used to link to the underlying contributing item changes, the data in the member variable is stale. This occurs only if a linking property in the underlying data store changes. For example, if you link to a contributing item descriptor using a login property, static linking can result in stale data only if the login property changes in an underlying repository.

**Dynamic link method**

If link-method attribute is set to dynamic, the composite repository queries the underlying repository for the contributing item every time a property is requested from it. This might result in slower performance, but it also means that data is never out of sync at the repository level.

**Methods compared**

Dynamic link mode might seem like the most technically correct implementation, because the data model is guaranteed to reflect the latest information. Because dynamic link mode requires a query each time information is needed from a composite item, it can impair performance. Usually, the information that links items rarely changes. Static linking is generally provides correct data model linking.
Creating Composite and Contributing Items

The contributing-item-creation-policy dictates how contributing items are created (if at all) in a MutableRepository. This attribute can have a value of eager, lazy, or none.

**eager**

When users create a new composite item via the createItem() method in MutableCompositeRepository, new instances of the primary item and of all contributing items are created. So, for example, if you have a user item type defined in your composite repository that borrows properties from the SQL and LDAP repositories, any new user composite repository item that is created creates both an SQL repository item and an LDAP repository item. However, before these items can be added to their respective repositories, the correct link needs to exist between them. If the items are linked by a certain property, this property needs to be set on the primary item before the items are added, otherwise an error occurs as those two items cannot be linked back together later.

**lazy**

If this option is chosen, contributing items are created only when they are needed. In this case, when users call setPropertyValue on a property that is defined in the contributing repository, the composite repository creates the item in the contributing then and there. There are two different behaviors depending on whether the CompositeItem is transient or not.

- If the item is transient, wait until the item is persisted before checking to see that all appropriate linking properties are set, so they can be propagated to the new contributing item.
- If the item is not transient, check whether the correct linking properties are set on the primary item, then add the contributing item to its repository. If there any properties used for linking are missing, an error is returned.

The check for valid linking properties occurs during the updateItem call, and not during the setPropertyValue call on the contributing item. So if you use lazy item creation and call setPropertyValue on a persistent item, you do not need to already have valid values set for any linking properties on the primary item at that exact point in time. As long as the values of the linking properties are set before updateItem is called, the item should be successfully created.

**none**

If this option is chosen, no repository items are created in the underlying repositories under any circumstance. Any contributing items used in the composite repository must already exist in order for valid results to be returned from property value requests.

Missing Contributing Items

The null-contributing-item-policy attribute determines how the composite repository should behave if it tries to get the value of a property from a contributing repository item, but the repository cannot find a contributing item that links with the primary item. There are three possible behaviors:

**error**

If there is no contributing item found, a RuntimeException is thrown.
If there is no contributing item found, the default value for that property in the contributing item descriptor is returned. If there is no default value, null is returned.

If there is no contributing item found, null is returned automatically.

**Configuring the Composite Repository Component**

The `CompositeRepository` component, whose class is `atg.adapter.composite.MutableCompositeRepository`, is the central component of a composite repository. Create a component of this class and set its `configurationFile` property to the Nucleus address of the composite repository definition file. You can configure the following properties of this component:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>configurationFile</td>
<td>The Nucleus address of an XML file that uses the Composite Repository DTD. See the Composite Repository Definition Tag Reference.</td>
<td></td>
</tr>
<tr>
<td>cumulativeDebug</td>
<td>If true, output from all debug levels lower than the current debug level are printed to the log.</td>
<td>Boolean Default: true</td>
</tr>
<tr>
<td>debugLevel</td>
<td>An integer from 0 to 23 that indicates the frequency with which debug log messages are generated. The higher the value, the greater the frequency of debug log entries. See Debug Levels in the Developing and Testing an SQL Repository chapter.</td>
<td>Integer Default: 5</td>
</tr>
<tr>
<td>queryBatchSize</td>
<td>The maximum number of items that are returned by a single query to an underlying repository.</td>
<td>Integer Default: 1000.</td>
</tr>
<tr>
<td>repositoryName</td>
<td>The name of the composite repository</td>
<td>String</td>
</tr>
</tbody>
</table>

**Composite Repository Queries**

All queries in the Repository Query API are supported in the composite repository. However, a query against the composite repository should use only queries that are supported in the underlying repositories. You can make queries that reference properties of different underlying repositories. Be aware, however, that queries with expressions that involve joins across multiple repositories may be slower than single-repository queries. Queries that may perform extremely poorly are of the form “find all users whose `dayPhone` is equal to their `workPhone`,” where `dayPhone` and `workPhone` are stored in different repositories. If you construct a complex query that needs to retrieve some properties from one underlying repository and other properties from a separate underlying repository, the query must be
broken down into separate queries directed at each repository. The results of the sub-queries are combined appropriately using AND or OR rules and the final result set is returned in the composite repository.

Note in particular that COUNT queries perform poorly if the query spans repository views or if the underlying repository does not support executing count queries. LDAP repositories do not support COUNT queries, for example, and you should avoid using COUNT queries if any part of the result set might come from the LDAP repository.

Composite Repository Caching

The composite repository does not maintain items or queries in its own caches. Instead, it relies on the caches maintained by its underlying repositories. See the SQL Repository Caching chapter and Configuring LDAP Repository Components in the LDAP Repositories chapter for information about how those repositories handle caching.

Composite Repository Definition Tag Reference

This section describes composite-repository-template elements as they are defined in composite-repository_1.0.dtd. The complete DTD for composite repository definition files can be found later in this chapter.

This section describes the XML tags that can be used in a composite repository definition file, as defined in the DTD for Composite Repository Definition Files.

<composite-repository-template>

<!ELEMENT composite-repository-template (header?, item-descriptor*)>

The composite-repository-template tag encloses the entire composite repository definition.

<header> (composite repository)

<!ELEMENT header (name?, author*, version?, description?)>

Parent: <composite-repository-template>

The header tag provides information that can help you manage create and modify repository definition files.

For example:
<header>
  <name>Catalog Template</name>
  <author>Neal Stephenson</author>
  <author>Emily Dickinson</author>
  <version>$Id: catalog.xml,v 1.10 2000/12/24 03:34:26 hm Exp $</version>
  <description>Template for the store catalog</description>
</header>

<item-descriptor>
  composite repository

  <!ELEMENT item-descriptor (attribute*, primary-item-descriptor, contributing-item-descriptor*)>

Parent: <composite-repository-template>

The <item-descriptor> tag specifies the primary and contributing item descriptors that comprise a composite item descriptor.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the composite item descriptor, unique within the repository (required). This property is case-insensitive.</td>
</tr>
<tr>
<td>default</td>
<td>Boolean, specifies whether this is the composite repository's default item descriptor. The default item descriptor is used for new repository items if no item descriptor is explicitly specified.</td>
</tr>
<tr>
<td></td>
<td>If no item descriptor is designated as the default, the first item descriptor in the repository definition file is the default.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>display-property</td>
<td>Specifies a property of this item descriptor that is used to represent items of this type in a user interface. For example, a profile item descriptor might set display-property to login. Then, each repository item is represented using the value of the item's login property.</td>
</tr>
</tbody>
</table>
## Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>display-name-resource</td>
<td>If a resource bundle is specified for this property with the tag <code>&lt;attribute name='resourceBundle'&gt;</code>, this attribute specifies the resource bundle key to the item descriptor’s display name.</td>
</tr>
<tr>
<td></td>
<td>See <a href="#">Localizing SQL Repository Definitions</a>.</td>
</tr>
<tr>
<td>link-method</td>
<td>The method for retrieving properties from contributing repository items, one of the following:</td>
</tr>
<tr>
<td></td>
<td>static (default)</td>
</tr>
<tr>
<td></td>
<td>dynamic</td>
</tr>
<tr>
<td></td>
<td>See <a href="#">Link Methods</a>.</td>
</tr>
<tr>
<td>contributing-item-creation-policy</td>
<td>Specifies how contributing repository items are created, one of the following:</td>
</tr>
<tr>
<td></td>
<td>lazy (default)</td>
</tr>
<tr>
<td></td>
<td>eager</td>
</tr>
<tr>
<td></td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>See <a href="#">Creating Composite and Contributing Items</a>.</td>
</tr>
<tr>
<td>null-contributing-item-policy</td>
<td>Specifies what to do if a contributing repository item is requested but not found in the underlying repository, one of the following:</td>
</tr>
<tr>
<td></td>
<td>default (default)</td>
</tr>
<tr>
<td></td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>null</td>
</tr>
<tr>
<td></td>
<td>See <a href="#">Missing Contributing Items</a>.</td>
</tr>
</tbody>
</table>

### Example

```xml
<item-descriptor name='compositeUser' default='true'
display-property='fooProperty'
display-name-resource='itemDescriptorUser'>
    <attribute name='resourceBundle' value='atg.userprofiling.CompositeProfileTemplateResources'
data-type='string'/>
    <primary-item-descriptor.../>
    <contributing-item-descriptor.../>
...
</item-descriptor>
```
**<primary-item-descriptor>**

```xml
<!ELEMENT primary-item-descriptor (property*)>
```

**Parent:** item-descriptor

One item descriptor is designated as the primary item descriptor; it provides the ID space for the composite item descriptor.

**Attributes**

**Note:** The following attributes are also defined for the <contributing-item-descriptor> element.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
</table>
| name                             | The name of the composite item descriptor, unique within the repository (required).  
This property is case-insensitive.            |
| repository-item-descriptor-name  | The name of this item descriptor in its source repository (required).         |
| repository-nucleus-name          | The Nucleus address of this item descriptor's repository (required).           |
| all-properties-propagate        | Boolean. If this attribute is set to true, the composite repository tries to make all properties of the primary or contributing item descriptor available to the composite item descriptor.  
Default: false            |
| all-properties-queryable        | Boolean, specifies whether properties of this item descriptor are queryable by default. This setting can be overridden by explicitly setting the property's queryable attribute.  
Default: true            |

**<contributing-item-descriptor>**

```xml
<!ELEMENT contributing-item-descriptor (property*, primary-item-descriptor-link)>
```

**Parent:** item-descriptor

The contributing-item-descriptor element whose properties are combined with the primary item descriptor properties.
Attributes

See <primary-item-descriptor>

<attribute>
composite repository

<!ELEMENT attribute EMPTY>

Parent: <item-descriptor>, <property>

The <attribute> tag associates arbitrary name/string value pairs with a property or item type, which determine its behavior. The name/value pairs are added to the property descriptor via the setValue method of java.beans.FeatureDescriptor, and can later be used by the application.

For example:

<property name="employeeNumber" data-type="string">
  <attribute name="PCCExpert" value="true" data-type="boolean"/>
</property>

See User-Defined Property Types for more information.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of the name/value pair, required. You can specify any name here and it is added to the list of feature descriptor attributes for your property.</td>
</tr>
<tr>
<td>value</td>
<td>The value of the name/value pair, required. The data type of this value is defined by the data-type attribute supplied to this tag. If no data-type attribute is provided, the value of the attribute is a string.</td>
</tr>
<tr>
<td>data-type</td>
<td>The primitive data-type of the value, one of the following:</td>
</tr>
<tr>
<td></td>
<td>string, int, byte, short, date, long, timestamp, float, double</td>
</tr>
</tbody>
</table>

* default
### <property> composite repository

```xml
<!ELEMENT property (attribute*)>
```

**Parent:** `<primary-item-descriptor>,<contributing-item-descriptor>`

The `<property>` tag maps a property in a composite repository to a property in a primary or contributing item descriptor. This allows two or more contributing item descriptors to have properties with the same name.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The name of this composite property.</td>
</tr>
<tr>
<td>mapped-property-name</td>
<td>The name of the property in the primary or contributing item descriptor to which this property maps (required).</td>
</tr>
<tr>
<td>queryable</td>
<td>Boolean. Default: <code>true</code></td>
</tr>
<tr>
<td>required</td>
<td>Boolean. Default: <code>false</code></td>
</tr>
<tr>
<td>expert</td>
<td>Boolean. Expert properties are not displayed in the default view of the ATG Control Center. Default: <code>false</code></td>
</tr>
<tr>
<td>hidden</td>
<td>Boolean, if <code>true</code>, suppresses display in the ATG Control Center. Default: <code>false</code></td>
</tr>
<tr>
<td>readable</td>
<td>Boolean. Default: <code>false</code></td>
</tr>
<tr>
<td>writable</td>
<td>Boolean. Default: <code>false</code></td>
</tr>
<tr>
<td>category-resource</td>
<td>If a resource bundle is specified for this property with the tag <code>&lt;attribute name=&quot;resourceBundle&quot;&gt;</code>, this attribute specifies the resource bundle key to the property's category. See Localizing SQL Repository Definitions.</td>
</tr>
<tr>
<td>display-name-resource</td>
<td>If a resource bundle is specified for this property with the tag <code>&lt;attribute name=&quot;resourceBundle&quot;&gt;</code>, this attribute specifies the resource bundle key to the property's display name. See Localizing SQL Repository Definitions.</td>
</tr>
<tr>
<td>exclude</td>
<td>Boolean. If set to <code>true</code>, excludes this property from the composite item descriptor. See Excluding Properties. Default: <code>false</code></td>
</tr>
</tbody>
</table>
Example

```xml
<property name="ldapFirstName" mapped-property-name="firstName"
  queryable="false" required="false" expert="false"
  hidden="false" readable="true" writable="true"
  category-resource="categoryBasics"
  display-name-resource="ldapFirstName">
  ...
</property>
```

**<primary-item-descriptor-link>**

```xml
<!ELEMENT primary-item-descriptor-link (link-via-id | link-via-property+)> 
```

Parent: `<contributing-item-descriptor>`

The `<primary-item-descriptor-link>` tag specifies how to link items in contributing item descriptors to items in the primary item descriptor, by embedding a `<link-via-id>` tag or one or more `<link-via-property>` tags:

- If the `<primary-item-descriptor-link>` tag encloses a `<link-via-id>` tag, the repository ID of the item is used for linking.
- If the `<primary-item-descriptor-link>` tag encloses a `<link-via-property>` tag, a unique item property specified in the `<link-via-property>` tag is used for linking.

**Examples**

In the first example, the contributing item descriptor’s items are linked to the primary item descriptor’s items by the common repository ID of the items:

```xml
<primary-item-descriptor-link>
  <link-via-id/>
</primary-item-descriptor-link>
```

In the next example, a primary item is linked to an item in this contributing item descriptor if two conditions are true:

- The value of the primary item’s `firstName` property matches the value of the contributing item’s `userFirstName` property
- The value of the primary item’s `lastName` property matches the value of the contributing item’s `userLastName` property.

This is useful when no one property in the primary item descriptor or the contributing item descriptor is uniquely valued. The relationships are AND’ed together.
<primary-item-descriptor-link>
  <link-via-property primary="firstName" contributing="userFirstName"/>
  <link-via-property primary="lastName" contributing="userLastName"/>
</primary-item-descriptor-link>

See Link Methods for more information.

<link-via-id>
<!ELEMENT link-via-id EMPTY>

Parent: <primary-item-descriptor-link>

The <link-via-id> tag specifies to use the item's repository ID to link the primary item descriptor to the contributing item descriptor.

