BEA Liquid Data for WebLogic®

Concepts Guide

Version 8.5
Document Date: June 2005
Revised: June 2005

Copyright

Restricted Rights Legend

Trademarks or Service Marks

July 29, 2005 3:59 pm

Contents

5. Designing Data Services

6. Using Service Data Objects (SDO)

7. Performance and Caching

8. Securing Data

CHAPTER 1

Introducing Liquid Data

This chapter provides an overview of BEA Liquid Data for WebLogic®. It covers the following topics:

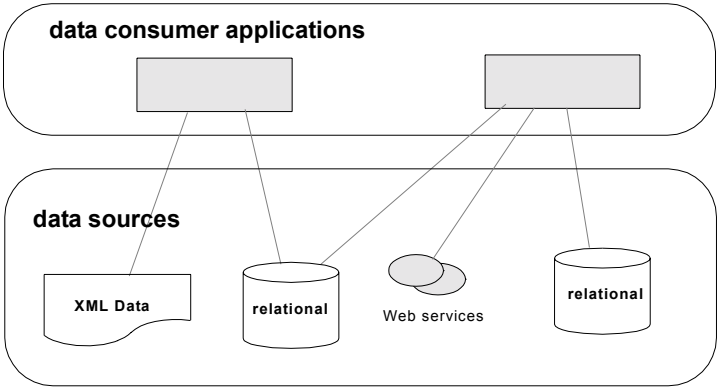
The Cost of Information Fragmentation

Information resources are often the most important assets an enterprise holds. Information can make or break an enterprise's ability to execute on its essential functions—from day-to-day operations to the marketing and delivery of products or services to strategic planning. The enterprise must be able to access, interpret, and use information as efficiently as possible.

Unfortunately, as enterprises evolve, their data resources tend to become increasingly fragmented. Corporate acquisitions and mergers, new technologies, and changing business strategies can all contribute to a state of data segregation, in which pockets of information become isolated by physical

and technical boundaries. The information comes to reside in a variety of places, in databases and files, within applications, or even outside the organization, supplied by business partners, vendors, and clients as Web services. Figure 1-1 illustrates this fragmentation of data.

Figure 1-1 Direct Data Access Applications



As a consequence, the information becomes effectively lost—inaccessible to developers and, in turn, the customers, employees, and decision makers who need it. Developers who do attempt to use the information need to know a daunting variety of data access mechanisms and APIs. The applications are difficult to create and maintain. Their source code tends to be dominated by data management logic, making them intolerant to changes in the data layer.

When changes do occur, the applications must be revised and retested, because their business logic is embedded with data access code. In the worst case, an application must be significantly modified and redeployed as a result of the changes to the data source layer.

When it comes to developing applications, the costs of information fragmentation are quantifiable and significant. By many estimates, up to 70% of the time required to develop an enterprise application goes into data programming.

Other costs may be more difficult to gauge but are certainly as significant—from hampered decision making and poor responsiveness to lost information resources.

What is BEA Liquid Data for WebLogic?

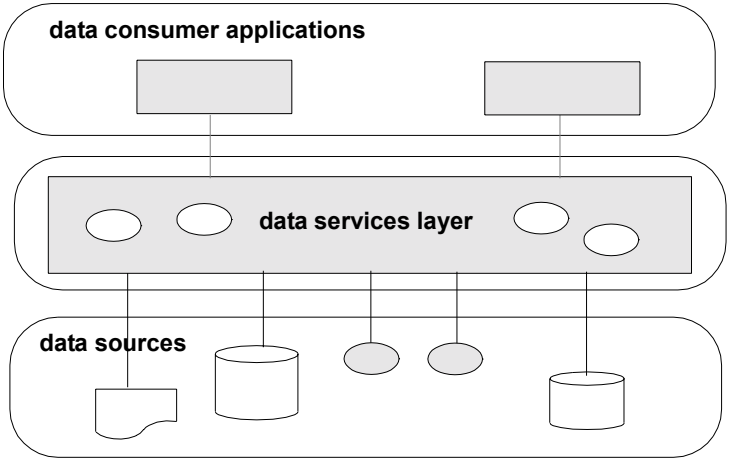
Liquid Data for WebLogic is the WebLogic Platform component for developing and deploying integrated data services. Data services give consumers an easy-to-use, uniform model for accessing heterogeneous, distributed data.

Liquid Data significantly simplifies the task of creating and deploying reusable data services. It has tools and frameworks for rapidly generating data services based on existing data sources and for creating logical data services based on those sources.

From the perspective of the data consumer, Liquid Data presents a sensible, uniform data model for getting and using information. The application simply instantiates a data service and retrieves or updates its data by calling one of the data service's public functions. The data may come from legacy applications, relational databases, Web services, delimited files, XML files, or any source from which a Java application can extract useful data. These data services can be viewed as a layer integrated between the data consumer and the data sources as illustrated in Figure 1-2.

Most data in an organization exists as queryable data or application data. Liquid Data lets you use all data together to compose a logical data model for the organization. The data model represents the business entities relevant to the organization, such as customers, orders, or products.

Figure 1-2 Data Integration Layer Between Data Users and Data Sources



Liquid Data is a virtual data layer in the sense that, except for caching, data is not kept in any Liquid Data or WebLogic component. Instead, data services dynamically retrieve data from the physical data

sources. Dynamic access ensures that applications have access to current (that is, real time) information.

A data service represents a unit of information in the data model. Like a Web service, it exposes a public interface in the form of one or more accessor functions. A data service accessor function typically returns data in the form of the data service's datatype. It uses XML and XQuery to acquire, transform, and aggregate data from external sources.

The data is poured into the XML data shape defined as the return type for the data service. Because an individual data service represents a relatively small unit of information, response time from the client's perspective is minimized. Other features, such as caching and query optimization, help to ensure the optimal performance of the data services layer.

Liquid Data insulates data consumers from the complexity of disparate data sources. It encapsulates the details of data integration logic and gives consumers a sensible, coherent model for accessing and updating data. Data access and business or presentation logic are effectively decoupled, resulting in applications that are easier to develop and maintain, without data access plumbing code.

As it does for reading data, Liquid Data gives client applications a unified interface for updating data. Liquid Data allows client applications to modify and update data from heterogeneous, distributed sources as if it were a single entity. The complexity of propagating changes to diverse data sources is handled by Liquid Data.

From the data service implementor's point of view, the task of building a library of update-capable data services is considerably eased by the Liquid Data update framework. For relational sources, Liquid Data can propagate changes to the data source automatically. For other sources or to customize relational updates, you can use the Liquid Data update framework artifacts and APIs to quickly implement customized, update-capable services. Either way, data consumers are isolated from the details of updating the data sources.

Liquid Data and a Service-Oriented Architecture (SOA)

A Service-Oriented Architecture is an architectural style for an IT environment. In this architecture, business processes are delivered as a set of loosely bound services. Web services provide an implementation of SOA. A web service is a modular, self-describing component that can be used in a platform-independent way.

Traditionally, applications are built as monolithic units. They impose platform dependencies on users and are not easily adaptable to new types of interactions. Web services break these applications into multiple reusable components that can interact with other independent components and services in meaningful ways. For example, instead of having a single financial application, an organization may deploy one or more services that expose the business activities of the application as self-contained,

callable processes. That way other systems (such as Human Resource services or applications) can easily leverage financial computing resources. Thus, SOA breaks down barriers between applications. With SOA the IT infrastructure operates in an integrated, organic fashion like a virtual application that spans the organization.