<link-via-property>
<!ELEMENT link-via-property EMPTY>

Parent: <primary-item-descriptor-link>

The <link-via-property> specifies to use one or more properties to link the primary item descriptor to the contributing item descriptor.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>The name of the property in the primary item descriptor used for linking. The property name used is the name in the underlying repository and not the names in the composite repository.</td>
</tr>
<tr>
<td>contributing</td>
<td>The name of the property in the contributing item descriptor used for linking. The property name used is the name in the underlying repository and not the names in the composite repository.</td>
</tr>
</tbody>
</table>

**DTD for Composite Repository Definition Files**

The DTD for composite repository definition files is installed in the archive <ATG10dir>/DAS/lib/classes.jar. It can also be referenced with this URL:
http://www.atg.com/dtds/gsa/composite-repository_1.0.dtd

<!DOCTYPE composite-repository-template [ 
<!ELEMENT composite-repository-template (header?, item-descriptor*)> 
<!ELEMENT header (name?, author*, version?, description?)> 
<!ELEMENT name (#PCDATA)> 
<!ELEMENT author (#PCDATA)> 
<!ELEMENT version (#PCDATA)> 
<!ELEMENT description (#PCDATA)> 
<!ATTLIST attribute 
 name CDATA #REQUIRED 
 value CDATA #REQUIRED 
 data-type CDATA #IMPLIED ]>
<!ELEMENT item-descriptor (attribute*, primary-item-descriptor, contributing-item-descriptor*)>

<!ATTLIST item-descriptor
    name CDATA #REQUIRED
    default %flag; "false"
    display-property CDATA #IMPLIED
    display-name-resource CDATA #IMPLIED
    link-method CDATA #IMPLIED
    contributing-item-creation-policy CDATA #IMPLIED
    null-contributing-item-policy CDATA #IMPLIED
>
<!-- =============================================================== -->
<!-- The primary item descriptor definition                        -->
<!--  The primary view's property values take precedence over      -->
<!--  contributing views' property values. Also, a composite item's -->
<!--  primary item provides the composite item's id.                -->
<!--  The repository-nucleus-name and view-name specify the primary -->
<!--  view. The unique-id-property specifies which property in the   -->
<!--  uniquely identifies items in the primary view.                -->
<!-- =============================================================== -->
<!ELEMENT primary-item-descriptor (property*)>
<!ATTLIST primary-item-descriptor
    name CDATA #REQUIRED
    repository-nucleus-name CDATA #REQUIRED
    repository-item-descriptor-name CDATA #REQUIRED
    all-properties-propagate %flag; "false"
    all-properties-queryable %flag; "true"
>
<!ELEMENT contributing-item-descriptor (property*, primary-item-descriptor-link)>
<!ATTLIST contributing-item-descriptor
    name CDATA #REQUIRED
    repository-nucleus-name CDATA #REQUIRED
    repository-item-descriptor-name CDATA #REQUIRED
    all-properties-propagate %flag; "false"
    all-properties-queryable %flag; "true"
>
<!ELEMENT property (attribute*)>
<!ATTLIST property
    name CDATA #IMPLIED
    mapped-property-name CDATA #REQUIRED
    queryable %flag; "true"
    required %flag; "false"
    expert %flag; "false"
Sample Composite Repository Definition File

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario-manager-configuration
PUBLIC "-//Art Technology Group, Inc./DTD Scenario Manager//EN"
'http://www.atg.com/dtds/composite-repository/composite-repository_1.0.dtd'>

<!-- composite repository definition -->
<composite-repository-template>

<!-- Header similar to GSA DTD -->
<header>
  <!-- name of this document -->
  <name>A sample Composite Repository template</name>
  <!-- author of this document -->
  <author>Graham Mather</author>
  <!-- version of this document -->
  <version>$Change: 226591 $DateTime: 2002/01/22 15:50:56 $Author: gm$
  </version>
</header>

<!-- composite item descriptor definition -->
<item-descriptor name="compositeUser" default="true"
  hidden %flag;  "false"
  readable %flag;  "true"
  writable %flag;  "true"
  category-resource CDATA #IMPLIED
  display-name-resource CDATA #IMPLIED
  exclude %flag;  "false">

<!ELEMENT primary-item-descriptor-link (link-via-id | link-via-property+)>    
<!ELEMENT link-via-id EMPTY>   
<!ELEMENT link-via-property EMPTY>  

<!ATTLIST link-via-property
  primary CDATA #REQUIRED
  contributing CDATA #REQUIRED
  sort-property %flag; #IMPLIED>
```

---

16 - Composite Repositories
display-property="fooProperty"
display-name-resource="itemDescriptorUser">

<!-- resource bundle from whence this item descriptor's resources come -->
<attribute name="resourceBundle"
    value="atg.userprofiling.CompositeProfileTemplateResources"
data-type="string"/>

<!-- icon for items of this type -->
<attribute name="icon" value="userIcon" data-type="string"/>

<!-- "basics" category sort priority -->
<attribute name="categoryBasicsPriority" value="10" data-type="int"/>

<!-- primary view definition -->

<!-- name: the name of the primary view, as it appears internally to the composite repository. The primary view and all composite views must have unique internal view names -->
<attribute name="repository-nucleus-name" value="/atg/userprofiling/ProfileAdapterRepository" data-type="string"/>
<attribute name="repository-item-descriptor-name" value="user" data-type="string"/>
<attribute name="all-properties-propagate" value="true" data-type="string"/>
<attribute name="all-properties-queryable" value="true" data-type="string"/>

<!-- Can also contain explicit property mappings and explicit property exclusions -->

<property mapped-property-name="lastName" exclude="true"/>
<property mapped-property-name="email" exclude="true"/>

</primary-item-descriptor>

<!-- contributing view definition -->

<!-- name: the name of this contributing view, as it appears to the composite repository -->
<attribute name="repository-nucleus-name" value="/atg/userprofiling/ProfileAdapterRepository" data-type="string"/>
<attribute name="repository-item-descriptor-name" value="user" data-type="string"/>
<attribute name="all-properties-propagate" value="true" data-type="string"/>
<attribute name="all-properties-queryable" value="true" data-type="string"/>
<contributing-item-descriptor name="UserProfile-LDAP"
  repository-nucleus-name="/atg/adapter/ldap/LDAPRepository"
  repository-item-descriptor-name="user"
  all-properties-propagate="true"
  all-properties-queryable="true">

  <!-- explicit property mapping
  sometimes it's advantageous to explicitly map a property in a composite view
  to a particular property in either the primary or a contributing view.
  For example, perhaps two contributing views have properties with the same
  name. This gets around the 'no contributing views with same property names'
  rule.
  -->

  <!-- name: name of this composite property -->
  <!-- mappedPropertyName: the property to which this property maps -->
  <!-- queryable: property queryable flag -->
  <!-- required: property required flag -->
  <!-- expert: property expert flag -->
  <!-- hidden: property hidden flag -->
  <!-- readable: property readable flag -->
  <!-- writable: property writable flag -->
  <!-- category-resource: resource for category name -->
  <!-- display-name-resource: resource for display name -->
  <property name="ldapFirstName" mapped-property-name="firstName"
    queryable="false" required="false" expert="false"
    hidden="false" readable="true" writable="true"
    category-resource="categoryBasics"
    display-name-resource="ldapFirstName">

    <!-- bundle for this property's resources -->
    <attribute name="resourceBundle" value="atg.userprofiling.CompositeProfileTemplateResources"
      data-type="string"/>

    <!-- flag for ui being able to write this property -->
    <attribute name="uiwritable" value="true" data-type="boolean"/>

    <!-- maximum length for this property -->
    <attribute name="maxLength" value="32" data-type="int"/>

    <!-- does this property's value have to be unique? -->
    <attribute name="unique" value="true" data-type="boolean"/>

    <!-- sort priority -->
    <attribute name="propertySortPriority" value="10" data-type="int"/>
  </property>
</contributing-item-descriptor>
Sometimes users will not want to expose absolutely every property from the underlying primary and contributing views in the composite view. An explicit property removal allows the user to make the composite view contain only those contributing properties that are desired.

```xml
<property mapped-property-name="login" exclude="true"/>
<property mapped-property-name="password" exclude="true"/>
<property mapped-property-name="id" exclude="true"/>
```

2) A composite view’s property names are determined thusly:

   a) If all-properties-propagate is true, all properties from the primary and contributing views are combined into the composite view, retaining their property names, property types, and any metadata they may have defined.

   b) All property exclusions are performed. This means that any properties to be excluded are removed from the composite view.

   c) All property mappings are performed. This means that a primary or contributing property that is to be mapped gets renamed in the composite view.

   d) If there are any two properties in the composite view that have the same name, error. The composite repository requires that all composite property names map explicitly to only one primary or contributing property.

```xml
<!-- the primary view link describes how items in the contributing view are linked to items in the primary view. For each primary-contributing relationship, the user picks a unique id attribute for the primary and the contributing view. The attribute can be either the repository id of the item or a uniquely-valued property of the item (e.g. login). A primary item is linked to a contributing item if its unique id attribute value matches the unique id attribute value of the contributing item. There must be at least one primary view link, but there is primary view link limit. -->
```

```xml
<!-- example: this primary view link defines a relationship where an item in the primary view is linked to an item in this contributing view if the contributing item has a repository id which is the same as the primary item's id. -->
```
<link-via-id/>
</primary-item-descriptor-link>

This primary view link defines a relationship where a primary view item is linked to an item in this contributing view if the value of the primary item's "login" property matches the value of the contributing item's "userLoginName" property.

<primary-item-descriptor-link>
<link-via-property primary="login" contributing="login"/>
</primary-item-descriptor-link>

This primary view link defines a relationship where a primary view item is linked to an item in this contributing view if the value of the primary item's "firstName" property matches the value of the contributing item's "userFirstName" property AND the value of the primary item's "lastName" property matches the value of the contributing item's "userLastName" property. This is useful in the case where no one property in the primary view or the contributing view is uniquely valued. The relationships are ANDed together.

<primary-item-descriptor-link>
<link-via-property primary="firstName" contributing="userFirstName"/>
<link-via-property primary="lastName" contributing="userLastName"/>
</primary-item-descriptor-link>

</contributing-item-descriptor>
</item-descriptor>
</composite-repository-template>
17 Secured Repositories

The ATG secured repository system works in conjunction with the ATG Security System to provide fine-grained access control to repository item descriptors, individual repository items, and individual properties through Access Control List (ACL) settings.

This chapter includes the following sections:

- Features and Architecture
- Creating a Secured Repository
- ACL Syntax
- Secured Repository Definition File Tag Reference
- DTD for Secured Repository Definition File
- Secured Repository Example

Features and Architecture

Secured repositories provide the following control features:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control access to repository item descriptors</td>
<td>Control who can create, add, remove, and query items defined by an item descriptor; similar to controlling access to a whole database table.</td>
</tr>
<tr>
<td>Control access to individual repository items</td>
<td>Control who can read, write, destroy, and query a repository item; similar to controlling access to a single database row.</td>
</tr>
<tr>
<td>Control access to properties of all repository items in a repository item descriptor</td>
<td>Control who can read or write a property in any repository item defined by an item descriptor; similar to controlling access to a database table column. A default ACL can be assigned to all items in the item descriptor that lack an explicit ACL.</td>
</tr>
</tbody>
</table>
## Feature Description

- **Control access to properties of an individual repository item**
  
  Control who can read or write a particular property in a repository item; similar to controlling the field of a database table row.

  An ACL that is assigned to a property overrides the ACL that is specified for that property in the item descriptor definition.

- **Limit query results**
  
  Control who can receive repository items that are returned by a repository query.

- **Set ownership of a repository item**
  
  At creation time, the current user is assigned as the owner of the new repository item. The owner can query a repository item and modify its ACL; otherwise this is simply an association of an identity to an item.

- **Automatically generate ACLs for new repository items**
  
  When a repository item is created, it is assigned an ACL that is constructed out of an ACL fragment and a template for the creator/owner (creator) and each group the owner belongs to.

These features are configured according to the needs of your application. Some features require additional storage in the underlying repository, or can have a significant impact on performance (see Performance Considerations later in this chapter). Consequently, you should only enable those features that the application requires.

### Access rights

Access to secured repositories is managed by building ACLs that associate certain access rights with certain identities—individual users, as well as groups, organizations, and roles that are associated with multiple users. The following table lists access rights that apply to the secured repository system.

**Note:** Not all access rights are available in all implementations or instances of a secured repository.

<table>
<thead>
<tr>
<th>Action/targets</th>
<th>Access right</th>
</tr>
</thead>
</table>
| CREATE RepositoryItemDescriptor | Create a repository item with an item descriptor.  
**Note:** Adding a new item to the repository also requires WRITE access to the same RepositoryItemDescriptor. |
| DELETE RepositoryItemDescriptor | Remove items of this RepositoryItemDescriptor type.  
**Note:** Deleting an item also requires DESTROY access to that Item. |
### Action/targets | Access right
---|---
DESTROY RepositoryItem | Remove the repository item from the repository and destroy its contents. **Note:** Most secured repositories also require DELETE access to the item’s RepositoryItemDescriptor.
LIST RepositoryItem | Query a repository item. LIST access is required in order for queries to return this repository item. An item’s owner implicitly has LIST access.
READ RepositoryItemDescriptor RepositoryItem Property | Enable read access to items of this RepositoryItemDescriptor type; or to the specified repository item; or to the specified item property.
READ_ACL RepositoryItem | Inspect the ACL of a repository item. This access right is implicitly granted to the repository item’s owner.
READ_OWNER RepositoryItem | Inspect the owner of a repository item.
WRITE RepositoryItemDescriptor RepositoryItem Property | Enable addition of items of this RepositoryItemDescriptor type; or updates to the contents of the specified repository item or the specified item property. **Note:** WRITE access to an item descriptor only enables addition of repository items; it does not allow updates to repository items.
WRITE_ACL RepositoryItem | Change the ACL of a repository item. This access right is implicitly granted to the repository item’s owner.
WRITE_OWNER RepositoryItem | Change the owner of a repository item.

**Note:** Securing a repository does not provide complete security within an application: the unprotected repository that it overlays is still available within the Nucleus name space, so it remains available to developers. The ATG Control Center can be configured to hide unprotected repositories, and an application can choose not to use an unprotected repository, so as not to expose unprotected data to end users.

### Creating a Secured Repository

To overlay an existing repository with a secured repository:

1. **Modify the underlying repository** by editing the definitions of the item descriptors you wish to secure.
2. **Configure the secured repository adapter** component.
3. Register the secured repository adapter component.

4. Create the secured repository definition file, an XML file that specifies access rights and owner information. Access rights are specified with the syntax described in the ACL Syntax section.

Modify the Underlying Repository

In order to secure a repository item descriptor, create a property that stores the ACL for that item. In order to define an owner for an item type, also create a property that stores the owner's name. For example:

```xml
<item-descriptor name="cheese">
    <property name="country" data-type="string" />
    <property name="runniness" data-type="int" />
    <property name="ACL" data-type="string" />
    <property name="cheeseOwner" component-type="user" />
</item-descriptor>
```

The properties that you add to the underlying repository are identified in the secured repository definition file by these two tags:

```xml
<owner-property name="value"/>
<acl-property name="value"/>
```

For example, given the previous example, you update the secured repository's `item-descriptor` definition as follows:

```xml
<acl-property name="ACL" />
<owner-property name="cheeseOwner" />
```

**ACL property length constraints**

The length of an ACL is limited by the amount of space available in the ACL property that is defined in the unsecure (underlying) repository. An overlong ACL generates a repository exception when it is set. This problem can occur when you use the `create-group-acl-template` in the secured repository definition to define an ACL for the owner's group, and the owner belongs to many groups.

To avoid this problem, define the ACL property as an array of strings, so the ACL is concatenated from the stored substrings. For example:

```xml
<item-descriptor name="cheese">
    ...
    <table name="test_items_acls" type="multi"
        id-column-names="id"
        multi-column-name="index">
        <property name="ACL" column-names="acl" data-type="array"
```
The `maxFragmentSize` attribute sets the maximum length of a string in any array index. The default value is 254. Set `maxFragmentSize` to the size of the database string column. For many databases, 254 is the appropriate value for a VARCHAR of unspecified length.

### Configure the Secured Repository Adapter Component

You configure a secured repository adapter component by setting the following properties:

- `$class`
- `name`
- `repositoryName`
- `repository`
- `configurationFile`
- `securityConfiguration`

**$class**

Java class, one of the following:

- Unversioned repository:
  `atg.adapter.secure.GenericSecuredMutableRepository`
- Unversioned content repository:
  `atg.adapter.secure.GenericSecuredMutableContentRepository`
- Versioned repository:
  `atg.adapter.secure.GenericSecuredMutableVersionRepository`
- Versioned content repository:
  `atg.adapter.secure.GenericSecuredMutableVersionContentRepository`

**name**

A description of the Secured Repository component that appears in the ACC.

**repositoryName**

The name of the Secured Repository component. For example:

`SecuredTestRepository`

**repository**

The name of the underlying repository that the secured repository adapter acts on. For example:
Test Repository

**configurationFile**

The repository definition file used by the secured repository adapter. See Create the Secured Repository Definition File. For example:

`secured-test-repository.xml`

**securityConfiguration**

The `atg.security.SecurityConfiguration` component to use. For more information about security policies and other security features, see the Managing Access Control chapter in the ATG Programming Guide. For example:

`/atg/dynamo/security/SecuredRepositorySecurityConfiguration`

Register the Secured Repository Adapter Component

After you configure a secured repository adapter, you register it with the `/atg/registry/ContentRepositories` component, by adding it to its `initialRepositories` property:

`initialRepositories+=/SecuredTestRepository`

This exposes the secured repository adapter to the ATG Control Center Repository Editor, and ensures its activation on application startup.

Create the Secured Repository Definition File

The secured repository adapter's `configurationFile` property specifies an XML file that defines the behavior of the secured repository. The default name of this file is `secured-test-repository.xml`. Its format is similar to that of the definition file for the underlying repository, using the same `item-descriptor` and `property` tags to delimit information about individual item descriptors and their related properties. The DTD for this file is described later in this chapter, in DTD for Secured Repository Definition File.

The following table describes the attributes that can be defined for each item descriptor:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>descriptor-acl</td>
<td>The ACL that applies to the item descriptor. This can contain any access right that applies to the item descriptor. The value of this tag is an ACL string, as defined in the ACL Syntax section.</td>
</tr>
<tr>
<td>default-acl</td>
<td>The <code>default-acl</code> element specifies the ACL that is applied to an item or property descriptor when it has no other ACL. This ACL can contain any access right that applies to the item descriptor or property. The value of this tag is an ACL string, as defined in the ACL Syntax section.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>owner-property</td>
<td>This defines the name of the string property in the underlying repository that is to be used to store the name of the owner of a repository item.</td>
</tr>
<tr>
<td>acl-property</td>
<td>This defines the name of the string property in the underlying repository that is used to store the ACL for an individual repository item.</td>
</tr>
<tr>
<td>creation-base-acl</td>
<td>An ACL fragment that is inserted into the default ACL for a newly created repository item. Typically this defines global access rights for administrators and limited access rights for the user base as a whole. This ACL fragment can contain any access right that applies to a repository item.</td>
</tr>
<tr>
<td>creation-owner-acl-template</td>
<td>An ACL template that is used to generate an ACL fragment that applies to the owner (creator) of a newly created repository item. This is a standard format ACL string with a dollar sign ($) used to indicate the owner identity. No other identities may be used in the template.</td>
</tr>
<tr>
<td>creation-group-acl-template</td>
<td>An ACL template that is used to generate an ACL fragment that applies to each group that the owner (creator) is a member of in a newly created repository item. This is a standard format ACL string with a dollar sign ($) used to indicate the group identity. No other identities may be used in the template.</td>
</tr>
</tbody>
</table>

Because a user may have a great many groups that they are a member of, it is suggested that this feature be used sparingly. For example, the ACC admin user may have enough groups to create an ACL that is too large for the example repository. For a description of what constitutes membership in a group, see Group Membership.

You can use a subset of these options to define ACLs for properties as well as item descriptors:

- descriptor-acl
- default-acl
- acl-property
- creation-base-acl
- creation-owner-acl-template
- creation-group-acl-template

See also the Secured Repository Definition File Tag Reference.

**Group Membership**

An identity is considered to be a group that the owner (creator) is a member of if the owner’s Persona lists it with its getSubPersonae() call. Exactly what is returned by this call varies according to the implementation of the User Authority.

The standard User Authority used here is implemented on top of the User Directory interface, and includes every Effective Principal of the user as a sub-Persona. For the Profile User Directory, this includes
all Organizations, Roles, and Relative Roles of the user as well as all Organizations, Roles and Relative Roles of any Organization they are members of (explicitly or implicitly). For the Admin User Directory, this includes all Groups that the ACC account is a member of, but not the Privileges that the Group is assigned.

**ACLs and Personae**

When creating ACLs, the Personae that are used for user identities must be created by the same User Authority that is used by the secured repository. The User Authority must not be a proxy even if the Personae produced by a proxy test are equivalent to the Personae produced by the User Authority for which it is a proxy. This is because the identity name spaces used by a User Authority and its proxies may not be the same, and the ACL parser cannot support multiple identity namespaces.

**Secured Repository Example**

This example shows how to add security to a simple repository. This repository contains a single secured item type with two properties, one secure and one unsecure.

You create a secure repository in the following steps:

1. Modify the SQL for the repository data store.
2. Modify the XML definition file of the unsecure repository.
3. Define the secured repository adapter’s definition file.
4. Configure a secured repository adapter component.
5. Register the repositories.

**Modify the SQL for the Repository Data Store**

The original SQL looks like this:

```sql
-- test-repository.ddl
create table test_items (  
  -- the ID of this item
  id varchar,
  -- a secured property of this item
  secured_property varchar,
  -- an unsecured property
  unsecured_property varchar,
)
```

**Modifications**

Add three fields to the SQL to enable storage of security information for the repository item’s owner and ACL, and the secured property’s ACL:

The modified SQL looks like this (changes are in boldface):

```sql
-- test-repository.ddl
create table test_items (  
  -- the ID of this item
  id varchar,
  -- a secured property of this item
  secured_property varchar,
  -- an unsecured property
  unsecured_property varchar,
  -- owner of this item
  owner varchar,
  -- ACL of this item
  acl varchar,
  -- secured property’s ACL
  secured_acl varchar,
)
```
Modify the XML definition file

The original repository definition file looks like this:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template PUBLIC "//Art Technology Group, Inc.//DTD Dynamo Security//EN" "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Test Repository</name>
  </header>
  <item-descriptor name="test_items" default="true">
    <table name="test_items" type="primary" id-column-names="id">
      <property name="secured_property" column-names="secured_property" data-type="string"/>
      <property name="unsecured_property" column-names="unsecured_property" data-type="string"/>
    </table>
  </item-descriptor>
</gsa-template>
```

**Modifications**

As with the SQL, you must add properties to the repository definition as follows (changes are in boldface):

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE gsa-template PUBLIC "//Art Technology Group, Inc.//DTD Dynamo Security//EN" "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">
<gsa-template>
  <header>
    <name>Test Repository</name>
  </header>
  <item-descriptor name="test_items" default="true">
    <table name="test_items" type="primary" id-column-names="id">
      <property name="secured_property" column-names="secured_property" data-type="string"/>
      <property name="unsecured_property" column-names="unsecured_property" data-type="string"/>
      <property name="item_owner" column-names="item_owner" data-type="string"/>
      <property name="item_acl" column-names="item_acl" data-type="string"/>
      <property name="secured_property_acl" column-names="secured_property_acl" data-type="string"/>
    </table>
  </item-descriptor>
</gsa-template>
```
Define the Secured Repository Adapter’s Definition File

Create the secured repository layer over this SQL repository. The secured repository’s XML definition file looks like this:

```xml
<!-- secured-test-repository.xml -->
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE secured-repository-template
 PUBLIC "-//Art Technology Group, Inc.//DTD Dynamo Security//EN" 
 "http://www.atg.com/dtds/security/secured_repository_template_1.1.dtd">
<secured-repository-template>
  <item-descriptor name="test_items">
    <!-- The ACL that applies to the item view/descriptor -->
    <descriptor-acl value="Admin$role$administrators-group:
      read,write,create,delete;
      Admin$role$everyone-group:read"/>

    <!-- The property where the ownership is stored -->
    <owner-property name="item_owner"/>

    <!-- The property where ACL is stored in -->
    <acl-property name="item_acl"/>

    <!-- An ACL fragment that is assigned to all new items -->
    <creation-base-acl value="Admin$role$administrators-group:
      read,write,list,destroy,read_owner,write_owner,read_acl,write_acl;
      Admin$role$everyone-group:read,list"/>
  </item-descriptor>
</secured-repository-template>
```
Configure a Secured Repository Adapter Component

Configure the secured repository adapter component as follows:

```
# SecuredTestRepository.properties
$class=atg.adapter.secure.GenericSecuredMutableRepository
$scope=global
name=Test repository for the secured repository implementation
repositoryName=SecuredTestRepository
# the repository that we're wrapping
repository=TestRepository
# The template file that configures the repository
configurationFile=secured-test-repository.xml
# The security configuration component used by the repository
securityConfiguration=/atg/dynamo/security/SecuredRepositorySecurityConfiguration
# Various Dynamo services we need
XMLToolsFactory=/atg/dynamo/service/xml/XMLToolsFactory
transactionManager=/atg/dynamo/transaction/TransactionManager
```
Note: The previous example uses the \texttt{creation-group-acl-template} feature for both repository items and the secured property. This setting should generally be removed if you are setting up a repository based on this code. For more information, see \texttt{creation-group-acl-template}.

Register the Repositories

In order to expose the two repositories to the ATG Control Center Repository Editor, and to activate them on application startup, you must add them to the \texttt{initialRepositories} property of the \texttt{/atg/registry/ContentRepositories} component:

\begin{verbatim}
initialRepositories+=/TestRepository,/SecuredTestRepository
\end{verbatim}

ACL Syntax

ACL strings in ATG are made up of a series of Access Control Entries (ACEs) separated from each other by semicolons:

\begin{verbatim}
ACL ::= ACE | ';' ACE +
\end{verbatim}

Each ACE is made up of colon-delimited parts:

- Identity
- List of access rights

These can be surrounded by an ACE type specifier that determines whether the ACE grants or denies rights:

\begin{verbatim}
ACE ::= ( IDENTITY ':' ACCESS_RIGHTS_LIST ) |
( ( "grant" | "deny" ) '{' IDENTITY ':' ACCESS_RIGHTS_LIST '}' )
\end{verbatim}

The "grant" modifier is the default, and can be omitted. If a "deny" ACE exists where a "grant" ACE also applies, the standard security policy denies access.

An identity is the literal string used by the User Authority to look up the identity's Persona. The standard User Authority (\texttt{/atg/dynamo/security/UserAuthority} in Nucleus) encodes the identity as follows:

\begin{verbatim}
UD_IDENTITY ::= UD-name '$' principal-type '$' UD-principal-key
\end{verbatim}

where:

- \texttt{UD-name} is the name of the User Directory as configured in the User Directory User Authority (usually \texttt{Admin} for the ACC account database, or \texttt{Profile} for the Profile Repository)
- \texttt{principal-type is user, org or role}
- **UD-principal-key** is the primary key for looking up the principal in the User Directory. The primary key varies among User Directory implementations. The primary key is a numeric ID for Profile User Directories, but is the account name—for example, admin, administrators-group—for the ACC account User Directory.

ATG comes configured with three other User Authorities:

- `/atg/dynamo/security/AdminUserAuthority` for looking up ACC accounts
- `/atg/userprofiling/ProfileUserAuthority` for looking up Profile accounts
- `/atg/dynamo/service/j2ee/J2EEUserAuthority` for looking up J2EE accounts and roles.

These user authorities look up Persona information based on the unencoded name of the identity and are typically used for performing authentication. They are, however, front-ends for the standard User Authority and produce Personae that are equivalent to those produced by the standard User Authority. (Note the caveat regarding the mixing of User Authorities in the Create the Secured Repository Definition File: ACLs and Personae topic.)

The list of access rights is a comma-separated list of access right names:

\[
\text{access-right-list ::= access-right [ ', ' access-right ]+}
\]

**Standard Access Rights**

The standard access right names are:

- create
- delete
- destroy
- execute
- list
- privilege
- read
- read_acl
- read_owner
- rename
- traverse
- write
- write_acl
- write_owner

Only the access rights appropriate for the ACL context are allowed. Access right names are tokens and cannot be internationalized.

**ACL Examples**

The following examples are coded using the syntax used by the standard `/atg/dynamo/security/UserAuthority` component.
The following ACL grants everyone with an ACC account the ability to read:

```
Admin$role$everyone-group:read;
```

Note that you should always end ACL strings with a semi-colon, as shown, even when the string is the last one in a list. Do not start ACLs with a semi-colon. Following this convention is important because it ensures that ACLs are interpreted correctly after XML-combine operations.

The following ACL grants the ACC admin account the ability to read and write, but every other ACC user only the ability to read:

```
Admin$user$admin:list,read,write;
Admin$role$everyone-group:list,read;
```

The following ACL grants the ACC Administrators group the ability to read, write and delete, but denies the ability to write and delete to ACC user Fnord even if he is a member of the Administrators group:

```
Admin$role$administrators-group:
list,read,write,delete;deny{Admin$user$Fnord:write,delete};
```

### Secured Repository Definition File Tag Reference

This section describes all XML tags that can be used in a secured repository definition file, as defined in the DTD for Secured Repository Definition File.

#### `<secured-repository-template>`

<table>
<thead>
<tr>
<th>Unauthorized Security Violation</th>
<th>Authorization Security Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;secured-repository-template&gt;</code></td>
<td><code>&lt;secured-repository-template&gt;</code></td>
</tr>
</tbody>
</table>

The `<secured-repository-template>` tag encloses the whole secured repository definition. The `<secured-repository-template>` tag encloses one `<item-descriptor>` tag for each item descriptor in the underlying repository for which you want to specify access rights.

**Example**

```
<secured-repository-template>
    <item-descriptor name="..." />
    ...
</secured-repository-template>
```
<item-descriptor>
  secured repository
</item-descriptor>

```
<item-descriptor name="feature">
  <descriptor-acl value="..."/>
  <owner-property name="..."/>
  <acl-property name="..."/>
  ...
</item-descriptor>
```

You should include one `item-descriptor` tag for each item descriptor in the underlying repository for which you want to specify access rights. Unlike the `item-descriptor` tag in the SQL repository, the `item-descriptor` tag in the secured repository has just one attribute, `name`, which must be the same as the `name` attribute in the underlying repository's `item-descriptor` tag.

**Example**

```
<item-descriptor name="yumminess">
  <descriptor-acl value="..."/>
  <acl-property name="yummy_acl"/>
  <creation-base-acl value="...
  ...
</item-descriptor>
```

You should include one `property` tag for each property in the underlying repository for which you want to specify access rights. Unlike the `property` tag in the SQL repository, the `property` tag in the secured repository has just one attribute, `name`, which must be the same as the `name` attribute in the underlying repository's `property` tag.

**Example**

```
<property name="yumminess">
  <descriptor-acl value="..."/>
  <acl-property name="yummy_acl"/>
  <creation-base-acl value="...
  ...
</property>
```
The default-acl element specifies the ACL that is applied to an item or property descriptor when it has no other ACL. This ACL can contain any access right that applies to the item descriptor or property. The value attribute of the tag is an ACL string, using the syntax described in the ACL Syntax section.

Example

```
<default-acl value="Admin$role$everyone-group:list,read;"/>
```

The descriptor-acl element specifies the ACL that applies to the item or property specified by the enclosing item-descriptor tag or property tag. This ACL can contain any access right that applies to the item descriptor or property. The value attribute of the tag is an ACL string, using the syntax described in the ACL Syntax section.

Example

```
<descriptor-acl value="Admin$role$administrators-group:list,read,write;
   Admin$role$everyone-group:list,read;"/>
```

The owner-property tag has one attribute, name, which specifies the name attribute of the property that stores the owner of the item in the underlying repository.

Example

If the item descriptor in the underlying repository stores the name of the item’s owner in a property named item_owner, the owner-property tag looks like this:
The `<acl-property>` tag has one attribute, `name`, which specifies the `name` attribute of the property that stores the ACL of the item in the underlying repository.

**Example**

If the item descriptor in the underlying repository stores the item's ACL in a property named `item_acl`, the `<acl-property>` tag would look like this:

```
<acl-property name="item_acl"/>
```

The `<creation-base-acl>` tag defines an ACL fragment that is inserted into the default ACL for a newly created repository item or property. Typically this defines global access rights for administrators and limited access rights for the user base as a whole. This ACL fragment can contain any access right that applies to a repository item or property.

**Example**

The following example gives all access rights to the administrators group, but only read and list rights to everyone else:

```
<creation-base-acl value="Admin$role$administrators-group:
    read,write,list,destroy,read_owner,write_owner,read_acl,write_acl;
Admin$role$everyone-group:read,list;"/>
```
Parent: <item-descriptor>, <property>

The creation-owner-acl-template tag specifies an ACL template that is used to generate an ACL fragment that applies to the owner (creator) of a newly created repository item. This is a standard format ACL string with a dollar sign ($) used to indicate the owner identity. No other identities may be used in the template.

**Example**

The following example gives the owners of an item access rights to read, write, list, or destroy items they own:

```
<creation-owner-acl-template value="$:read,write,list,destroy;"/>
```

<creation-group-acl-template>

```xml
<!ELEMENT creation-group-acl-template (#PCDATA)>
```

Parent: <item-descriptor>, <property>

The creation-group-acl-template tag specifies an ACL template that is used to generate an ACL fragment that applies to each group that the owner is a member of in a newly created repository item. This is a standard format ACL string with a dollar sign ($) used to indicate the group identity. No other identities may be used in the template.

Because a user may be a member of a large number of groups, you should use this feature sparingly. It can result in ACL strings that are too long to be stored in your repository’s ACL property. For example, the ACC admin user may have enough groups to create an ACL that is too large for the example repository. For a description of what constitutes membership in a group, see Group Membership.

**Example**

The following example gives read and list access rights to every member of every group of which the item’s owner is a member:

```
<creation-group-acl-template value="$:read,list;"/>
```

**DTD for Secured Repository Definition File**

The document type definition (DTD) for a secured repository definition is available at:

http://www.atg.com/dtds/security/secured_repository_template_1.1.dtd
The secured repository definition DTD looks like this:

```xml
<?xml encoding="UTF-8"?>
<!ELEMENT secured-repository-template (item-descriptor)*>
<!ATTLIST item-descriptor name CDATA #REQUIRED>
<!ELEMENT property (descriptor-acl | default-acl | creation-base-acl | creation-owner-acl-template | creation-group-acl-template | property)*>
<!ATTLIST property name CDATA #REQUIRED>
<!ELEMENT descriptor-acl (#PCDATA)>
<!ATTLIST descriptor-acl value CDATA #REQUIRED>
```
<!-- default-acl - specifies the ACL that is applied to either an item or property descriptor when it has no other ACL. -->
<!ELEMENT default-acl (#PCDATA)>  
<!ATTLIST default-acl value CDATA #REQUIRED>  

<!-- owner-property - specifies the name of the property in which the name of the owner of the item is stored. -->
<!ELEMENT owner-property (#PCDATA)>  
<!ATTLIST owner-property name CDATA #REQUIRED>  

<!-- acl-property - specifies the name of the property in which the ACL for the item or property is stored. -->
<!ELEMENT acl-property (#PCDATA)>  
<!ATTLIST acl-property name CDATA #REQUIRED>  

<!-- creation-base-acl - specifies the base ACL fragment that will be applied to all new items or properties when they are created. -->
<!ELEMENT creation-base-acl (#PCDATA)>  
<!ATTLIST creation-base-acl value CDATA #REQUIRED>  

<!-- creation-owner-acl-template - specifies the ACL fragment -->
Performance Considerations

While care is taken to maintain high performance, use of the secured repository does have some impact on the performance of the repository and, in some cases, the impact is considerable.

For access control defined at the item descriptor level (for example, `Repository.getItem()`, `MutableRepository.createItem()`, `MutableRepository.addItem()`, `MutableRepository.updateItem()`) the overhead of handling access checks amounts to the testing of the access control list for the item descriptor. This is normally minimal.

The exception to this rule is with the use of the `RepositoryView.executeQuery()` and `RepositoryView.executeCountQuery()` family of methods whenever ACLs are specified for individual repository items. In this case, the ACL of each repository item must be consulted to determine if it should be allowed in the result of the query, or counted as part of a count query. If the result set is large, the time required to parse and check all ACLs can be long. Furthermore, in the count query case, a full query must be done and its results counted. Thus, if your application uses count queries to limit expensive queries, the features afforded by a secured repository are very expensive.
Access control overhead at the repository item level is noticeable, but is incremental. When the repository item is loaded, its ACL is parsed before any access checking occurs. Results of ACL parsing are cached to improve performance where possible. If ACLs are not being stored for each individual repository item, no parsing needs to be done beyond what is done during the initialization of the secured repository.

Because the secured repository sits on top of an underlying repository, you can consider whether features that need best possible performance should be written to use the underlying repository rather than going through the secured repository at the cost of the security features.

Exceptions Thrown by the Secured Repository

Most methods implemented by the secured repository can throw any exception that is a sub-class of atg.security.SecurityException. Each method that can throw a SecurityException is marked appropriately in one of the following interfaces:

- SecuredRepository
- SecuredMutableRepository
- SecuredRepositoryItemDescriptor
- SecuredRepositoryView
- SecuredRepositoryItem

Methods that are inherited from the normal Repository interfaces and are marked as capable of throwing an atg.repository.RepositoryException instead throw an atg.repository.RepositorySecurityException. You can use the method RepositorySecurityException.getSecurityException() to determine the nested exception.

Two methods do not throw a RepositoryException:

- RepositoryItem.getPropertyValue()
- MutableRepositoryItem.setPropertyValue()

It is necessary, however, for them to throw a SecurityException, so they throw an atg.security.RuntimeSecurityException. You can use the method RuntimeSecurityException.getSecurityException() to determine the nested exception.
18 LDAP Repositories

The ATG LDAP Repository is an implementation of the Repository API that enables you to store and access profile data in an LDAP (Lightweight Directory Access Protocol) directory. The LDAP repository is similar in functionality to the SQL repository, as described earlier in this guide. While by default ATG Scenario Personalization is configured to use an SQL profile repository, you can change the configuration to use an LDAP repository instead. See the ATG Personalization Programming Guide for information about configuring ATG to use an LDAP profile repository. LDAP directories are widely used to store personnel information and other kinds of data. LDAP repository lets you to tap into the profile data you already have in an LDAP directory, and to share user information across multiple applications.

Also, you can configure ATG’s application security scheme to use an LDAP repository, rather than an SQL repository. See the Managing Access Control chapter in the ATG Programming Guide for more information.

Just like the SQL repository, the LDAP repository implements the ATG Repository API to allow you to store, access, modify, and query user profile information. As in the SQL repository, repository items are first created as transient items (RAM profiles); they become persistent after they are added to the database.

It is important to note, however, that the LDAP repository implementation is not specific to user profiles in any way. Because an LDAP directory can be used to store any kind of data—people, groups, mailing lists, documents, printers—you can use the LDAP repository to expose any of that data in ATG. This chapter focuses on using LDAP as a profile repository, because that is the most common application of LDAP. However, other uses are possible.

This chapter includes the following sections:

- **Overview: Setting Up an LDAP Repository**: An overview of the steps you should take in designing and setting up an LDAP repository.
- **LDAP Directory Primer**: A brief introduction to LDAP concepts and terminology
- **LDAP Repository Architecture**: A description of the way item descriptors and repository items work in the LDAP repository.
- **Repository Views in the LDAP Repository**: How multiple Repository Views can support multiple item descriptors in the same repository.
- **LDAP Repository Queries**: A brief look at how queries work in the LDAP repository.
- **Configuring LDAP Repository Components**: How to configure the components that make up the LDAP repository.
- **LDAP Repository Definition Tag Reference**: A detailed reference on the XML tags used to define an LDAP repository.
Overview: Setting Up an LDAP Repository

Setting up an LDAP repository on ATG involves the following steps:

1. Create the LDAP schema on your LDAP directory server. The methods for creating and modifying the LDAP schemas differ from server to server. Consult the documentation for your LDAP directory server.

   If your LDAP directory already exists, and you want to perform ID matching queries on the LDAP repository, make sure that LDAP entries include a property that corresponds to the ID property of the repository items. See LDAP Repository Queries.

2. Create the XML LDAP repository definition file for the LDAP repository to use. This XML template defines the item descriptors and repository item properties contained in your LDAP repository. It also describes the relationship of your LDAP directory entries to the item descriptors and repository items of the LDAP repository. See the LDAP Repository Architecture section, and especially the Item Descriptors and LDAP Object Classes subsection therein, for information about designing your LDAP repository components. See also the LDAP Repository Definition Tag Reference for full details of the XML tags used to create the LDAP repository definition file. Note that while the LDAP repository definition is similar in many ways to the SQL repository definition, the LDAP repository definition uses its own XML document type definition and syntax.

3. Configure the ATG LDAP repository components. See Configuring LDAP Repository Components.

4. Configure ATG so ATG's user profiling components point to the LDAP repository, rather than to an SQL profile repository. See the Setting Up an LDAP Profile Repository chapter in the ATG Personalization Programming Guide.

LDAP Directory Primer

This section briefly outlines the structure and contents of an LDAP directory, introduces the relevant terminology, and tries to summarize what you must know about LDAP in order to understand the LDAP repository. It includes the following topics:

- Hierarchical Tree Structure
- LDAP Data Representation
- Hierarchical Entry Types
- Directory Schema
- LDAP and JNDI
Hierarchical Tree Structure

An LDAP directory is organized into a tree of directory entries. Each directory entry is uniquely identified by its distinguished name (DN). The root point of the tree is represented by a special entry whose DN is called the directory suffix.

For example, a company directory for Quincy Funds might have a directory suffix of o=quincyfunds.com. Branching off the tree root, there may be entries for the various departments within the organization, such as ou=Finance, o=quincyfunds.com, ou=Marketing, o=quincyfunds.com, and so on. Under the organizational unit subtrees, there might be entries representing individual people, for example, uid=nat, ou=Finance, o=quincyfunds.com.

As you can see above, a DN consists of a series of comma-separated attribute name/value pairs. The hierarchy is represented right-to-left in a DN, with the right-most pair indicating the top of the hierarchy. For example, ou=Finance, o=quincyfunds.com is a child of o=quincyfunds.com. The left-most attribute name/value pair is called a relative distinguished name (RDN).

The examples in this section demonstrate some standard attribute names, such as:

- o for organization
- ou for organizational unit
- cn for common name

These standard attribute names are inherited from the X.500 standard, which preceded LDAP. Their use is not required, but is a good convention to follow when possible. Note that you can also define an organization like this:

dc=quincyfunds, dc=com

The directory tree may be highly branched, with the entire organizational hierarchy reflected in the tree structure, or it may be almost flat, depending on the needs of the organization. An example of an almost flat directory structure is one where all the people entries reside under the same organizational unit entry, such as ou=person, o=quincyfunds.com. There may also be organizational unit entries for storing other types of information, for example, ou=Groups, o=quincyfunds.com, ou=Customers, o=quincyfunds.com, ou=Devices, o=quincyfunds.com, and so on.

A directory may have more than one directory suffix. This typically comes into play with very large directories which are spread across multiple machines, extranets, and ISPs. For example, an ISP whose directory service needs to support multiple enterprises might have a separate directory suffix for each of the enterprises.

LDAP Data Representation

All data associated with an LDAP entry is contained in the entry's attributes. For example, the entry whose distinguished name is uid=nat, ou=person, o=quincyfunds.com might have the following attributes:
Many attributes in an LDAP directory can be multi-valued (such as the \texttt{cn}, \texttt{givenName}, and \texttt{objectClass} attributes in the example above).

One interesting point to note is that the attribute values comprising the entry's distinguished name do not necessarily have to correspond to the attribute values contained in the entry itself. For example, the entry above does not contain an \texttt{ou} attribute or an \texttt{o} attribute, even though the DN implies an \texttt{ou} value of \texttt{person} and an \texttt{o} value of \texttt{quincyfunds.com}. Even more confusing situations are possible (although, of course, not recommended by the directory providers), where the attribute is specified both in the DN and in the entry itself, but the two values differ.

For these kinds of cases, the thing to keep in mind is that the actual directory data is contained in the entry's attributes. The distinguished name is simply a name that can be used to uniquely identify the entry; it does not represent the actual attribute values. For example, when the directory is searched, it is not searched against the DN, but against the attribute values stored in the entries themselves.

Note however that you do use the DN to access a directory entry directly, without searching. Also, you must specify the DN when you create a new entry.

**Hierarchical Entry Types**

Each LDAP entry is associated with a type, or \texttt{object class}, which determines the attributes an entry is required to contain and allowed to contain. For example, the \texttt{person} object class has required attributes \texttt{cn} and \texttt{sn}, and optional attributes \texttt{description}, \texttt{seealso}, \texttt{telephonenumber}, and \texttt{userpassword}.

The entry's object class is stored in the entry itself, as the value of its \texttt{objectClass} attribute. When you create an LDAP entry, you must specify values for all the attributes required by the entry's object class, and you may specify values for any optional attributes.

The object class type can be a subtype of another object class. For example, the object class \texttt{organizationalPerson} is a subtype of the object class \texttt{person}. It happens to not add any required attributes, but it adds a number of optional ones, like \texttt{title}, \texttt{postaladdress}, and so on. The base (abstract) object class that every type inherits from is called \texttt{top}. Its single required attribute is \texttt{objectClass}.

Notice that the example entry in the **LDAP Data Representation** section above has three values for its \texttt{objectClass} attribute: \texttt{top}, \texttt{person}, and \texttt{organizationalPerson}. The first two values seem unnecessary, because they are both ancestors of the \texttt{organizationalPerson} type. However, they are required because not all directory servers support type inheritance.
The objectClass values in an entry do not all have to be each other’s ancestors, however. For example, one can create an entry that is both an organizationalPerson and a mailGroupMember, which itself inherits from top. In other words, multiple inheritance of types is allowed.

**Directory Schema**

The total set of object classes and attributes known to the LDAP directory is referred to as the directory schema. Each LDAP directory server comes with a standard schema that includes predefined object classes and attributes. Also, you can extend this standard schema to represent information unique to your enterprise.

For each object class, the schema contains information such as the names of the superior object classes from which this object class is derived, and the names of the required and optional attributes of the object class. For each of the attributes, the schema contains information about its syntax and whether the attribute is single- or multi-valued.

All LDAP directory implementations are expected to support the minimal default schema specified in RFC 2256. The tables below summarize those object classes and attributes in the default schema used by ATG’s LDAP repository. For the full list of object classes and attributes, please refer to the RFC.

**Sample LDAP Schema**

The examples in this chapter use the LDAP schema described in the following two tables. The inetorgPerson object class represents a person entry. This object class inherits from organizationalPerson but is not part of the default LDAP schema. It is specific to the Oracle (formerly Sun ONE) Directory Server. The inetorgPerson object class and its associated attributes are shown in italic in the tables that follow.

**Sample LDAP Object Classes**

<table>
<thead>
<tr>
<th>Name</th>
<th>Parent</th>
<th>Required Attributes</th>
<th>Optional Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>objectClass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>person</td>
<td>top</td>
<td>sn, cn</td>
<td>userPassword, telephoneNumber</td>
</tr>
<tr>
<td>organizationalPerson</td>
<td>person</td>
<td></td>
<td>title, employeeNumber, telephoneNumber, facsimileTelephoneNumber</td>
</tr>
<tr>
<td>inetorgPerson</td>
<td>organizationalPerson</td>
<td></td>
<td>mail, uid</td>
</tr>
</tbody>
</table>
Sample LDAP Entry Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Single Value?</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectClass</td>
<td>describes the kind of object an entry represents</td>
<td>false</td>
</tr>
<tr>
<td>cn</td>
<td>common name of an object, for example, person’s full name</td>
<td>false</td>
</tr>
<tr>
<td>sn</td>
<td>surname, or family name, of a person</td>
<td>false</td>
</tr>
<tr>
<td>o</td>
<td>name of an organization</td>
<td>false</td>
</tr>
<tr>
<td>ou</td>
<td>name of an organizational unit or department</td>
<td>false</td>
</tr>
<tr>
<td>givenName</td>
<td>person’s first name</td>
<td>false</td>
</tr>
<tr>
<td>userPassword</td>
<td>user password as an Octet String</td>
<td>false</td>
</tr>
<tr>
<td>title</td>
<td>person’s title in organizational context</td>
<td>false</td>
</tr>
<tr>
<td>telephoneNumber</td>
<td>telephone number</td>
<td>false</td>
</tr>
<tr>
<td>facsimileTelephoneNumber</td>
<td>fax number</td>
<td>false</td>
</tr>
<tr>
<td>uid</td>
<td>unique id</td>
<td>false</td>
</tr>
<tr>
<td>mail</td>
<td>e-mail address</td>
<td>false</td>
</tr>
<tr>
<td>employeeNumber</td>
<td>employee number</td>
<td>false</td>
</tr>
</tbody>
</table>

Notice that all attributes listed above are multi-valued. There are actually very few single-valued attributes in LDAP, for maximum flexibility.

LDAP and JNDI

The LDAP repository accesses data in the underlying LDAP directory using JNDI (Java Naming and Directory Interface). Oracle’s (formerly Sun’s) LDAP directory service provider, which implements the JNDI Service Provider Interface, is plugged in to allow ATG to use JNDI to access LDAP data. JNDI was designed so all the major JNDI operations are easily mapped onto the corresponding LDAP operations. Thus, JNDI provides a natural way to access LDAP data from Java applications.

Note that all the standard attributes are represented in Java as `String` or `byte[]` data types. That is, Oracle’s LDAP service provider for JNDI expects as input and returns as output all attribute values as one of these two types. When using the LDAP provider, you must explicitly configure all the attributes that should be treated as `byte[]`; the rest are treated as `String`. 
LDAP Sources

For more information about LDAP, consult the following:

- RFC 2254: The String Representation of LDAP Search Filters, describes the LDAP search filters, which are used by JNDI to perform directory searches. The LDAP repository’sQueryBuilder implementation constructs Query objects, which are essentially LDAP search filters.
- RFC 2256: A Summary of the X.500(96) User Schema for use with LDAPv3, default LDAP v3 schema that all LDAP directory servers are expected to support.
- Your LDAP server-specific documentation.

LDAP Repository Architecture

The ATG LDAP repository is an implementation of the Repository API that enables you to store and access profile data in an LDAP directory. Like other implementations of the Repository API, the LDAP repository uses Java components as representations of different elements of the data store. The LDAP directory corresponds to an LDAP repository. Each entry in the LDAP directory corresponds to a repository item, with the distinguished name of the LDAP entry serving as the repository ID of the corresponding repository item. The object classes of the LDAP directory map generally to item descriptors and Repository Views in the repository. These corresponding elements of the LDAP directory and the LDAP repository are described further in the following sections:

- LDAP Repository Items and Repository IDs
- Item Descriptors and LDAP Object Classes
- Item Descriptor Hierarchies and Inheritance
- Id and ObjectClasses Properties
- Additional Property Tag Attributes
- New Item Creation

You define the relationship between the LDAP directory schema and an ATG LDAP repository in an XML file called an LDAP repository definition file. This XML template needs to be located in the application’s configuration path. Unlike the SQL repository, the LDAP repository cannot use XML file combination to combine XML files in the configuration path that have the same name. Instead, you must use a single LDAP repository definition file in the configuration path.

This section introduces the principal features of the LDAP repository definition file. A complete reference to the repository definition file syntax is found in the LDAP Repository Definition Tag Reference section of this chapter.
LDAP Repository Items and Repository IDs

The LDAP repository uses the distinguished name of an LDAP entry as the repository ID of the repository item that corresponds to the LDAP entry. This is a natural choice for an ID, because each LDAP entry has a DN, and all DNs are unique. Also, the entry’s DN carries with it information about its location in the directory tree. This makes it very easy to retrieve items. No searching needs to be done; you simply access the entry directly with its DN.

The main question with using DNs for item IDs is what happens when a new item is created and added to the repository. If the ID is supplied for the item (for example, the DN `uid=nat,ou=Marketing,o=quincyfunds.com`), simply create the new entry with the specified DN in its parent context (in the example, `ou=Marketing,o=quincyfunds.com`). If the ID is not supplied, generate the DN before creating the directory entry.

The approach taken by the LDAP repository is to give the newly created repository item a temporary unique ID for the duration of the item’s existence as a RAM profile. When the time comes to add the item to the repository, generate a DN for the new LDAP entry, and assign this DN as the value of the persistent item’s ID. The DNs are generated with a pattern that you can configure. For example, one such pattern might be `uid=<login>,ou=Marketing,o=quincyfunds.com`, where `<login>` is the value of the item’s login attribute. If an item is created with the login value of `nat`, its DN is `uid=nat,ou=Marketing,o=quincyfunds.com`. See the New Item Creation section of this chapter for details on how to configure the way a new item’s DN is set.

Item Descriptors and LDAP Object Classes

The repository items in an LDAP repository are defined by one or more item descriptors. Each item descriptor defines the properties that a repository item can have. There is a natural mapping between a repository item descriptor and an object class of an LDAP directory. For example, a repository might include a user item descriptor that describes people, with properties such as login, password, firstName, lastName, phone, and so on. In the LDAP directory, this item descriptor corresponds to an object class such as inetorgPerson. The object class named inetorgPerson has attributes uid, userpassword, givenName, sn, and telephoneNumber, corresponding to the properties of the item descriptor. In other words, the user item descriptor is analogous to the inetorgPerson object class schema.

Mapping an LDAP Schema onto an Item Descriptor

In the most straightforward mapping between an ATG LDAP repository and an LDAP directory, the repository’s item descriptors have the same required and optional properties as the corresponding object classes of the LDAP directory. However, it is often desirable for the item descriptor to present a slightly different view of the LDAP directory schema. For example, the LDAP userpassword attribute is not required for the inetorgPerson object class. You might want to make the corresponding Profile property required, so when new profiles are created, the user password must be specified. Also, the inetorgPerson object class schema contains some attributes that might not be important to your Web application, such as seesAlso or x500UniqueIdentifier. Unnecessary attributes should not be exposed as properties in the repository item descriptor.

Similarly, although each attribute in the LDAP directory already has a schema associated with it, it is often desirable for the repository’s item descriptors to present a somewhat different view of the attribute schema. For example, the password attribute in the LDAP schema has the name userpassword, and a
Binary syntax (in other words, it is represented in Java as `byte[]`). In the ATG profile repository, the name of the Profile property should be `password`, and its type should be `String`. An LDAP attribute such as `age` is represented in Java as a `String`. The Profile's `age` property type should probably be an `Integer`. Also, you probably want to provide default values for some of the properties.

The LDAP repository lets you specify an item descriptor with all the capabilities previously mentioned. To demonstrate, the following is a portion of the sample XML template that describes the `user` item descriptor. Pieces of the item descriptor definition are omitted for clarity; these are discussed later on. You can see the complete Sample LDAP Repository Definition File later in this chapter. All tags you can use in the XML file that defines the LDAP repository are described in the LDAP Repository Definition Tag Reference in this chapter.