Viewed as a whole, a data service deployment constitutes a data abstraction layer between data users and sources. It normalizes diverse data and exposes a uniform, well-defined interface that data consumers can use to access data. Client developers do not have to know how to connect to a data source, what its native format is, or even where it comes from, whether from one or many underlying sources. The application can simply call a public function of a data service.

SOA is considered a loosely coupled—not completely decoupled—architecture because, while the service provider has no knowledge of the interfaces or business concerns of its consumers, the consumers do include explicit references to the service provider interface, in the form of service calls.

The benefits of loose coupling include simplified data access, reusability, and adaptability. The effect of changes to the data source are localized, requiring only relatively few changes in the data services layer rather than in every application that uses the data source. New services can be exposed and used without requiring extensive changes to existing applications. The result is a data integration layer that is highly adaptive and change tolerant.

Liquid Data extends SOA to the realm of data. With Liquid Data, you can expose information as data services. A data services represents a self contained business entity—such as a customer, product, or order—that is reusable across the organization.

Simply put, Liquid Data disentangles the data provider from the applications that use the data.

Other benefits of Liquid Data include:

- **Leverage domain resources**. Information "locked away" in legacy systems can be made available to application developers. Applications, and their users, benefit by being able to access and update the entire range of information resources of an enterprise.
- **Dramatically reduce coding**. With a high-level query language and visual development tools for service implementation, Liquid Data makes it easy to implement a data services layer that lets developers use simple function calls instead of tedious custom code. Data services employ a SQL-like language called the XML Query Language (XQuery).
- **Consolidate data management**. Liquid Data provides a single point for applying security and caching policies.
- **Enable uniform service governance**. Service-level agreements can be verified and documented.

Modeling the Enterprise

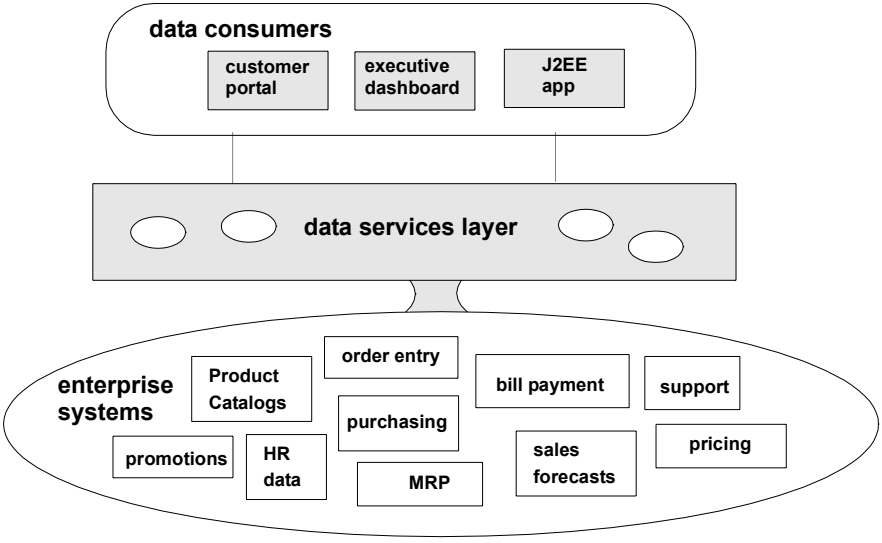
Liquid Data enables you to capture and deploy a data model specific for the needs of your organization. A data model typically organizes data in a way that represents business entities. Similarly, in your implementation, you can define the data model that makes sense for your organization. Business entities common to most organizations include, for example, products, orders and customers. A Customer object can have all of the attributes that are relevant to a customer in the deployed environment, such as contact information, shipping preferences, payment information, and so on. The data model can also specify security, caching, and other policies.

Instead of having to contend with multiple sources to acquire the information relevant to a customer, developers can simply use a Customer object from a data service. The policies relevant to the data are applied to the data consistently, without requiring developers to implement them in the various applications they develop.

To help you get from a complex, distributed physical data landscape into a sensible data model, Liquid Data supports a visual, model-driven approach to developing data services. Modeling provides a graphical representation of the data resources in your environment, providing a bird's eye view of a large system, or giving you a way to break a larger problem into smaller pieces.

The result is real-time access to externally persisted data through a logical data model (as illustrated in Figure 1-3) while providing a single point for applying policies, such as security and caching, on information used across applications and services.

Figure 1-3 Logical Data Model



In SOA, services are conventionally thought of in terms of processes. However, most SOA initiatives actually start with data. By its nature, data has broader, more generalized usage than typically does a business process. Data consumers can use the same piece of data in different ways. Once you have developed the data model for your enterprise data, the data types, as encapsulated by data services, can be reused by numerous data consuming applications.

The data model represented by a Liquid Data deployment is captured by metadata. In Liquid Data, metadata serves several purposes:

- Developers can use metadata to determine what data services are available, what functions they provide, where data comes from, and much more.
- Administrators can use metadata for impact analysis, that is, to determine how changes to the underlying data layer affect the data services layer. When changes occur, as they inevitably do, tools for synchronizing source metadata ease the difficulty of applying changes to data services.

The metadata browser, a component of the Liquid Data administration console, provides an easy-to interface for searching and browsing through metadata.

Implementing Data Services

A data service encapsulates the logic for normalizing, transforming, and integrating disparate data. A client uses a data service by calling one of its public functions. A data service can have any number of public functions, each of which returns data in the form of the data shape described by the XML Schema associated with the data service.

The logic encapsulated in a data service is in the form of queries written in the XML Query (XQuery) language. XQuery, based on an emerging standard specification, is a SQL-like language specifically intended for working with structured, XML data. As a declarative language, XQuery lends itself to compiler-based optimization techniques, alleviating data service developers from having to learn extensive rules and techniques for writing optimized XQuery functions.

Those familiar with SQL should find XQuery easy to learn. They have many features in common. Liquid Data provides an XQuery editor that allows you to get started with XQuery by using an easy-to-use, intuitive graphical tool.

Data Consumers

Once the Liquid Data application has been deployed to a WebLogic Server, clients can use it to access real-time data. Liquid Data supports a number of different types of data clients with these mechanisms:

- Java
- Web services
- Liquid Data Control
- JDBC/SQL

Liquid Data supports a service-oriented approach to data access through Java interfaces (the mediator API and the Liquid Data Workshop control) or through Web services that act as wrappers for the data services.

As an additional option, the Liquid Data JDBC driver gives SQL clients (such as reporting and database tools) and JDBC applications a traditional, database-oriented view of the data layer. To users of the JDBC driver, the set of data served by Liquid Data appears as a single virtual database. Note that, given the two-dimensional view of data inherent in SQL, SQL access is supported only for data services that provide a flat view of data.

Life Cycle of a Liquid Data Development Project

The following are the basic steps for developing, deploying, and maintaining a Liquid Data implementation:

1. **Establish project requirements.** This entails identifying the data resources that are available, business logic requirements, security needs, access information, and caching requirements.
2. **Import metadata from available sources.** Once you have chosen the data resources you want to include in your implementation, you can define them as data sources by creating data source profiles for them. Importing metadata for the sources automatically creates the physical data services that are the building blocks of your data services layer and stores the connection information for the source.
3. **Model the data access layer.** A model is a graphical representation of the data services in your environment and the relationships between them. Modeling helps you plan and document your data services implementation. Data services can be created and modified directly from a model diagram.
4. **Implement logical data services.** Logical data services provide a sensible view of the information resources that are available. They can transform, filter, and aggregate data, as well as prescribe constraints on the data, security, caching and other properties. For details see "Creating Data Services" in the *Building Queries and Data Views*.
5. **Identify update requirements.** For relational sources, update capabilities are automatically available through SDO. For other sources and for situations where you need to implement custom transaction rollback logic or other processes, you can use Java to customize the update routines provided with Liquid Data. Test your update logic and ability to deploy the application.
6. **Deploy the Liquid Data services**. For more information, see "Deploying Liquid Data Applications" in the *Administration Guide*.
7. **Configure cache and security settings**.
8. **Create the Liquid Data client applications**.
9. **Maintain and monitor the data services layer**. Liquid Data includes tools for evaluating the impact of changes to the data source layer and for synchronizing data services with the modifications.