```xml
<!-- special properties -->
...

<!-- object classes -->
<object-class>top</object-class>
<object-class>person</object-class>
<object-class>organizationalPerson</object-class>
<object-class>inetorgPerson</object-class>

<!-- properties -->
<property name="login" ldap-name="uid" data-type="string" required="true"/>
<property name="password" ldap-name="userpassword" data-type="string" required="true"/>
<property name="lastName" ldap-name="sn" data-type="string" required="true"/>
<property name="firstName" ldap-name="givenName" data-type="string"/>
<property name="names" ldap-name="cn" data-type="string" multi="true" required="true"/>
<property name="email" ldap-name="mail" data-type="string"/>
<property name="phone" ldap-name="telephonenumber" data-type="string"/>
<property name="fax" ldap-name="facsimiletelephonenumber" data-type="string" default="(617) 555-1211"/>
<property name="department" ldap-name="ou" data-type="string"/>
<property name="title" data-type="string"/>
<property name="employeeNumber" data-type="long"/>

<!-- new item creation -->
...
```

The `object-class` tags specify all the object class values corresponding to the given item descriptor. If the object class has ancestor object classes, they must all be specified, as demonstrated above. The object class information is required so when a new item is created for the given item descriptor and added to the repository, the corresponding LDAP entry is created with the given object class values. Thus, for example,
if an item is created in the context of the user item descriptor, the new LDAP directory entry has
objectclass attribute values of top, person, organizationalPerson, and inetorgPerson.

The LDAP repository definition uses <property> tags to map Profile properties to LDAP attributes. Each
such <property>tag describes a property descriptor of its item descriptor. The <property>tags in the
example above demonstrate that:

- Repository item property names can be different from LDAP attribute names. For example, the lastName property in the item descriptor maps to the sn attribute in the
  LDAP directory schema. If the ldap-name tag attribute is not specified, the repository
  item property name and the LDAP attribute name are the same.

- Repository item property types can be different from JNDI service provider types. For example, userpassword is exposed as a binary type by Oracle's LDAP service
  provider, but is a String property in the repository; employeeNumber is a String in
  Oracle's LDAP service provider, but a Long in the repository.

- Repository item properties can have default values. For example, the fax property has
  a default value of (617) 555-1211.

Also, the user item descriptor exposes only those attributes that are meaningful, and promotes some of
the optional attributes into required ones. For example, the password attribute is optional in LDAP, but
required in the Profile repository item.

Although attributes such as givenName and sn are multi-valued in the LDAP directory, they are exposed
as single-valued properties in the repository. When getting the values for these properties, the LDAP
repository ignores all but one of the returned values. It is not specified which of the values are returned.
On the other hand, the LDAP repository item's names property is multi-valued, and corresponds to the
LDAP directory's multi-valued cn attribute; in this case, the attribute value is returned as a String array.

For all of this to work, the repository item descriptor must not violate the LDAP directory schema. For
example, because cn is a required attribute for the inetorgPerson class, one of the properties specified
in the item descriptor must map to the cn attribute, and it must be required. As another example, the
item descriptor cannot contain a property that does not correspond to an LDAP attribute. That is, the
ldap-name tag attribute value must be a legal LDAP attribute name. The LDAP repository does no
checking to ensure that the item descriptor conforms to the LDAP schema. If the schema is violated, a
runtime exception (an object schema violation) is thrown by JNDI.

### Item Descriptor Hierarchies and Inheritance

An LDAP repository may have any number of item descriptors. Because the LDAP directory might contain
any kind of data, the item descriptors may represent different kinds of items—people, computers, and
mailing lists. Each item descriptor simply maps to a particular set of object classes specified in the LDAP
schema.

The LDAP model also nicely supports hierarchies of item descriptors that map to object class subtypes.
For example, suppose the inetorgPerson object class has several subclasses, such as
engineeringPerson and salesPerson. The engineeringPerson class contains all the same attributes
as the inetorgPerson class, and adds a few, such as engineerType and currentProject. In the LDAP
repository, you can define an engineer item descriptor that inherits from the user item descriptor but
supports these additional attributes. The following example shows how a portion of an LDAP repository definition might describe such an engineer item descriptor:

```xml
<item-descriptor name="engineer" parent="user">

<!-- object classes (added to parent classes) -->
<object-class>engineeringPerson</object-class>

<!-- properties (added to parent properties) -->
<property name="engineerType" data-type="enumerated" default="products"
  description="Type of engineer: products or services">
  <option>products</option>
  <option>services</option>
</property>
<property name="currentProject" data-type="string"
  description="Project or product the engineer is currently working on"/>

<!-- child properties (override parent properties) -->
<child-property name="department" default="Engineering"/>

<!-- item creation (overrides parent behavior) -->
...

</item-descriptor>
```

The optional `parent` property of an `item-descriptor` specifies that the item descriptor inherits all parent's object classes and properties. Any additional `object-class` and `property` values are added to the list of the parent's object classes and properties.

You can also specify `<child-property>` tags to override any parent properties that have the same name. The only aspect of the parent property definition that can be overridden is the property's default value. The property's `data-type` and other attributes must stay the same. The example above demonstrates how the `<child-property>` tag can be used to assign the default value of `Engineering` to the parent's `department` property; the `salespeople` item descriptor might assign the default value of `Sales` to the same property.

**Id and ObjectClasses Properties**

In addition to the properties you specify, the LDAP repository creates two special properties for every item descriptor: the `id` attribute and the `objectClasses` attribute. Here is the relevant XML from the `user` item descriptor definition that was examined earlier:

```xml
<item-descriptor name="user">

<!-- special properties -->
<id-property name="id" in-ldap="true" ldap-name="dpsid"/>
<object-classes-property name="objectClasses" ldap-name="objectclass"/>
```

ATG Repository Guide
<!-- object classes -->
<object-class>top</object-class>
<object-class>person</object-class>
<object-class>organizationalPerson</object-class>
<object-class/inetorgPerson</object-class>
<object-class>dpsUser</object-class>

<!-- properties -->
...

<!-- new item creation -->
...
</item-descriptor>

The purpose of the `<id-property>` tag is to expose the repository ID of a repository item as an attribute (of type String). Thus, assuming the definition above, an item with repository ID `uid=nat,ou=Marketing,o=quincyfunds.com` has an LDAP attribute named `dpsid` with the same value. The attribute value does not need to be set by the user; it is set automatically by ATG. Note that the ID property is populated from the DN; you should not try to create the DN from the ID property.

The rest of the `<id-property>` definition above specifies whether the `id` property of the repository item maps to an actual LDAP attribute, and if so, the LDAP attribute's name. If the value of `in-ldap` is `false` (the default), the `id` attribute exists only as a property of the repository item, and does not exist as an attribute in the LDAP entry. In that case, when the item's attribute values are written out to the LDAP directory, the ID attribute value is ignored, because there is no equivalent for it in the directory entry. If the value of `in-ldap` is `true`, as above, the `ldap-name` tag attribute specifies the name of the LDAP attribute to which the `id` should be written. As usual, if `ldap-name` is not specified, it is assumed to be the same as `name`. Thus, with the example item descriptor, when an item with ID `uid=nat,ou=Marketing,o=quincyfunds.com` is created and added to the repository, the resulting LDAP entry has an attribute named `dpsid` with value `uid=nat,ou=Marketing,o=quincyfunds.com`.

Saving the ID attribute value in the LDAP entry makes it easier to perform ID matching repository queries, as discussed in the LDAP Repository Queries section in this chapter.

The `<object-classes-property>` tag is similar to `<id-property>`: it exposes the item's object class values as an attribute. The attribute's type is String[], which allows for a multi-valued attribute. For example, an item with a user item descriptor has an objectClasses attribute, whose value is an array with elements `top`, `person`, `organizationalPerson`, `inetorgPerson`, and `dpsUser`. The `dpsUser` object class supports the `dpsid` attribute, which allows incorporation of the repository ID as an attribute in the LDAP entry.

The `<id-property>` and `<object-classes-property>` tags are both required in a definition of a base item descriptor (that is, an item descriptor that does not have a parent); however, they are not allowed in child descriptor definition. The child item descriptors inherit the `id` and `objectClasses` properties from their parent.
Additional Property Tag Attributes

Each item descriptor includes \(<property>\) tags that define the properties of its repository items, how they correspond to the attributes of the corresponding LDAP entry, and how they are displayed in the ATG Control Center interface. Just as in the case of the SQL repository, the \(<property>\) tag has optional XML attributes such as \(required\), \(readable\), and \(hidden\).

Here is an example of a property definition that contains all the optional tag attributes:

```xml
<property name="department" ldap-name="ou"
    data-type="string"
    multi="false"
    display-name="Department"
    description="Department within the organization"
    default="unknown"
    required="false"
    readable="true"
    writable="false"
    queryable="true"
    hidden="false"
    expert="false"/>
```

See the LDAP Repository Definition Tag Reference in this chapter for full details.

For properties whose \(data-type\) attribute is set to \(enumerated\), use \(<option>\) tags to specify the property’s value choices, as in the \(engineerType\) attribute of the \(engineer\) item descriptor:

```xml
<property name="engineerType" data-type="enumerated" default="products"
    description="Type of engineer: products or services">
    <option>products</option>
    <option>services</option>
</property>
```

This approach is again inherited from the SQL repository definition file.

Just like the SQL repository, an LDAP repository’s \(<property>\) tags can have zero or more \(<attribute>\) child tags. These child tags allow you to associate arbitrary name/string value pairs with any attribute. The name/value pairs are added to the attribute's property descriptor via \(java.beans.FeatureDescriptor.setValue\), and can later be used by the application. Here is an example:

```xml
<property name="employeeNumber" data-type="string">
    <attribute name="unique" value="true"/>
</property>
```
You might use a descriptor like `unique` to specify that a property value can be assigned to only one repository item within the item type. This LDAP repository feature is similar to the feature described in User-Defined Property Types.

You can also specify a property editor class to use with a property with the `editor-class` attribute. For example, the following tag associates a special property editor with the password property:

```xml
<property name="password" ldap-name="userpassword" data-type="string" required="true" editor-class="atg.beans.PasswordPropertyEditor"/>
```

### New Item Creation

Finally, an item descriptor definition includes a `<new-items>` tag. This tag describes the item descriptor’s new item creation behavior. It specifies whether a new item of that item type can be created, and if so, describes how to create the DN (which is also the repository ID) for that item. This example completes the sample `user` and `engineer` item descriptor definitions:

```xml
<item-descriptor name="user">
  <!-- special properties -->
  ...
  <!-- object classes -->
  ...
  <!-- properties -->
  ...
  <!-- new item creation -->
  <new-items allowed="false"/>
</item-descriptor>

<item-descriptor name="engineer" parent="user">
  <!-- object classes (added to parent classes) -->
  ...
  <!-- properties (added to parent properties) -->
  ...
  <!-- child properties (override parent properties) -->
  ...
  <!-- new item creation (overrides parent behavior) -->
  <new-items parent-dn="ou=Engineering,o=quincyfunds.com" rdn-property="login"/>
</item-descriptor>
```
The `<new-items>` tag in the `user` descriptor indicates that this descriptor does not allow new items to be created. The `user` descriptor basically acts as an abstract class — it provides a base set of object classes and properties for children descriptors to build on, but it does not allow items with those object classes and properties to be instantiated.

The `engineer` descriptor, on the other hand, does allow new items to be created. The `new-items` tag specifies where the newly created items should be placed in the LDAP directory. The new item’s DN is constructed by appending the value of the `parent-dn` attribute to the RDN. The RDN is created from the value of the LDAP attribute that corresponds to the repository item property specified by the `rdn-property` XML attribute. For example, if a new item is created whose `login` property is `nat`, the corresponding RDN is `uid=nat` (because the Profile’s `login` property maps to the `uid` attribute in the LDAP directory), and the DN is `uid=nat, ou=Engineering, o=quincyfunds.com`.

If a child descriptor definition does not contain a `<new-items>` tag, it inherits the parent’s item creation behavior.

Repository Views in the LDAP Repository

In addition to supporting multiple item descriptors, the LDAP repository supports multiple Repository Views. For example, the `user` view might encompass all the people entries in the LDAP directory; the `engineer` and `salespeople` sub-views might contain only those people who are engineers and sales people, respectively. A `Devices` Repository View might span a completely separate space of device entries; and so on.

Repository View Definition

As demonstrated by the above example, there is typically a one-to-one correspondence between Repository Views and item descriptors. The `user` view is associated with the `user` item descriptor; the `engineer` view with the `engineer` item descriptor; and so on. In a sense, the item descriptor (in particular, its object classes) determines which items are contained by the view.

A Repository View’s contents can also be restricted to a particular location or set of locations within the directory tree. For example, one might want to specify that the `engineer` view contains only entries in the `ou=Engineering, o=quincyfunds.com` branch of the directory tree. Even if other items that satisfy the `engineer` item descriptor are encountered somewhere in the LDAP directory (perhaps for testing purposes), they are not considered to be part of the `engineer` view. The tree branches that comprise a Repository View are known as `search roots`, as they determine which parts of the directory tree are searched when a repository query is constructed on the view.

To summarize, the contents of each Repository View are determined by two factors: the object classes of its item descriptor, and its search roots. When a query is performed on the view, only those items that reside in one of the specified search roots and satisfy the view’s item descriptor are returned. At least one search root must always be specified, but it may well point to the directory suffix (i.e., the search root may span the entire directory tree).
LDAP Repository View Example

The following example shows how the user and engineer Repository Views are defined in an LDAP profile repository definition. The one-to-one correspondence between Repository Views and item descriptors in the XML template is enforced by making the item-descriptor tag a sub-tag of view. The view tag also contains the search-root tags, if any.