Roles in Liquid Data Development

Liquid Data facilitates a team-based approach to application development. Instead of developers having to understand how to connect to and use numerous data sources—along with the business or

presentation logic that needs to be implemented once the data is acquired—development tasks can be divided in ways that naturally correspond to the roles that often exist in an organization:

- *Data Architects* know about the desired business entities to be created and the data sources that are required. They have a deep understanding of the data, underlying schema, and relationships across the various data sources. They create the schemas and Data Services and update logic used by the Application Developers.
- *Application Developers* create client applications that use the services of Liquid Data to access updateable, real-time information.
- *System Administrators* for a Liquid Data implementation perform the following types of tasks:
  - Deploy Liquid Data in WebLogic domains.
  - Configure access to data sources.
  - Set up and configure the Liquid Data caching.
  - Implement security using the WebLogic Server security features, such as configuring users, groups, and defining security roles.
  - Monitor Liquid Data operation, tune performance, and provide support.

CHAPTER 2

Architecture

This chapter describes the architecture of BEA Liquid Data for WebLogic. The following topics are covered:

- How Liquid Data Fits in the WebLogic Platform
- Liquid Data Architecture
- Liquid Data Components

How Liquid Data Fits in the WebLogic Platform

Liquid Data services can be developed as a Workshop application or as a project within another type of Workshop application, such as a portal, web application, or business process. The runtime platform for Liquid Data is WebLogic Server. Liquid Data can take full advantage of the services provided by WebLogic Server, such as scalability and clustering, and the full array of J2EE features and services. It operates seamlessly with other applications running in a WebLogic Server.

Client applications that want to use Liquid Data services only need to establish an initial context to the WebLogic Server where Liquid Data is deployed. The initial context, a JNDI mechanism for identifying a resource on a network, is similar to a database connection in that it identifies the location of a server and includes any required connection parameters, such as user names and passwords.

Once the initial context is established, the client can instantiate data services and use them to get and update information.

For complete information on BEA WebLogic Server, see the WebLogic Server documentation at the following URL:

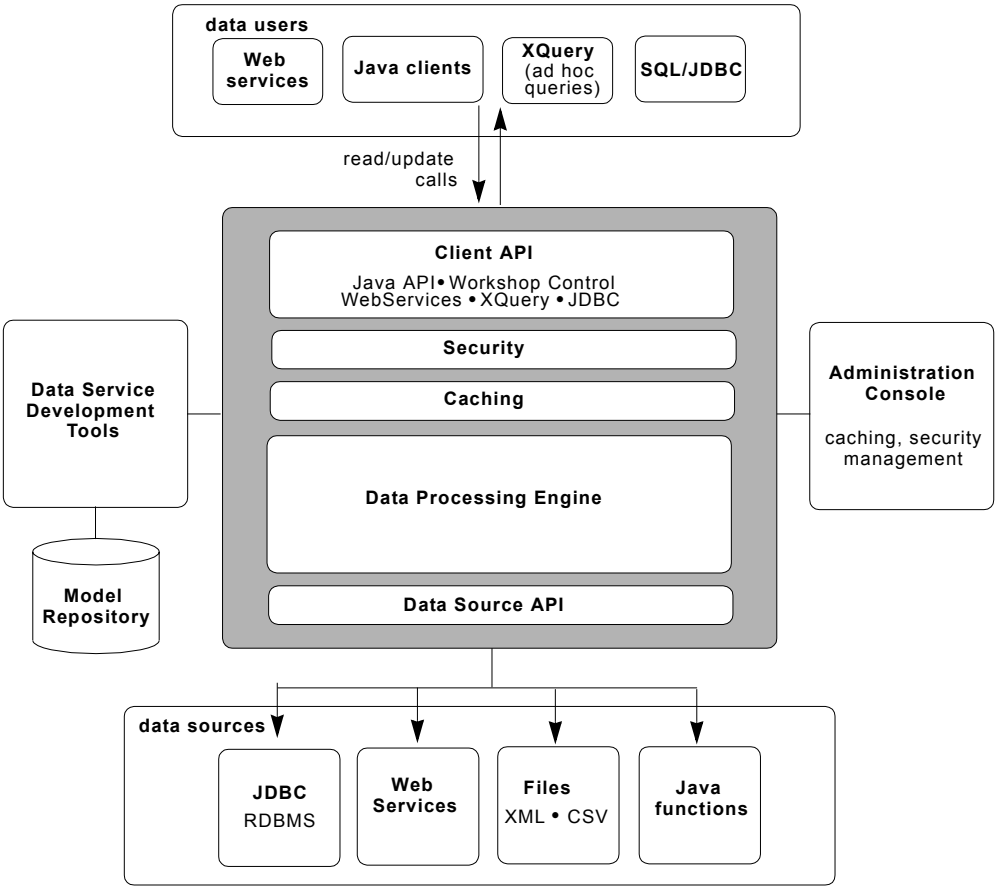
http://edocs.bea.com/wls/docs81/index.html

Liquid Data Architecture

As shown in Figure 2-1, Liquid Data provides a data integration layer between data sources and data users.

The core of the Liquid Data runtime component is the data processing engine. It is a distributed query processor that divides user requests into optimized sub-queries, which when possible are processed concurrently against the data sources. The core is supplemented by security and caching components and interfaces for acquiring and delivering information, as described further in the following section.

Figure 2-1 Liquid Data Components Architecture



Liquid Data Components

As depicted in Figure 2-1, the components and features of Liquid Data include:

- **Data Processing Engine**. The data processing engine is optimized for distributed data access to databases, Web services, and files.
- **Cache**. By caching frequently accessed data, you can improve response time and reduce load on back-end resources. For more information, see Chapter 7, "Performance and Caching."

- **Security**. In addition to taking advantage of security features of the underlying WebLogic server, Liquid Data lets you set read and update permissions for functions. For more information, see Chapter 8, "Securing Data."
- **Client API**. Application developers have several options for accessing data. The Service Data Objects API allows client applications to read and update the data through a typed or untyped interface. Query-oriented access is supported for XQuery through the ad hoc query mechanism. In addition, the Liquid Data JDBC driver enables JDBC clients, including SQL tools, access the information served by Liquid Data services.
- **Data source API**. Liquid Data supports many data source types, including:
  - **Relational sources**
  - **Web services**
  - **XML files**
  - **Delimited files**
  - **Custom Java functions**
- **Administration Console**. The administration console allows you to control Liquid Data features such as caching and access restrictions, as well as internal server behavior such as thread usage and memory. It also gives usage statistics regarding data services.
- **Design Tools**. Liquid Data includes tools for creating the data services layer, including tools for modeling and developing data services and creating XQuery functions. For more information, see Chapter 3, "Unifying Information with Data Services."

CHAPTER 3

Unifying Information with Data Services

This section describes data services, the fundamental components of the Liquid Data integration layer. The following topics are covered:

- What is a Data Service?
- Understanding Data Service Functions
- Data Service Transformations
- Advertising and Maintaining Data Services with Metadata

What is a Data Service?

A data service is the Liquid Data component that gives data users access to structured information. Typically, a data service models a unit of information in an enterprise, such as a customer, sales order, or product. The set of data services collectively comprise the data integration layer in an IT environment.

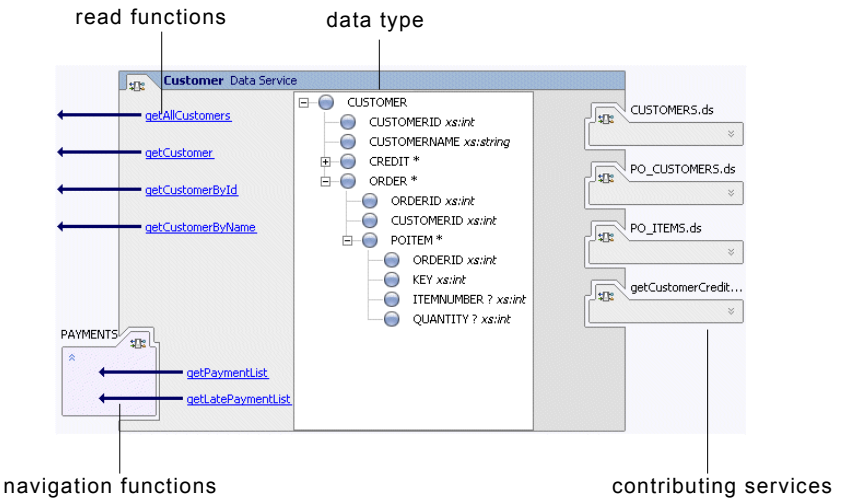
In concrete terms, a data service is a file with a `.ds` extension that contains XML Query Language instructions for querying and transforming data for data consumers. In many ways, a data service is no different from a conventional Web service. It is modular and reusable. It has a contract made up of public functions for accessing its services. It conceals the details of the service implementation—in this case, generally involving data acquisition and transformation—from the user

However, unlike a conventional Web service, at the core of each data service is an XML data type (shown in the Design View in Figure 3-1). When it acquires and integrates data in response to a request of a data user, the data service transforms the data into the shape defined by the XML data

type. The purpose of a data service is to give data consumers access to information in the format of this XML data type.

A service consumer—a client application or another data service—uses read functions defined for the data service to get its information. It can also call a navigation function to get information from a related data service, typically passing an instance of the current data service as an argument. A data service can have any number of read and navigation functions.

Figure 3-1 Workshop Representation of a Data Service



In addition to transforming data as prescribed by its target XML schema, a data service can operate on the data in other ways as well. It can perform mathematical calculations on numeric data, for example, or concatenate text strings. Whatever data-oriented operations you would like to be performed in order to provide a uniform, useful view of the data for its consuming applications can be encapsulated by a data service.

Understanding Data Service Functions

A data service function allows clients to get data from a data service. The data service interface consists of the public functions of the data service, which can be of several types:

- Read functions return data in the form of the data service XML type.
- Submit function allow users to persist changes to the back-end storage.

- Navigate functions return data from related data services.

Keep in mind that—in accord with the aims of SOA—data services in general (and Liquid Data functions in particular) are intended to provide "course grain" access to data. Coarse grain access alleviates clients from having to identify and aggregate information that collectively makes up a distinct business entity.

In addition to public functions, a data service can have private functions. Private functions can be used only by other functions within the data service. They generally contain common processing logic, that is, operations for use in more than one function in the data service.

For auxiliary functions of interest across multiple data services, you can use function libraries. A function library (`.xfl` extension) contains operations that can be called from various data services.

Read functions on a data service can be defined to return information in various ways. For example, the data service may define read functions for getting all customers, customers by region, or customers with a minimum order amount.

However, data users often want to access information in ways that are not entirely pre-specified or limited by the design of a data service. The filtering and ordering API allow client applications to control further what data is returned by a data service read function call based on conditions specified at runtime. Instead of having to create a read function for every possible client requirement, the service designer can create generalized read function against which clients can apply their own filtering or ordering conditions at runtime. It is left to the service designer to decide whether to create a narrowly-defined read functions, with access conditions built into the interface, or to create more generic read functions and allow the client to specify filtering conditions.

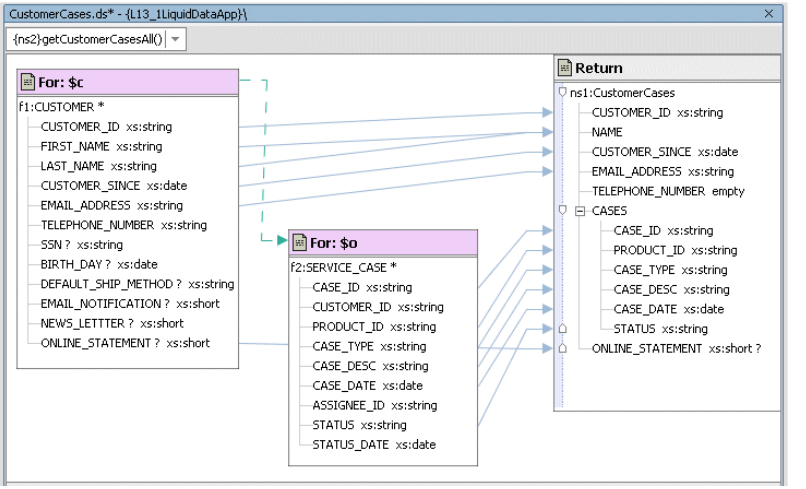
Data Service Transformations

Between acquiring data from backend systems and presenting that data to clients, a data service usually transforms the data in some way. The nature of the transformation is determined by the XQuery body of the called function.

Transformations can involve simple element mappings, more general shape transformations, data joins, or value modifications by string operators, mathematical operators, or date operators. The structure and data in a transformation can be controlled by logical constructs such as if-then-else statements. The transformation may join data from multiple sources or filter data.

The XQuery Editor (shown in Figure 3-2) enables you to create a transformation query by dragging and dropping nodes between source and target schemas. A target schema is the XML type of the data service, that is, the shape of the data (in XML schema) returned in response to service requests.

Figure 3-2 Graphical XQuery Transformation Designer



The internal language of a data service, XQuery (short for XML Query Language), is an emerging standard language for querying XML-based information. XQuery has many features in common with SQL, such as where clauses. However, unlike SQL, which was intended for working with simple two-dimensional data format (rows and columns), XQuery is designed to work with the richer, Web service data that uses the nested structure approach of XML.

The XQuery functions in a data service determine what data is acquired by the service and how it is transformed. But the transformation is not limited to the format of the data—it can also affect the data value itself, for example, by concatenating strings or calculating mathematical operations on numeric values.

Advertising and Maintaining Data Services with Metadata

A significant challenge for enterprises is not only in integrating disparate data sources, but in advertising the availability of data in a consistent way. If potential users of a data source do not know it exists, the data is effectively lost. Once a developer has gone to the trouble of creating a unified Customer data service, for example, other developers need to be able to discover and reuse it for their own work.