```xml
<view name="user" default="true">
  <!-- item descriptor -->
  <item-descriptor name="user">
    ...  
  </item-descriptor>
  <!-- search roots -->
  <search-root dn="o=quincyfunds.com"/>
</view>

<view name="engineer">
  <!-- item descriptor -->
  <item-descriptor name="engineer" parent="user">
    ...  
  </item-descriptor>
  <!-- search roots -->
  <search-root dn="ou=Engineering,o=quincyfunds.com" recursive="false" check-classes="true"/>
</view>
```

In this example, the user view spans all of o=quincyfunds.com, including ou=Engineering, o=quincyfunds.com, ou=Sales, o=quincyfunds.com, and so on, whereas the engineer view is restricted to ou=Engineering, o=quincyfunds.com.

Note the default attribute in the user view specification, which designates user as the default view name.

**Search Root Attributes**

There are a couple of optional attributes specified in the <search-root> tag of the engineer view above. The recursive attribute specifies whether the tree branch should be searched recursively; the default is true. You can set this to false if you want to include only the root’s immediate children, or if you know for sure that lower levels of the branch do not contain any relevant entries. This might be used for optimization purposes.

Similarly, in some cases you might be able to set the check-classes attribute to false to optimize search performance. In the default case, with check-classes set to true, when a repository query is
constructed, it is automatically augmented with the object class constraints, so items that do not satisfy the item descriptor are not returned by the search. For example, suppose you had a repository query (using the LDAP search filter syntax), such as:

\[(curren t\ Project=Quinc y)\]

If this query is applied to the \(ou=Engineering, o=quincyfunds.com\) search root of the engineer Repository View, the query becomes:

\[(\&(curren t\ Project=Quinc y)\n\(\ (object class=top)\n\(\ (object class=person)\n\(\ (object class=organizati onal\ Person)\n\(\ (object class=inetorgPerson)\n\(\ (object class=dpsUser)\n\(\ (object class=engineeringPerson))\n\]

If the value of check-classes is false for the search root, however, the query is left as is, and no object class checking is performed. Obviously, this optimization should only be turned on if you are absolutely sure that the search root contains only entries that satisfy the item descriptor.

**LDAP Repository Queries**

The LDAP repository supports most of the standard set of repository queries. See the Unsupported Queries in the LDAP Repository topic of this section for a list of exceptions. You can use targeting rules and services (described in the Creating Rules for Targeting Content and Setting Up Targeting Services chapters of the ATG Personalization Programming Guide) to query the LDAP repository. The LDAP query builder builds up an LDAP search filter string (as described in RFC 2254), which is used by the view to execute a series of searches (one for each search root) on the LDAP directory.

**ID Matching Queries**

The idMatching query is a special case. The LDAP search filter can only search for entries based on their attribute values. However, the LDAP repository uses the entry’s DN, rather than any attribute value, as its ID. Thus, ID matching queries cannot be constructed with search filters, unless the LDAP entry’s DN is also an LDAP attribute.

To implement ID matching queries, add an ID attribute to the LDAP entries, as described earlier in Id and ObjectClasses Properties. In this example, all user LDAP entries have an attribute called \(dpsid\), which is mapped to the repository item’s id attribute. The value of \(dpsid\) is automatically set to the DN when the item is first added to the repository. Because the ID can now be accessed as an attribute of an LDAP entry, full support for the ID matching query is provided in this case. Note, however, that directory entries that were not created by the repository must be manually modified to include a dpsid attribute, or they are not returned by the queries on the view.
If no ID attribute exists in the LDAP entries, the ID matching query is only supported as the top level query. That is, you can have a targeting rule that matches only items with specified IDs, but you cannot have a rule that matches items with specified IDs and satisfies some other criteria. The top level query is implemented by simply calling `Repository.getItems` with the specified IDs. No checking is done to verify that the resulting items are actually contained by the view. ATG does not check that they have the correct object classes, and are located inside one of the search roots.

Unsupported Queries in the LDAP Repository

The LDAP repository does not support queries of the following types:

- `includesAll`
- `elementAt`
- `indexOf`
- `count`
- `includesItem`
- `textSearch`
- `property` (when referring to sub-property (for example, `target="address.zip"`)
- `patternMatch` (with "ignore case" flag; that is, `containsIgnoreCase`, `startsWithIgnoreCase`, `endsWithIgnoreCase`)

Configuring LDAP Repository Components

When you set up the LDAP repository, you must configure the `InitialContextEnvironment` component to point to your LDAP server. You probably should also configure a number of other components that control the LDAP repository’s settings, performance, and caching behavior. The LDAP repository includes the following components:

<table>
<thead>
<tr>
<th>LDAP Repository Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/atg/adapter/idap/LDAPRepository</code></td>
<td>The repository of LDAP profiles.</td>
</tr>
<tr>
<td><code>/atg/adapter/idap/InitialContextPool</code></td>
<td>A resource pool (<code>JNDIInitialContextPool</code>) used to pool connections to the LDAP server.</td>
</tr>
<tr>
<td><code>/atg/adapter/idap/InitialContextEnvironment</code></td>
<td>Specifies the JNDI environment properties used to create a JNDI <code>InitialDirContext</code>.</td>
</tr>
<tr>
<td><code>/atg/adapter/idap/LDAPItemCache</code></td>
<td>An LRU cache that maps repository item IDs to persistent repository items.</td>
</tr>
<tr>
<td><code>/atg/adapter/idap/LDAPItemCacheAdapter</code></td>
<td>A component used by the <code>LDAPItemCache</code> to retrieve persistent repository items from the directory.</td>
</tr>
</tbody>
</table>
### LDAP Repository Component Description

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/atg/adapter/ldap/LDAPQueryCache</td>
<td>An LRU cache that maps repository search queries to the repository item IDs of the query results.</td>
</tr>
<tr>
<td>/atg/adapter/ldap/LDAPQueryCacheAdapter</td>
<td>A component used by the LDAPQueryCache to perform repository queries.</td>
</tr>
</tbody>
</table>

These LDAP repository components can be configured with the properties described below.

### /atg/adapter/ldap/LDAPRepository

This component is the repository of LDAP profiles.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$class</td>
<td>class name</td>
</tr>
<tr>
<td></td>
<td>Default: atg.adapter.ldap.LDAPRepository</td>
</tr>
<tr>
<td>cacheItemProperties</td>
<td>Should repository items cache their properties?</td>
</tr>
<tr>
<td></td>
<td>Default: true</td>
</tr>
<tr>
<td>cacheItems</td>
<td>Should the repository cache directory items?</td>
</tr>
<tr>
<td></td>
<td>Default: true</td>
</tr>
<tr>
<td>cacheQueries</td>
<td>Should Repository Views cache query results?</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
<tr>
<td>definitionFile</td>
<td>The location of the XML template in the application’s configuration path. Note that the LDAP repository uses XML file combination to combine XML files in the application configuration path that have the same name. See the Nucleus: Organizing JavaBean Components chapter of the ATG Programming Guide.</td>
</tr>
<tr>
<td></td>
<td>Default: /atg/adapter/ldap/ldapUserProfile.xml</td>
</tr>
<tr>
<td>idGenerator</td>
<td>The Nucleus address of the component that creates repository IDs for new repository items.</td>
</tr>
<tr>
<td></td>
<td>Default: /atg/dynamo/service/IdGenerator</td>
</tr>
<tr>
<td>initialContextPool</td>
<td>The Nucleus address of the JNDI InitialContextPool used to obtain InitialDirContext objects.</td>
</tr>
<tr>
<td></td>
<td>Default: /atg/adapter/ldap/InitialContextPool</td>
</tr>
</tbody>
</table>
**Property** | **Description**
--- | ---
itemCache | The Nucleus address of the repository item cache component. Default: \atg\adapter\ldap\LDAPItemCache
prefetchItemProperties | Should repository items prefetch their properties? If true, the first time any item property is accessed, all item property values are retrieved and cached. This value is used only if cacheItemProperties is set to true. Default: true
queryCache | The Nucleus address of the Query cache component. Default: \atg\adapter\ldap\LDAPQueryCache
repositoryName | Name of repository. Default: LDAP
shutdownDelay | How long (in seconds) to delay before shutting down. This value is used only if shutdownDynamoOnFatal is set to true. Default: 30
shutdownDynamoOnFatal | Should your application be shut down on fatal repository errors? Default: True
transactionManager | The Nucleus address of the Transaction Manager component. Default: \atg\dynamo\transaction\TransactionManager
XMLToolsFactory | The Nucleus address of the XMLToolsFactory component. Default: \atg\dynamo\service\xml\XMLToolsFactory

/\atg\adapter\ldap\InitialContextPool

This component is a \_\_JNDIInitialContextPool_ used to pool connections to the LDAP server. This component’s class extends \_\_atg\_\_service\_\_resourcepool\_\_ResourcePool_. See the Core Dynamo Services chapter of the ATG Programming Guide and the Javadoc for the ResourcePool class in the ATG API Reference for more information about the many properties available for configuring a connection pool.

Getting connections from a resource pool yields better performance than creating a new connection for each request that needs to access the LDAP server. The following properties are particular to the \_\_JNDIInitialContextPool_:

**Property** | **Description**
--- | ---
$\text{class}$ | class name
Default: \_\_atg\_\_service\_\_resourcepool\_\_JNDIInitialContextPool
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNDI Environment</td>
<td>The Nucleus address of the JNDI environment component to use when creating initial context objects. Default: InitialContextEnvironment</td>
</tr>
<tr>
<td>createDirContexts</td>
<td>Should InitialDirContext objects be created rather than InitialContext objects? Default: true</td>
</tr>
<tr>
<td>createMonitoredContexts</td>
<td>Should the resource pool InitialContext (or InitialDirContext) objects be wrapped in MonitoredContext (or MonitoredDirContext) objects? If monitored contexts are being created, any JNDI service provider errors which occur as a result of operations performed on the contexts are reported, and the associated resource pool objects are invalidated. Default: True</td>
</tr>
</tbody>
</table>

/atg/adapter/ldap/InitialContextEnvironment

This component specifies the JNDI environment properties used to create a JNDI InitialDirContext. You must configure this component to point to your LDAP directory server. Typically, you set the following properties (other than the class definition):

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$class</td>
<td>class name</td>
</tr>
<tr>
<td>Default:</td>
<td>atg.adapter.ldap.LDAPJNDIEnvironment</td>
</tr>
<tr>
<td>providerURL</td>
<td>URL of the LDAP server</td>
</tr>
<tr>
<td>Default:</td>
<td>ldap://localhost:389</td>
</tr>
<tr>
<td>securityAuthentication</td>
<td>Authentication mechanism for the provider to use. Some valid values are: Simple (default) Use weak authentication (cleartext password) none Use no authentication (anonymous). CRAM-MD5 Use the CRAM-MD5 (RFC-2195) SASL mechanism.</td>
</tr>
<tr>
<td></td>
<td>See securityAuthentication Property below for more information.</td>
</tr>
</tbody>
</table>
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>securityPrincipal</strong></td>
<td>The identity of the principal to be authenticated, in the form of a distinguished name.</td>
</tr>
<tr>
<td></td>
<td>Default: cn=ldapadmin</td>
</tr>
<tr>
<td><strong>securityCredentials</strong></td>
<td>The credentials of the principal to be authenticated</td>
</tr>
<tr>
<td></td>
<td>Default: ldapadmin</td>
</tr>
<tr>
<td><strong>otherProperties</strong></td>
<td>Any additional environment properties you might need to set. The value of the otherProperties property is one or more comma-separated property/value pairs. For example, you can set:</td>
</tr>
<tr>
<td></td>
<td>otherProperties = com.sun.jndi.ldap.someProperty=someValue</td>
</tr>
<tr>
<td></td>
<td>Default: null</td>
</tr>
</tbody>
</table>

### securityAuthentication Property

The `securityAuthentication` property must be set to match an appropriate type of security authentication for your LDAP server. For example, you can use the CRAM-MD5 setting only if you have configured your LDAP directory server appropriately. Note also that if you set this property to `none`, the LDAP server treats the LDAP repository as an anonymous client. Depending on how your LDAP server is configured, you may therefore be unable to create, modify, or delete LDAP directory entries through the LDAP repository.

### Other Environment Properties

Also, the `InitialContextEnvironment` component has the following properties, which correspond to environment properties of a JNDI context (as documented in the `javax.naming.Context` interface):

- `initialContextFactory`
- `objectFactories`
- `controlFactories`
- `stateFactories`
- `URLPkgPrefixes`
- `DNSURL`
- `authoritative`
- `batchSize`
- `referral`
- `securityProtocol`
- `language`

See the Javadoc for `javax.naming.Context` for more information about these properties.

Furthermore, the `InitialContextEnvironment` component has the following properties that apply to LDAP service providers in general or are specific to Oracle's (formerly Sun's) JNDI LDAP service provider:
LDAPVersion
binaryAttributes
connectControls
deleterDN
derefAliases
typesOnly
refSeparator
socketFactory
referralLimit
BERTrace
schemaBugs


/atg/adapter/ldap/LDAPItemCache
This component is an LRU cache that maps repository item IDs to persistent repository items.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$class</td>
<td>Class name</td>
</tr>
<tr>
<td></td>
<td>Default: atg.service.cache.Cache</td>
</tr>
<tr>
<td>cacheAdapter</td>
<td>The Nucleus address of the adapter that knows how to get objects not found in the cache</td>
</tr>
<tr>
<td></td>
<td>Default: /atg/adapter/ldap/LDAPItemCacheAdapter</td>
</tr>
<tr>
<td>maximumEntryLifetime</td>
<td>The maximum time in milliseconds an entry lives in the cache.</td>
</tr>
<tr>
<td></td>
<td>0: Cache nothing, always get objects from the cacheAdapter</td>
</tr>
<tr>
<td></td>
<td>-1: Cache entries never expire</td>
</tr>
<tr>
<td></td>
<td>Default: -1</td>
</tr>
<tr>
<td>maximumCacheEntries</td>
<td>The maximum number of entries in the cache.</td>
</tr>
<tr>
<td></td>
<td>0: Cache nothing, always get objects from the cacheAdapter</td>
</tr>
<tr>
<td></td>
<td>-1: Unlimited</td>
</tr>
<tr>
<td></td>
<td>Default: 500</td>
</tr>
</tbody>
</table>

/atg/adapter/ldap/LDAPItemCacheAdapter
This component is used by the LDAPItemCache to retrieve persistent repository items from the directory.
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$class</code></td>
<td>Class name</td>
</tr>
<tr>
<td></td>
<td>Default: <code>atg.adapter.ldap.LDAPItemCacheAdapter</code></td>
</tr>
<tr>
<td><code>repository</code></td>
<td>The Nucleus address of the <code>LDAPRepository</code> that contains the cache</td>
</tr>
<tr>
<td></td>
<td>Default: <code>/atg/adapter/ldap/LDAPRepository</code></td>
</tr>
</tbody>
</table>

### /atg/adapter/ldap/LDAPQueryCache

This component is an LRU cache that maps repository search queries to the repository item IDs of the query results.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$class</code></td>
<td>Class name</td>
</tr>
<tr>
<td></td>
<td>Default: <code>atg.service.cache.Cache</code></td>
</tr>
<tr>
<td><code>cacheAdapter</code></td>
<td>The Nucleus address of the adapter that knows how to get objects not found in the cache</td>
</tr>
<tr>
<td></td>
<td>Default: <code>/atg/adapter/ldap/LDAPQueryCacheAdapter</code></td>
</tr>
<tr>
<td><code>maximumEntryLifetime</code></td>
<td>The maximum time in milliseconds an entry lives in the cache.</td>
</tr>
<tr>
<td></td>
<td>0: Cache nothing, always get objects from the <code>cacheAdapter</code></td>
</tr>
<tr>
<td></td>
<td>-1: Cache entries never expire</td>
</tr>
<tr>
<td></td>
<td>Default: -1</td>
</tr>
<tr>
<td><code>maximumCacheEntries</code></td>
<td>The maximum number of entries in the cache.</td>
</tr>
<tr>
<td></td>
<td>0: Cache nothing, always get objects from the <code>cacheAdapter</code></td>
</tr>
<tr>
<td></td>
<td>-1: Unlimited</td>
</tr>
<tr>
<td></td>
<td>Default: 1000</td>
</tr>
</tbody>
</table>

### /atg/adapter/ldap/LDAPQueryCacheAdapter

This component is used by the `LDAPQueryCache` to perform repository queries.
### Property Description

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$class</code></td>
<td>Class name</td>
</tr>
<tr>
<td>Default:</td>
<td><code>atg.adapter.ldap.LDAPQueryCacheAdapter</code></td>
</tr>
<tr>
<td><code>repository</code></td>
<td>The Nucleus address of <code>LDAPRepository</code> that contains the cache</td>
</tr>
<tr>
<td>Default:</td>
<td><code>/atg/adapter/ldap/LDAPRepository</code></td>
</tr>
</tbody>
</table>

### LDAP Password Encryption

The `passwordHasher` property of the `/atg/userprofiling/PropertyManager` component points to a password hasher component that handles password encryption. By default, this property is set like this:

```
passwordHasher=/atg/dynamo/security/DigestPasswordHasher
```

Change this property to ensure consistency with the LDAP password encryption method you've chosen. For Oracle (formerly Sun ONE) Directory Servers, set the `passwordHasher` property like this:

```
passwordHasher=/atg/adapter/ldap/NDSPasswordHasher
```

The `NDSPasswordHasher` component supports SHA or no encryption. Set the `encryption` property of the `/atg/adapter/ldap/NDSPasswordHasher` to the appropriate value:

- `encryption=SHA`: use SHA password encryption
- `encryption=clearText`: disable password encryption

For LDAP servers other than Oracle Directory Server, you might need to create your own `PasswordHasher` implementation, if none of the `PasswordHasher` implementations included in the ATG platform meet your requirements.

See the Working with User Profiles chapter of the ATG Personalization Programming Guide for more information about configuring the `PropertyManager` component.

### LDAP Repository Definition Tag Reference

The LDAP repository definition file uses the XML tags described in this section. See also, at the end of this chapter:

- Sample LDAP Repository Definition File
- DTD for LDAP Repository Definition Files

The LDAP repository definition is similar in many ways to the SQL repository definition. The following differences apply:

- LDAP repository definition uses its own XML document type definition and syntax.
The LDAP repository cannot use XML file combination to combine XML files in the application configuration path that have the same name. Instead, you must use a single LDAP repository definition file in the application configuration path.

`<!DOCTYPE>`

**LDAP repository**

All SQL repository templates start with a `DOCTYPE` declaration that references this document type definition (DTD) file:

```xml
<ldap_1.0.dtd>
```

This DTD is installed within the `<ATG10dir>/DAS/lib/classes.jar` archive, but can be referenced with this URL:

```url
http://www.atg.com/dtds/gsa/ldap_1.0.dtd
```

For example:

```xml
<!DOCTYPE ldap-adapter-template PUBLIC "-//Art Technology Group, Inc.//DTD LDAP Adapter//EN" "http://www.atg.com/dtds/ldap/ldap_1.0.dtd">
```

`<ldap-adapter-template>`

```xml
<!ELEMENT ldap-adapter-template (header, view+)>
```

The `<ldap-adapter-template>` is the top-level tag in a repository definition file.

`<header>`

**LDAP repository**

```xml
<!ELEMENT header (name?, author*, version?)>
```

Parent: `<ldap-adapter-template>`

The `<header>` tag provides information that can help you create and modify a repository definition files.

`<view>`

```xml
<!ELEMENT view (item-descriptor, search-root*)>
```

Parent: `<ldap-adapter-template>`
A repository definition file must include one `<view>` tag for each `RepositoryView` in your repository.

**view attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The <code>RepositoryView</code> name: required and must be unique in the definition file.</td>
</tr>
<tr>
<td>default</td>
<td>Boolean, specifies whether this is the default view for repository items. Default: <code>false</code></td>
</tr>
</tbody>
</table>

**<item-descriptor>**

**LDAP repository**

```xml
<!ELEMENT item-descriptor (id-property | object-classes-property | object-class | property | child-property | new-items)>```

Parent: `<view>`

Each `RepositoryView` in the LDAP repository includes a single item descriptor, which is defined by an `<item-descriptor>` tag.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The item descriptor name: required and must be unique in the definition file.</td>
</tr>
<tr>
<td>parent</td>
<td>The item descriptor from which this item descriptor inherits.</td>
</tr>
</tbody>
</table>

**<id-property>**

```xml
<!ELEMENT id-property EMPTY>```

Parent: `<item-descriptor>`

The `<id-property>` tag defines the profile ID property in the `RepositoryItem` and the LDAP entry. The tag is always empty. For example:

```xml
<id-property name="id" i n-ldap="true" ldap-name="dpsid"/>```
The `<id-property>` tag is required in a definition of a base item descriptor (an item descriptor that does not have a parent) but is not allowed in a child item descriptor that inherits from a parent.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required, the ID property’s name in the <code>RepositoryItem</code>.</td>
</tr>
<tr>
<td>in-ldap</td>
<td>Boolean, specifies whether the ID property corresponds to a single LDAP attribute. Default: <code>false</code></td>
</tr>
<tr>
<td>ldap-name</td>
<td>The ID attribute’s name in the LDAP directory.                             Default: value of <code>name</code></td>
</tr>
<tr>
<td>display-name</td>
<td>The text identifying the ID property in the ATG Control Center. Default: value of <code>name</code></td>
</tr>
<tr>
<td>description</td>
<td>The description of the ID property displayed in the ATG Control Center. Default: value of <code>name</code></td>
</tr>
</tbody>
</table>

### `<object-classes-property>`

```xml
<ATG Repository Guide>

The `<object-classes-property>` tag exposes the object classes of an LDAP entry as a property of a `RepositoryItem`. This tag is always empty. For example:

```xml
<object-classes-property name="objectClasses" ldap-name="objectclass"/>
```

Like the `<id-property>` tag, the `<object-classes-property>` tag is required in a definition of a base item descriptor (an item descriptor that does not have a parent) but is not allowed in a child item descriptor that inherits from a parent. The property's type is `String[]`, a multi-valued String. For example, if an item descriptor definition has the `<object-classes-property>` tag in the preceding example and has the following object classes definition:

```xml
<object-class>top</object-class>
<object-class>person</object-class>
<object-class>organizationalPerson</object-class>
<object-class/inetorgPerson</object-class>
```

its repository items have the following `objectClasses` property:
objectClasses=top, person, organizationalPerson, inetorgPerson

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required, the name of the repository item property that stores the item’s LDAP object class values.</td>
</tr>
<tr>
<td>ldap-name</td>
<td>The property’s name in the LDAP directory. Default: value of name</td>
</tr>
<tr>
<td>displayName</td>
<td>The text identifying the object classes property in the ATG Control Center. Default: value of name</td>
</tr>
<tr>
<td>display-property</td>
<td>Specifies a property of this item descriptor that is used to represent items of this type in a user interface. For example, a profile item descriptor might set display-property to login. Then, each repository item is represented using the value of the item’s login property.</td>
</tr>
<tr>
<td>description</td>
<td>The description of the object classes property displayed in the ATG Control Center. Default: value of name</td>
</tr>
</tbody>
</table>

<object-class>

<!ELEMENT object-class (#PCDATA)>

Parent: <>item-descriptor>

The object-class tags specify all the object class values corresponding to the given item descriptor. If the object class has ancestor object classes, they must all be specified. For example:

<object-class>top</object-class>
<object-class>person</object-class>
<object-class>organizationalPerson</object-class>
<object-class>inetorgPerson</object-class>

The object class information is required in the item descriptor specification so when a new item is created for the given item descriptor and added to the repository, the corresponding LDAP entry can be created with the given object class values. Thus, for example, if an item is created in the context of the user item descriptor, the new LDAP directory entry has objectclass attribute values of top, person, organizationalPerson, and inetorgPerson.
**<property>**

*LDAP repository*

```xml
<!ELEMENT property (option*, attribute*)>
```

**Parent:** `<item-descriptor>`

Property tags define the properties of a repository item and map the repository item properties to LDAP entry attributes.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required, the property's name</td>
</tr>
<tr>
<td>ldap-name</td>
<td>The property's name in the LDAP directory.</td>
</tr>
<tr>
<td></td>
<td>Default: value of <code>name</code></td>
</tr>
<tr>
<td>data-type</td>
<td>Required, the property's Java data type.</td>
</tr>
<tr>
<td></td>
<td><code>String</code> <code>int</code> <code>byte</code> <code>bigstring</code> <code>short</code> <code>binary</code> <code>enumerated</code> <code>long</code> <code>date</code></td>
</tr>
<tr>
<td></td>
<td><code>boolean</code> <code>float</code> <code>timestamp</code> <code>double</code></td>
</tr>
<tr>
<td>multi</td>
<td>Is this a multi-valued property? If <code>true</code>, the type is an array.</td>
</tr>
<tr>
<td></td>
<td><code>Boolean</code></td>
</tr>
<tr>
<td></td>
<td>Default: <code>false</code></td>
</tr>
<tr>
<td>display-name</td>
<td>The text identifying the property in the ATG Control Center.</td>
</tr>
<tr>
<td></td>
<td>Default: value of <code>name</code></td>
</tr>
<tr>
<td>description</td>
<td>The description of the property displayed in the ATG Control Center.</td>
</tr>
<tr>
<td></td>
<td>Default: value of <code>name</code></td>
</tr>
<tr>
<td>default</td>
<td>A default value for the property, if another value is not specified when the profile is created</td>
</tr>
<tr>
<td>required</td>
<td><code>Boolean</code>. Default: <code>false</code></td>
</tr>
<tr>
<td>readable</td>
<td><code>Boolean</code>. Default: <code>false</code></td>
</tr>
<tr>
<td>writable</td>
<td><code>Boolean</code>. Default: <code>false</code></td>
</tr>
</tbody>
</table>
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queryable</td>
<td>Boolean. Note that non-queryable properties are not indexed in the ATG Control Center. Default: false</td>
</tr>
<tr>
<td>hidden</td>
<td>Boolean. Hidden properties are not displayed in the ATG Control Center. Default: false</td>
</tr>
<tr>
<td>expert</td>
<td>Boolean. Default: false</td>
</tr>
<tr>
<td>editor-class</td>
<td>Associates a property editor class with the property. See the JavaBeans specification for a description of PropertyEditors.</td>
</tr>
</tbody>
</table>

### LDAP Data-type Correspondences

The data-type attribute defines the data-type of a repository item property. The following table shows how the data-type attribute names correspond to Java object types.

<table>
<thead>
<tr>
<th>Data-type attribute value</th>
<th>Java object type</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String</td>
</tr>
<tr>
<td>big string</td>
<td>String</td>
</tr>
<tr>
<td>date</td>
<td>java.util.Date</td>
</tr>
<tr>
<td>timestamp</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>enumerated</td>
<td>String</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>binary</td>
<td>byte[]</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
</tbody>
</table>
<option>
LDAP repository
</option>

<!-- ELEMENT option (#PCDATA) -->

Parent: <property>

If a property's data-type property is set to enumerated, use <option> tags to indicate possible values of the enumerated properties. For example:

<property name="gender" data-type="enumerated">
  <option>male</option>
  <option>female</option>
</property>

<attribute>
LDAP repository
</attribute>

<!-- ELEMENT option (#PCDATA) -->

Parent: <property>

The <attribute> tag associates arbitrary name/string value pairs with a property or item type, which determine its behavior. The name/value pairs are added to the property descriptor via the setValue method of java.beans.FeatureDescriptor, and can later be used by the application.

For example:

<property name="employeeNumber" data-type="string">
  <attribute name="unique" value="true"/>
</property>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required, the name of the name/value pair.</td>
</tr>
<tr>
<td>value</td>
<td>Required, the value of the name/value pair.</td>
</tr>
</tbody>
</table>
<child-property/>

<!ELEMENT child-property EMPTY>

Parent: <item-descriptor>

If an item descriptor has a parent item-descriptor, use <child-property> tags to override inherited property values. The only aspect of the parent property definition that can be overridden is the property's default value. For example, given a parent item descriptor with the following property:

```
<item-descriptor name="user">
  ...
  <property name="department" default="Other"/>
  ...
</item-descriptor>
```

You can create a child property that overrides the default value of the department property:

```
<item-descriptor name="engineer" parent="user">
  <!-- object classes (added to parent classes) -->
  <object-class>engineeringPerson</object-class>
  <!-- properties (added to parent properties) -->
  ...
  <!-- child property (overrides parent properties) -->
  <child-property name="department" default="Engineering"/>
</item-descriptor>
```

See Item Descriptor Hierarchies and Inheritance in the LDAP Repository Architecture section of this chapter.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required, the attribute name, which is the same as the name of an attribute of the parent item descriptor.</td>
</tr>
<tr>
<td>default</td>
<td>Required, the default value for the attribute in the child item descriptor, overrides the default value of the corresponding attribute in the parent item descriptor.</td>
</tr>
</tbody>
</table>
The `<new-items>` tag describes how new items within the item descriptor are created and identified.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>allowed</code></td>
<td>Boolean. If <code>false</code>, no new items can be created in this item descriptor; the item descriptor acts like an abstract class. Default: <code>true</code></td>
</tr>
<tr>
<td><code>parent-dn</code></td>
<td>The distinguished name (DN) of the parent. The new item's DN is constructed by appending the value of <code>parent-dn</code> to the relative distinguished name, specified by <code>rdn-property</code>.</td>
</tr>
<tr>
<td><code>rdn-property</code></td>
<td>The name of the repository item property that specifies the relative distinguished name of a new item.</td>
</tr>
</tbody>
</table>

For example, given the following `<new-items>` tag:

```xml
<nw-items parent-dn="ou=Marketing, o=quincyfunds.com"
            rdn-property="login">
```

A new item whose `login` property is `nat` has a corresponding RDN of `uid=nat` (because the LDAP repository's `login` property maps to the `uid` attribute in the LDAP directory), and the DN is `uid=nat, ou=Marketing, o=quincyfunds.com`.

If a child descriptor definition does not contain a `<new-items>` tag, it simply inherits the parent's item creation behavior. See New Item Creation in the LDAP Repository Architecture section of this chapter.

A Repository View's contents can be restricted to a location or set of locations within the directory tree. For example, you might want to specify that the `engineer` view contains only entries in the `ou=Engineering, o=quincyfunds.com` branch of the directory tree. Even if other items that satisfy the `engineer` item descriptor are encountered somewhere in the LDAP directory (perhaps for testing.
purposes), they are not considered to be part of the engineer view. The tree branches that comprise a Repository View are called search roots, as they determine which parts of the directory tree are searched when a repository query is constructed on the view. The `<search-root>` tag is a child tag of the `<view>` tag that limits the Repository View to the specified roots of the LDAP directory tree.

When a query is performed on the view, only those items that reside in one of the specified search roots and satisfy the view’s item descriptor are returned. At least one search root must always be specified, but it may well point to the directory suffix (i.e., the search root may span the entire directory tree). See the Repository Views in the LDAP Repository section of this chapter.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dn</td>
<td>Required, the distinguished name (DN) of directory branches that can be part of the Repository View</td>
</tr>
<tr>
<td>recursive</td>
<td>Boolean, specifies whether the directory tree branch specified by the <code>dn</code> attribute should be searched recursively. Set to <code>false</code> if you want to include only the root’s immediate children, or if you know for sure that lower levels of the branch do not contain any relevant entries. This might be used for optimization purposes. Default: <code>true</code></td>
</tr>
<tr>
<td>check-classes</td>
<td>Boolean. If set to <code>true</code>, when a repository query is constructed, it is automatically augmented with the object class constraints, so items that do not satisfy the item descriptor are not returned by the search. If set to <code>false</code> for the search root, the query is left as is, and no object class checking is performed. This optimization should only be turned on if you are absolutely sure that the search root contains only entries that satisfy the item descriptor. Default: <code>true</code></td>
</tr>
</tbody>
</table>

**Sample LDAP Repository Definition File**

The following sample LDAP repository definition file defines a base item descriptor and view named user.

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE ldap-adapter-template PUBLIC "-//Art Technology Group, Inc.//DTD LDAP Adapter//EN" "http://www.atg.com/dtlds/ldap/ldap_1.0.dtd">
```
<header>
  <name>ldapUserProfile.xml</name>
  <author>ATG</author>
  <version>$Id: ldapUserProfile.xml,v 1.5 2000/06/23 00:16:14 nat Exp $</version>
</header>