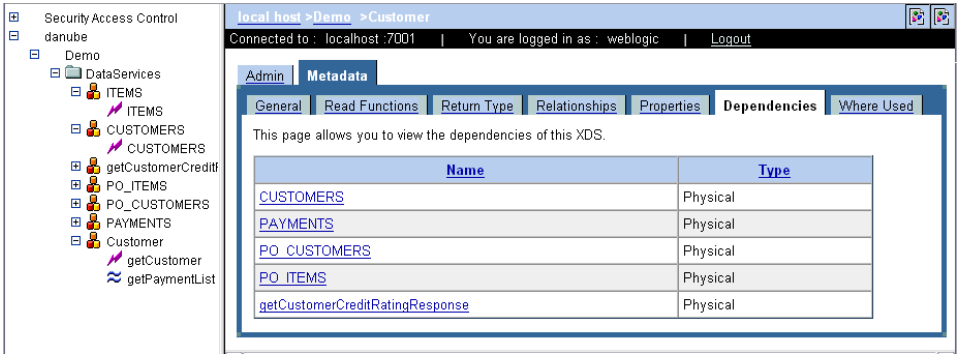
Liquid Data uses a set of data service descriptors (or metadata) to provide information on data services. The metadata describes the data services—what information they provide and where the information comes from (that is, its lineage).

In addition to documenting services for potential consumers, metadata helps administrators determine what services are affected when inevitable changes occur in the data source layer. If a database changes, the administrator can easily tell which data services are affected by the change.

When changes do occur, a metadata synchronization wizard helps you to absorb data source level changes into the data services layers. The wizard, which is launched from Workshop, detects disparities between the schema of the data source and the physical data services. It highlights differences and allows them to be incorporated with a few clicks.

Those interested in metadata can access it in several ways. The metadata browser (shown in Figure 3-3) provides an HTML interface for browsing and searching metadata in various ways. Also, an API provides programmatic access to the same information.

Figure 3-3 Metadata Browser



CHAPTER 4

Modeling and a Service-Oriented Architecture

This section discusses how you can use modeling to build a data services layer. The following topics are covered:

- Addressing Complexity with Modeling
- Modeling Data With XML Schema
- Modeling Data Services

Addressing Complexity with Modeling

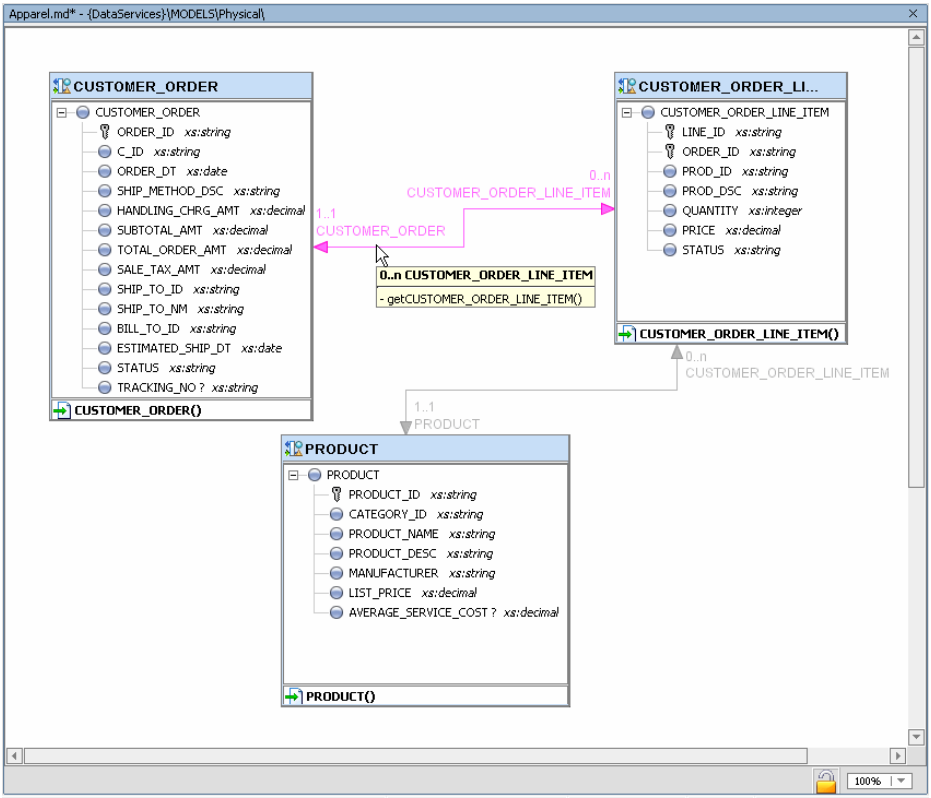
Generally, modeling helps to mitigate the complexity of large, intricate systems. In Liquid Data, modeling is no different. It provides a visual, representational view of data resources. It can present a high-level view of a complex system or it can break a complex problem into smaller pieces.

Modeling helps you to organize and document data services. It enables you to bridge the gap between the complexity of the underlying physical data landscape, on the one hand, and a logical, intuitive data model exposed by data services, on the other.

Modeling is generally the first step in developing a data services layer. It provides a roadmap for the development of the data services to be built on top of physical data resources.

The Liquid Data Modeler (shown in Figure 4-1) is a visual modeling tool for creating, managing, and modifying data services. You can create data services and define the relationships between them directly in the modeler.

Figure 4-1 Model Diagram



Modeling Data With XML Schema

Like a dictionary does for a language, a schema defines the vocabulary, rules, and conventions for representing information in a system. Liquid Data uses the XML Schema language to model data. XML Schema (a standard, XML-based schema definition language) gives Liquid Data a way to define normalized data, enabling consuming applications to use the data without regard for the inherent differences of its various underlying source representations.

Each data service has a data type, a named data shape defined in XML Schema. The schema defines the content, that is, the elements, attributes, and types of the data. For example:

```
<xs:element name="CUSTOMER">
   <xs:complexType>
      <xs:sequence>
```

```
            <xs:element name="CUSTOMERID" type="xs:int"/>
            <xs:element name="CUSTOMERNAME" type="xs:string" />
            ...
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

An XML schema file, named with a `.xsd` extension, is automatically generated when you import the metadata of a physical data source. For logical data services, you can create the schema by hand. In any case, an XML schema definition must exist for any information that you want to be made available as a data service.

A data service is validated against its associated schema at design time. Design-time validation prevents query development mistakes. At runtime, physical data service data is validated as well. Runtime schema checking ensures the validity of information returned by query execution.

The XQuery Editor helps you to create schemas for logical data services. As a starting point, you can associate the schema to a data service from an existing data source, such as a physical data source (and save the schema with a different name). Once you have saved and associated the new schema from the editor, you can change field names, remove fields, and so on, from the graphical, Design View editor.

Modeling Data Services

Just as it does for database design, data modeling helps to ensure the optimal design of your data services layer. Modeling lets you analyze and develop data services efficiently and in a way that most effectively resolves a complex data landscape into a sensible, business-level data model. Modeling also serves to document data services for use by others.

Physical data services can be considered the building blocks for logical data services. In the case of a relational data source, for example, importing metadata results in a data service being created for each table.

Modeling not only represents the data services in your Liquid Data implementation, it also captures the relationships between the data services. A relationship is a logical connection between two data services, such as between a customer data service and an orders data service.

The directionality of a relationship can be either:

- One way, in which data service *a* can navigate to data service *b* but *b* does not navigate to *a*.
- Two way, in which data service *a* can navigate to *b* and *b* can navigate to *a*.

Relationships are manifested in a data service by navigation functions. By traversing the relationship a client application can acquire data from a related data service from an instance of the current data service (for example, to get the orders that belong to a particular customer). Similarly, the Order data service may have a function that gets the customer associated with a particular order. This is an example of a two-way relationship.

A relationship—and, by extension, the navigation function derived from them—serves these purposes:

- It advertises a logical connection between data elements.
- It streamlines client programming by supporting the acquisition of related data from a single data service instance. For a code example, see "A First Look at SDO" on page 6-3.
- It encapsulates the details of how the logical connection of a relationship is physically manifested, that is, its key mappings, the nesting hierarchy, and so on.

CHAPTER 5

Designing Data Services

This section contains general guidelines and patterns for creating a data services layer. The following topics are covered:

- Using a Layered Data Integration and Transformation Approach
- Taking Advantage of Data Service Reusability
- Modeling Relationships

Using a Layered Data Integration and Transformation Approach

When planning a data service deployment, it is helpful to think of the data service layer in terms of an assembly line. In an assembly line, a product is built incrementally as it passes through a series of machines or assemblers that specialize in one aspect of the fabrication of the product.

Similarly, a well-designed data services layer transforms input (source data) into output (structured information) incrementally, through a series of small transformations. Such a design eases development and maintenance of the data services and increases the opportunity for reuse.

Note: Keep in mind that a multi-level data service implementation model described here are flattened when the data services are compiled for deployment. That is, adding conceptual layers does not add overhead to the data integration work performed by the Liquid Data deployment, and therefore does not affect performance.

By this design, distinct subsets of data services comprise sublayers in the overall transformation layer. As data passes from layer to layer—each of which specializes in one aspect of the transformation—the data is transformed from a more generalized state to a more application-specific state.

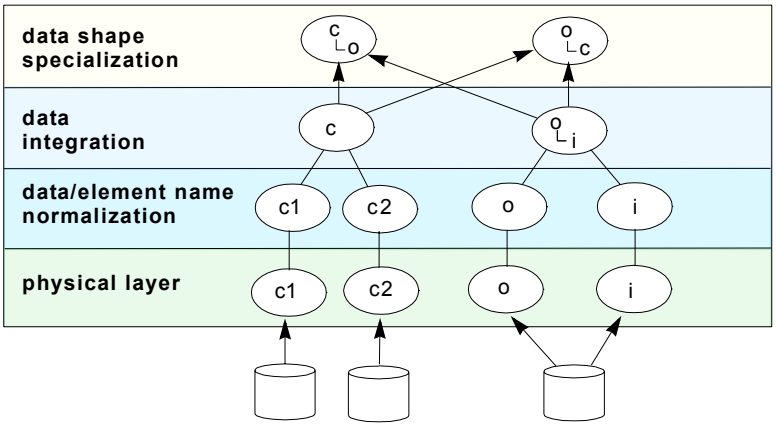
To further illustrate this design, consider a deployment with the following sublayers:

- The first sublayer (that is, the first one to touch the raw data) is the physical data services layer. This layer exists in any Liquid Data deployment, whether or not data is further transformed. The data services in this layer are generated for you when you import the metadata for a data source and normally should not be modified.
- The second sublayer of data services should normalize the data while retaining the data shape as imported. For example, it can change element names (that is, tag names) to make them consistent with other sources and make minor modifications to data values, for example, concatenating names or adjusting time values for a time zone.
- Data services in the next sublayer can then use the normalized data to represent integrated business entities in the data domain, such as an a unified view of a customer. The data services can unify data sources, for example, or change the shape of the data in any way desired.

  This sublayer does most of what might be called the integration work of the overall data services layer; it is where the integration logic and predicates and primary relationships are specified. (In small projects, this layer may be combined with the second sublayer. That is, it would contain data services that both normalize the data and define data shapes for the integration layer.)
- A final sublayer can specialize information specifically for applications. This layer, which can be thought of as the extended services layer, tailors information in a way that makes sense to particular applications or types of applications, such as executive dashboards, sales portals, or HR applications. For example, it might specify nesting in its data shape a way that is useful for particular applications, such as having order items as a child of a customer item or, on the other hand, customers as a child of orders (as shown in Figure 5-1).

For very large database sources, instead of creating a single master data service, it is best to decide what a client application needs and build corresponding, minimal data services. The concept is to build client-specific data services from a manageable number of views that query a reasonable number of data sources, providing an abstraction from the lowest level and most common relationships while keeping the overall view reasonably simple. Liquid Data also provides a metadata API that allows client applications to discover relationships between data services at run-time, allowing applications to navigate the data services without the need for a master data service.

Figure 5-1 Layered Data Services Design Strategy

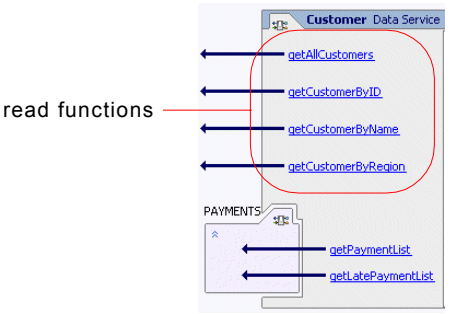


The most significant benefit of this approach is that it increases the opportunity for reuse within the overall data services layer. As shown in Figure 5-1, once you have defined a single form of a business entity (such as a customer) in a data service dedicated to the task, you can have multiple application-specific data services use the information without having to repeat data normalization and integration tasks. An additional benefit is that it aids maintenance because there is a clear separation of concerns between the data service layers.

Taking Advantage of Data Service Reusability

A typical data service design pattern within a data service is to have a single read function that defines the data shape without filtering conditions. The function may be declared private so that it can only be called by other functions within the same data service. Also, it would be the only one with integration logic. Additional functions, either in the same data service or in other data services, can use the private function to specify the filtering criteria users. Figure 5-2 shows the design view of a data service exhibiting this pattern.

Figure 5-2 Customer Data Service functions



The following XQuery sample demonstrates the mechanics behind data service reuse. This function, `getCustomerByName()`, filters instances based on the customer name:

```
declare function l1:getCustomerByName($c_name as xs:string)
as element(t1:CUSTOMER)*
{
    for $c in l1:getAllCustomers()
    where $c/CUSTOMERNAME eq $c_name
    return $c
};
```

The `getAllCustomers()` function, in turn, would assemble the data shape for the returned data, providing join logic and transformation, as shown its return clause:

```
...
return
     <t1:CUSTOMER>
       <CUSTOMERID>{fn:data($c/CUSTOMERID)}</CUSTOMERID>
       <CUSTOMERNAME>{fn:data($c/CUSTOMERNAME)}</CUSTOMERNAME>
       {
          for $a in f2:ADDRESS()
          where $c/CUSTOMERID eq $a/CUSTOMERID
          return
             <ADDRESS>
             <STREET>{fn:data($a/RTL_STREET)}</STREET>
             <CITY>{fn:data($a/RTL_CITY)}</CITY>
             <STATE>{fn:data($a/RTL_STATE)}</STATE>
             </ADDRESS>
```

```
        }
      </t1:CUSTOMER>
```

Keep in mind that client application themselves can specify filtering conditions on a data service function call. Therefore, the data service designer can choose whether to have broadly defined data access functions (that is, without filter condition), and let the client to apply filtering as desired, or narrowly by defining the criteria in the API.

Note: All functions whose bodies are FLWOR (For-Let-Where-Order-Return) statement should be be declared to return a plural rather than a singulr result; for example, `element(purchase_order)*` rather than `element(purchase_order)`.

Modeling Relationships

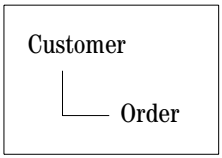
There are several ways to implement a logical relationship between distinct units of information with data services:

- Data shape containment
- Navigation functions

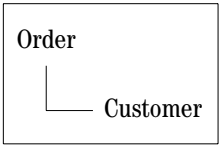
When containment is implemented in the data shape, it means that the XML data type of the data service is nested; that is, one element is the parent of another element. For example, in the following sample a customer element contains orders:

```
<customer>
   <customerId>...</customerId>
   <customerName>...</customerName>
   <orders>
      <order>...</order>
      <orderId>...</orderId>
   </orders>
</customer>
```

A diagram of this XML structure would be:

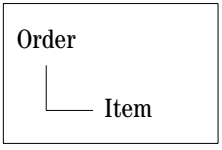


In this type of containment, the parent-child hierarchy between the customer and order is locked into the data shape. This nesting might make sense for most applications, particularly those oriented by customer. However, other applications may benefit from an orders-oriented view of the data. For example, an inventory application may prefer to work with the data in an orders-first fashion, with the customer as a child element of each order.



Conceptually, in this case it could also be said that an Order is not existence-dependent on a Customer. If a Customer deleted, it may not necessarily follow that the customer's deleted should be deleted as well.

Alternatively, other relationships do not require this type of hierarchical flexibility. In most cases, this also implies that the business entity's existence does depend on the existence of the parent. For example, consider an order that contains items.



In most logical data models, it would not make sense to have an item outside of the context of the order that contains it. When deleting an order, it is safe to say that composing order items would need to be deleted as well.

The choice when modeling such containment either through a relationship or through data shape nesting is informed by these considerations. When choosing whether to model containment either through data shape nesting or using relationships, it is recommended that:

- Existence-dependent entities are modeled as nested elements.
- Existence-independent entities are modeled as relationships.

By modeling independent entities with bi-directional relationships, data service users and designers can easily specialize the logical hierarchy between business entities as best suited for their applications.

CHAPTER 6

Using Service Data Objects (SDO)

This section describes the Liquid Data client-side application programming object model, Service Data Objects (SDO). The following topics are covered:

Introducing SDO

SDO is a Java-based data programming model and architecture for accessing and updating data. SDO is defined in a joint specification proposed by BEA and IBM (JSR 235). SDO is intended to give applications an easy-to-use, uniform programming model for accessing and updating data, regardless of the underlying source or format of the data.

Unlike existing data access technologies such as JDO or JDBC, SDO specifies both a static and a dynamic interfaces for accessing data. The static (or strongly typed) interface gives client developers an easy-to-use, update-capable model for accessing values. A static interface function is in the form:

```
getCUSTOMERNAME()
```

As implied by the sample function name, to use the static interface the developer must know the types of the data (that is, CUSTOMERNAME) at development time. If the types are unknown at development time, the developer can use the dynamic (or loosely typed) interface. Such calls are in the form:

```
cust.get("CUSTOMERNAME")
```

A dynamic interface is useful when the data type is not known or defined at development time and is particularly useful for creating programming tools and frameworks across data source types.

In addition to a programming interface, SDO defines a data programming architecture. A component of this architecture called the mediator serves as the adapter between the client and data source. A mediator specializes in exchanging data with the data source. It knows how to retrieve data from a particular source and pass changes back to it. Data is passed in a datagraph, which consists of a graph of data objects.

As mentioned in the SDO specification, a particular SDO implementation is likely to have mediators intended that are specialized for a particular source type, such as for databases. Liquid Data provides a data service mediator that resides between SDO clients and the data integration layer.

With SDO, clients use data in an essentially stateless, disconnected fashion. When Liquid Data gets data from a source, such as a database, it acquires one or more database connections only long enough to retrieve the data. Once the data is acquired, Liquid Data releases the connections. The client works on a local copy of the data, disconnected from the data source.

Disconnected data programming means that the client can operate on the data without generating additional network traffic. The network is accessed again only when the client wants to apply the data changes to the source. Disconnected data access contributes to a scalable, efficient computing environment because back-end system resources are never tied up for very long.

Optimistic concurrency rules are used to ensure the integrity of data when updates occur. With optimistic concurrency, data at the data source is not locked while a client works with it. Instead, if updates are made to the data, potential conflicts (such as other clients changing the same data after the client got it) are detected when the updates or propagated back to the sources.

For complete information on Service Data Objects, including the specification and Javadoc, go to:

http://dev2dev.bea.com/technologies/commonj/sdo/index.jsp

Getting Data Objects

Liquid Data uses SDO as its client-side programming model. Simply put, this means that when a Java client invokes a data service's read function—either through the Liquid Data Mediator API or through the Workshop Control—it gets a return value in the form of an SDO data object. A data object is the fundamental component in the SDO programming model. It represents a unit of structured information, with static and dynamic interfaces for getting and setting its data values.

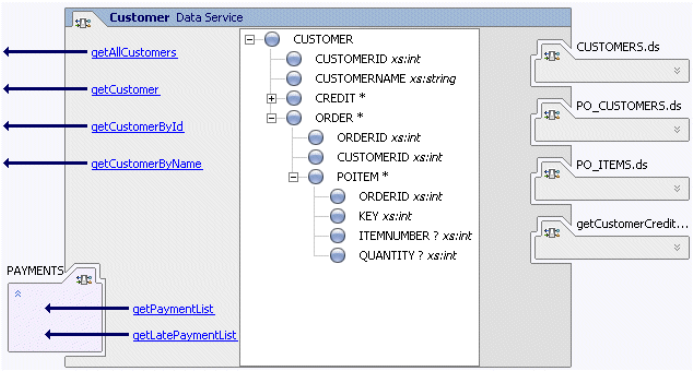
Once the data is acquired, the client works on it in disconnected fashion. It passes the changed data back to data access layer only if it wants to save the changes to the back-end data source.

A First Look at SDO

This section introduces you to SDO programming through a small code sample. The sample shows how SDO provides a simple, easy to use client-side data programming model.

The sample is written against the data service shown in Figure 6-1.

Figure 6-1 Data Service Design



The data type for the Customer data service is CUSTOMER. A data type is a structured XML document, in this example composed of properties such as customer ID, name, and orders. The data for a CUSTOMER instance comes from four other data services, CUSTOMERS.ds, PO_CUSTOMERS.ds, PO_ITEMS.ds, and getCustomerCredit.ds (although the details of the composition of the data service do not need to be of concern to the data service user).

The sample Customer data service has several methods for getting data type instances, including `getCustomer()`, `getCustomerById()`, `getPaymentList()`, and so on. The function `getPaymentList()` is a navigation function. It does not return a CUSTOMER document; instead it returns data from a data service for which a logical relationship has been defined. In the case of `getPaymentList()`, the related data service named PAYMENTS returns the payment history for a given customer.

The navigation function makes it easy for client applications to acquire additional related data that is likely to be of interest when working with CUSTOMER documents. With the same data service handle used to get a customer, they can get that customer's list of payments.

The result is code that is concise and readable and easy to maintain, as shown in the following snippet.

Listing 6-1 SDO Sample

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.bea.ld.dsmediator.client.*;
import org.openuri.temp.schemas.customer.CUSTOMERDocument;

public class myCust {
   public static void main(String[] args) throws Exception {
      Hashtable h = new Hashtable();
      h.put(Context.INITIAL_CONTEXT_FACTORY,
      "weblogic.jndi.WLInitialContextFactory");
      h.put(Context.PROVIDER_URL,"t3://localhost:7001");
      h.put(Context.SECURITY_PRINCIPAL,"weblogic");
      h.put(Context.SECURITY_CREDENTIALS,"weblogic");

      DataService ds = XmlDataServiceFactory.newXmlService(
                new InitialContext(h),
                "Demo",
                "ld:DataServices/Customer");
      Object arg[]={new Integer("987655")};
      CUSTOMERDocument myCustomer =
                (CUSTOMERDocument) ds.invoke("getCustomer",arg);
      myCustomer.getCUSTOMER().setCUSTOMERNAME("BigCo, Inc");
      ds.submit(myCustomer,"ld:DataServices/Customer");
      System.out.println(" Customer information: \n" + myCustomer);
   }
}
```

Notice that once the proper packages have been imported and the initial context to the server has been made, in about five lines of code the client application:

- Gets data from ultimately four different data sources.
- Modifies a value (customer name).
- Submits the change to Liquid Data, which propagates the change back to the data sources.