<!-- user view -->
<view name="user" default="true">

<!-- item descriptor -->
<item-descriptor name="user" display-name="User" display-property="login">

<!-- special properties -->
  <id-property name="id" in-ldap="false"/>
  <object-classes-property name="objectClasses" ldap-name="objectclass"/>

<!-- object classes -->
  <object-class>top</object-class>
  <object-class>person</object-class>
  <object-class>organizationalPerson</object-class>
  <object-class/inetorgPerson</object-class>

<!-- properties -->
  <property name="login" ldap-name="uid" data-type="string" required="true">
    <attribute name="unique" value="true"/>
  </property>
  <property name="password" ldap-name="userpassword" data-type="string" required="true">
    <attribute name="unique" value="true"/>
  </property>
  <property name="fullName" ldap-name="cn" data-type="string" required="true"/>
  <property name="lastName" ldap-name="sn" data-type="string" required="true"/>
  <property name="firstName" ldap-name="givenName" data-type="string"/>
  <property name="email" ldap-name="mail" data-type="string"/>

<!-- item creation -->
  <new-items parent-dn="o=yourcompany.com" rdn-property="login"/>

</item-descriptor>

<!-- search roots -->
<search-root dn="o=yourcompany.com"/>
</view>
</ldap-adapter-template>
DTD for LDAP Repository Definition Files

This is the XML Document Type Definition for LDAP repository definition files. Do not modify this file. You can also view this file at:

http://www.atg.com/dtds/ldap/ldap_1.0.dtd

```xml
<?xml encoding="UTF-8"?>
<!--
====================================================================
ldap-adapter-template.dtd - document type for LDAP Adapter templates
@version $Id: //product/DAS/version/10.0.1/java/atg/dtds/ldap/ldap_1.0.dtd#1
$Change: 531151 $
====================================================================
-->

<!ENTITY % flag "(true | false)">

<!-- The whole template -->
<!ELEMENT ldap-adapter-template (header, view+)>

<!-- The header -->
<!ELEMENT header (name?, author*, version?)>

<!-- Name of template -->
<!ELEMENT name (#PCDATA)>

<!-- The author(s) -->
<!ELEMENT author (#PCDATA)>

<!-- Version string -->
<!ELEMENT version (#PCDATA)>

<!-- View(s) -->
<!ELEMENT view (item-descriptor, search-root*)>

<!ATTLIST view
 name CDATA #REQUIRED
 default %flag; "false"
>

<!-- Item descriptor(s) -->
<!ELEMENT item-descriptor (id-property | object-classes-property | object-class | property | child-property | new-items)*>

<!ATTLIST item-descriptor
 name CDATA #REQUIRED
 parent CDATA #IMPLIED
 display-name CDATA #IMPLIED
>
```
description | CDATA | #IMPLIED
hidden | %flag; "false"
expert | %flag; "false"
display-property | CDATA | #IMPLIED

<!-- Id property -->
<!ELEMENT id-property EMPTY>
<!ATTLIST id-property
    name CDATA #REQUIRED
    in-ldap %flag; "false"
    ldap-name CDATA #IMPLIED
    display-name CDATA #IMPLIED
    description CDATA #IMPLIED

<!-- Object classes property -->
<!ELEMENT object-classes-property EMPTY>
<!ATTLIST object-classes-property
    name CDATA #REQUIRED
    ldap-name CDATA #IMPLIED
    display-name CDATA #IMPLIED
    description CDATA #IMPLIED

<!-- Object class(es) -->
<!ELEMENT object-class (#PCDATA)>

<!-- Property(s) -->
<!ELEMENT property (option*, attribute*)>
<!ATTLIST property
    name CDATA #REQUIRED
    ldap-name CDATA #IMPLIED
    data-type CDATA #REQUIRED
    multi %flag; "false"
    display-name CDATA #IMPLIED
    description CDATA #IMPLIED
    default CDATA #IMPLIED
    required %flag; "false"
    readable %flag; "true"
    writable %flag; "true"
    queryable %flag; "true"
    hidden %flag; "false"
    expert %flag; "false"
    editor-class CDATA #IMPLIED

<!-- Options are possible values for enumerated properties -->
<!ELEMENT option (#PCDATA)>
<!-- Feature descriptor values -->
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    name CDATA #REQUIRED
    value CDATA #IMPLIED
    bean CDATA #IMPLIED>

<!-- Child property(s) -->
<!ELEMENT child-property EMPTY>
<!ATTLIST child-property
    name CDATA #REQUIRED
    default CDATA #REQUIRED>

<!-- Item creation -->
<!ELEMENT new-items EMPTY>
<!ATTLIST new-items
    allowed %flag; 'true'
    parent-dn CDATA #IMPLIED
    rdn-property CDATA #IMPLIED>

<!-- Search root(s) -->
<!ELEMENT search-root EMPTY>
<!ATTLIST search-root
    dn CDATA #REQUIRED
    recursive %flag; 'true'
    check-classes %flag; 'true'
Index

A
access rights
repository items, 306
ACL (Access Control Lists)
access rights, 317
syntax, 316
add-item tag, 166
alias derivation method, 76
atg.repository.content.ContentRepositoryItem, 237
atg.repository.NamedQueryView interface, 114
atg.repository.query.QueryDescriptorImpl, 115
atg.repository.QueryDescriptor interface, 115
atg.repository.RepositoryUtils, 168

B
BLOBs, 193

C
caches
restore item cache, 155
caches (composite repository), 289
caches (LDAP repository)
item cache, 349
query cache, 350
caches (SQL repository)
distributed cache invalidation, 162
caching (SQL repository), 127
configure caches, 150
disabling, 129
distributedJMS mode, 143
dump cache contents, 128, 154, 206
flush caches, 161
group properties, 101
isolation levels, 135
item cache, 128, 152, See also item cache
load, 155
load items, 206
metrics, 153
modes, 128
query cache, 128, 152, See also query cache
simple mode, 130
timeout for item caches, 152
timeout for query caches, 152
write locks, 135
cascade operations, 62
delete, 63
delete sequence, 64
element, 64
insert, 62
update, 63
CLOBs, 193
collectiveUnion derivation method, 77
composite repositories, 283
caching, 289
configuring, 285
definition files, 289
excluding properties, 286
item descriptors, 283
link methods, 286
link-via-id, 296
link-via-property, 296
non-serializable properties, 284
primary-item-descriptor-link, 295
property mapping, 285
transient properties, 284
CompositeRepository components
configuring, 288
configuration files, secured repositories. See secured
repositories, definition files
constraints
REFERENCES, 57
content repository
register, 228
content repository, SQL, 237
ContentHandler, 259

database column names, 58
database meta data, 120
data-type correspondences
LDAP repository properties to Java types, 357
SQL database to SQL repository, 192
date properties
SQL repository, 83
debugLevel property, 178
definition files
LDAP repository. See LDAP repository definition files
secured repositories. See secured repositories,
definition files
SQL repository. See SQL repository definition files
derivation methods
alias, 76
collectiveUnion, 77
firstNonNull, 74
firstWithAttribute, 74
firstWithLocale, 75
union, 76
derived properties, 70, 193
alias derivation method, 76
collectiveUnion derivation method, 77
derivation methods, 73
firstNonNull derivation method, 74
firstWithAttribute derivation method, 74
firstWithLocale derivation method, 75
override properties, 72
union derivation method, 76
distinguished names. See LDAP repositories, distinguished names
DNs. See LDAP repositories, distinguished names
Document Type Definition (DTD)
LDAP repository definition files, 363
Repository Loader manifest file, 262
secured repository definition files, 322, 323
DTD. See Document Type Definition (DTD)

E
enumerated properties
LDAP repository, 339
SQL repository, 79
EventServer components, 141
exceptions
secured repositories, 326

F
file combination, XML, 50
FileSystemMonitorScheduler
configuring, 251
FileSystemMonitorService, 252
configuring, 252
filterQuery property, 105
firstNonNull derivation method, 74
firstWithAttribute derivation method, 74
firstWithLocale derivation method, 75
full text search queries, 39

G
Generic SQL Adapter (GSA). See SQL repositories
GetRepositoryItem Web services, 275
GSAEventServer, 141
GSAPropertyDescriptor, 101

I
inheritance, item descriptor
LDAP repository, 336
SQL repository, 65
item cache, 128
timeout, 152
item caches. See caches
item descriptors, 18, 19
Dynamic Beans, 20
inheritance, LDAP repositories, 336
inheritance, SQL repositories, 65
LDAP repository, 334

J
java.sql.CallableStatement, 113
java.sql.ResultSet, 113
javax.naming.Context, 348
JNDI
accessing an LDAP directory, 332

L
last-modified properties
SQL repository, 84
LDAP (Lightweight Directory Access Protocol), 327, 328
data representation, 329
directory schema, 331
distinguished names, 329
distinguished names, 334
entries, 329
timeout, 152
entry attributes, 329
timeout, 152
entry types, 330
hierarchical tree structure, 329
JNDI, access through, 332
timeout, 152
object classes, 330
LDAP repositories, 327
LDAP repository
attribute tags, 339
timeout, 152
configuring components, 344
creating new repository items, 340
timeout, 152
directory schema sample, 331
timeout, 152
distinguished names, 334
timeout, 152
enumerated properties, 339
timeout, 152
inheritance, item descriptor, 336
timeout, 152
InitialContextEnvironment component, 344, 347
InitialContextPool component, 346
timeout, 152
queries, 349
repository IDs, 334, 337
item cache, 349
repository items, 334
item descriptors, 334, 336
Repository API implementation, 333
LDAPItemCacheAdapter component, 349
repository views, 341
LDAPQueryCacheAdapter component, 350
search roots, 341, 342
LDAPRepository component, 345
security authentication, 348
query cache, 350
LDAP repository definition files, 351
attribute tag, 358
child-property tag, 359
data-type correspondences, 357
id-property tag, 353
item-descriptor tag, 353
ldap-adapter-template tag, 352
new-items tag, 360
object-class tag, 355
object-classes-property tag, 354
option tag, 358
property tag, 356
sample, 361
search-root tag, 360
view tag, 353
Lightweight Directory Access Protocol. See LDAP
LoaderManager, 253
configuring, 254
Lock Managers, 131

M
manifests
Repository Loader, 262
Repository Loader tags, 263
meta data
database, 120

N
named queries, 109
null reference
remove, 63
null values
SQL repository, 84
SQL repository queries, 119

O
Oracle
ConText full text search engine, 39
outer joins
SQL repository queries, 119
override property
in derived properties, 72

P
parameterized queries, 106
password encryption
LDAP repository, 351
password hashing. See password encryption
PerformRQLCountQuery Web services, 279
PerformRQLQuery Web services, 277
PropertiesChangedEvent, 25
property
class for item type, 101
property mapping
Composite repositories, 285
property-type attributes, 96
PublishingFileRepository
configure TypeMapping components, 268

Q
queries. See also Repository Query Language (RQL)
ATG Control Center, 34
bypassing RQL, 105
composite repository, 288
LDAP repository, 343
named, 109
null values in NOT query, 119
parameterized, 106
QueryBuilder interface, 29
QueryOptions, 30
Repository API, 29
Repository API example, 31
wildcards, 118
queries, unsupported
LDAP repository, 344
SQL repository, 121
query cache, 128
timeout, 152
query caches. See caches
parameterized queries, 108

R
REFERENCES constraints, 57
remove-item tag, 168
repositories
data-types, 20
item descriptors. See item descriptors
LDAP. See LDAP repositories
mutable, 21, 23
queries. See queries
Repository API summary, 21
secured. See secured repositories
SQL. See SQL repositories
repository filter, 103
create, 104
repository IDs, 19
repository item
add, 166
clean, 26
delete, 168
fetch properties, 101
IDs, 19
LDAP repository, 334
PropertiesChangedEvent, 25
property-type attributes, 96
removing references, 168
update, 168
updating, 24
user-defined properties, 95
repository items, 18, 23
Repository Loader, 237, 263
administration UI, 260
client, 260
configure TypeMapping components for PublishingFileRepository, 268
ErrorPolicy, 255
example, 268
manifest tags, 263
manifests, 262
start, 263
targeters, 267
use after site initialization, 266
VersionedLoaderEventListener, 264
Repository Query Language (RQL), 34
ALL queries, 38
count queries, 35
examples, 42
full text search queries, 39
grammar definition, 42
ID-based queries, 39
INCLUDES ITEM queries, 38
IS NULL queries, 38
limiting result sets, 40
logic operators (AND, OR, NOT), 37
multi-valued queries, 37
ORDER BY directives, 40
ordering query result sets, 40
parameterized queries, 41
pattern match queries, 36
property of property queries, 37
RANGE directive, 40
text comparison queries, 36
Repository Views
LDAP repository, 341
RepositoryItem, 18
RepositoryUtils. See atg.repository.RepositoryUtils
required properties
SQL repository, 82
RL module. See Repository Loader
RLClient, 260
hints file, 261
RQL. See Repository Query Language (RQL)
RQL filters. See repository filter
rql-filter tag, 104
rqlFilterString property, 105
secured repositories, 305
access rights, 306
ACL property, 308, 321
ACLs, 316
configuration files. See secured repositories, definition files
files, See definition files
creating, 307
definition files, 310, 314, 318
document type definition, 322, 323
dependencies, 312
exceptions, 326
limitations, 307
owner-property, 308, 320
secured repository adapter components, 309
secured repository adapter components
configuring, 309
security
repositories. See secured repositories
security authentication
LDAP repository, 348
serializable repository items
composite repositories, 284
ServerLockManagers, 131
session backup
repository items, 77
simple caching, 130
SQL content repository, 237
SQL repositories
distributed cache invalidation, 162
SQL repository, 45
<add-item> tag, 166
architecture, 47
cache groups, 101
locking. See caching (SQL repository)
cascading operations, 62
closet repository item, 26
configure component, 228
Content window, 228
database column names, 58
deployLevel property, 178
derived properties, 70, 193
defined properties, 79
did properties, 53
didSpaces, 56
document owner, 120
large database columns, 102
null values in NOT query, 119
outer joins, 119
property-type attributes, 96
queries, unsupported, 121
register, 228
remove references to items, 168
rql-filter tag, 104
session backup, 77
set up, 45
streams, 102
tag attribute, 166
transactions, 47
uiqueryable attribute, 92
uiwritable attribute, 92
user-defined properties, 95
wildcards in queries, 118
SQL repository definition files, 47
SqlPassThroughQuery, 105
SqlPassThroughQuery, 105
startSQLRepositoryQuery, 105
arguments, 172
syntax, 171

table ownership, 120
tag attribute, 166
targeters
import with Repository Loader, 267
timeout
  SQL repository caches, 152
timestamp properties
  SQL repository, 83
transactions
  repositories, 47
transient properties, 91
  composite repositories, 284
TypeMapper, 256
  configuring, 256
TypeMapping, 256
  configuring, 257

U
  uinamequeryable attribute, 92
  uinwritable attribute, 92
  union derivation method, 76
  unique properties
    SQL repository, 83

unsupported queries. See queries, unsupported
update-item tag, 168
User Authority components, 317

V
  versioned repositories
    import assets, 175
  VersionedLoaderEventListener, 264
    properties, 264

W
  Web services, 275
    GetRepositoryItem, 275
    PerformRQLCountQuery, 279
    PerformRQLQuery, 277
  security policies, 281
  wildcard characters
    SQL repository queries, 118

X
  XML file combination, 50
  Xml2RepositoryContentHandler, 259
    configuring, 259