The complexity of this entire procedure is hidden from the client application developer. Instead, the complexity is handled at the data services layer and by the Liquid Data framework.

Data Updates

As it does for reading data, SDO gives client applications a unified interface for updating data. With Liquid Data, client application can modify and update data from heterogeneous, distributed sources as if the data were from a single entity. The complexity of propagating changes to diverse data sources is hidden from the client programmer. With a single update call, a client can propagate changes to a variety of data sources.

Data source updates occur in a transactionally secure manner; that is, given an update call that affects multiple sources, all updates to individual data sources within the update call either succeed or fail together. (Note that it is possible to override this behavior if needed.)

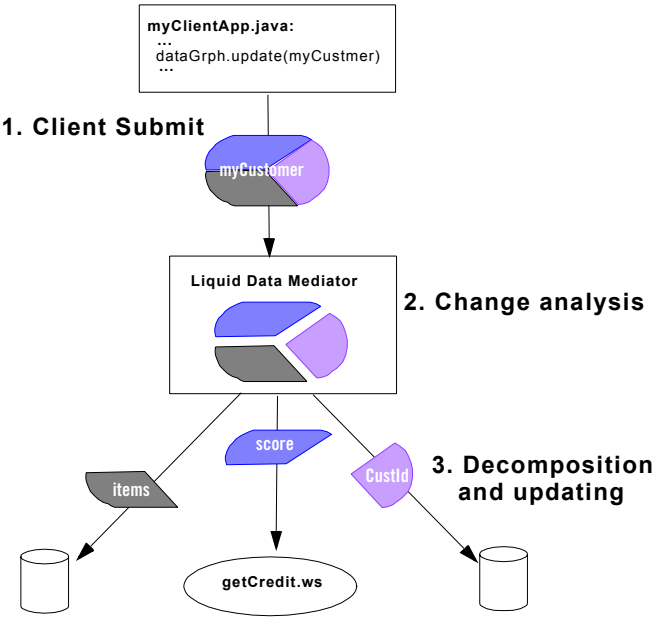
From the data service implementor's point of view, the task of building a library of update-capable data services is considerably eased by the Liquid Data update framework. For relational sources, Liquid Data can often propagate changes to the data sources automatically. For other sources, or to customize relational updates, you can use the Liquid Data update framework and tools to quickly implement a wide range of update-capable services.

As shown in the Figure 6-2, updates occur through a process in which the requested change is first analyzed to determine, among other things, the lineage of the data. The Liquid Data mediator then decomposes the submitted object into its constituent parts and propagates the changes to the data source.

At any point in this process, you can have your own code programmatically intervene, for example, to validate the update values or for auditing purposes.

For more information on updates, see the *Liquid Data Application Developer's Guide*.

Figure 6-2 Liquid Data Source Updates



CHAPTER 7

Performance and Caching

This chapter describes the performance features of BEA Liquid Data for WebLogic. It covers these topics:

- Overview
- Query Optimization
- Caching
- Strategies for Security and Caching Policies

Overview

Poor performance can outweigh many of the advantages that an otherwise carefully designed data services deployment provides. From a data user's perspective, response times of the data access layer should be close to that of native data access mechanisms.

Liquid Data includes a number of configurable settings and features for ensuring good performance, as well as numerous internal features. With caching, data response times actually improves upon those provided by native access mechanisms.

Query Optimization

The data integration language behind data service functions is the XQuery language. As a declarative language, XQuery affords many opportunities for optimization. In general terms, a declarative language focuses on what needs to be done, not on how things are to be done (as is the case for an imperative language). As such, the Liquid Data engine is free to choose the most effective way to

execute a given query, not only for the best performance of the data services layer but also to minimize the burden on the data sources as well.

Among the types of query optimization Liquid Data performs include:

- When possible, Liquid Data pushes predicate evaluation, join operations, aggregates, and other data manipulation functions from XQuery to the underlying SQL databases to minimize the number of database calls required and reduce the volume of data returned from those calls.
- Liquid Data handles layered data services by a process called view unfolding. Instead of retrieving the full data set encompassed by a multi-level function call, the Liquid Data retrieves only the data relevant to the actual request.
- Liquid Data combines multiple "trips" to the same database implied by a function into a single access when sensible.

You can view how the engine has compiled a query using the plan view.

Caching

Caching improves the responsiveness of the client application and minimizes the burden on back-end resources. With caching, Liquid Data stores the results returned from a data service function locally. When a function call is made again with the same parameters, Liquid Data can respond with the cached copy of the data, thereby avoiding repeated calls to the back-end data sources.

Caching with Liquid Data can be managed at a highly granular level. You can enable or disable caching and set the time-to-live on a per function basis. This allows you to apply caching policies as best suited for the type of information. If the information is apt to be long standing without change, the cache can expire that data less frequently. If information changes frequently, you can have the cache expire it frequently as well.

If a client application wants to be sure that it acquires the latest information, it can deliberately bypass the cache, specifying a cache pass through in the data service functions call.

You can manage the cache, for example, by setting time-to-live values and purging the cache through the Liquid Data Administration Console.

Strategies for Security and Caching Policies

You can apply security policies to secure resources by application, data service function, and element. You can also implement data-driven security policies. For more information about security, see "Securing Data" in the *Liquid Data for WebLogic Administration Guide*.

The benefits of caching are conferred to calling data services as well as external clients. By caching , you can improve response times for clients and reduce the processing burden on back-end systems. For informaton about caching see "Configuring the Results Query Cache" in the Liquid Data for WebLogic Administration Guide.

CHAPTER 8

Securing Data

This chapter discusses the security features of Liquid Data. It covers the following topics:

- Ensuring Data Security
- Securable Liquid Data Resources
- Understanding Security Policies

Ensuring Data Security

Integrating enterprise data with Liquid Data does not mean having to compromise the security of sensitive information. Because different data has different security requirements, the ability to apply access control policies to data items is essential. Not all users who need access to general customer information, for example, should have access to sensitive information such as credit card numbers.

Like other components of the WebLogic Platform, Liquid Data supports role-based security authorization. Authorization involves granting a user (either individually or as a member of a group or security role) permission to access resources provided by a Liquid Data deployment.

The WebLogic Platform provides the security framework that handles authorization based upon information in the context of the user request. By default, Liquid Data uses the WebLogic Authorization provider for authorization. If desired, other modules, including third-party authorization modules, can be used as well.

Security policies are enforced no matter how the client attempts to access a resource, from the Mediator API, the Liquid Data Control API, JDBC, or a web service.

Securable Liquid Data Resources

Liquid Data enables you to secure resources at a range of granularity levels, from the application level to the level of individual data elements.

Specifically, securable resources in Liquid Data include:

- **Applications**. An application-level policy applies to all data services in a Liquid Data deployment. You can configure the application setting to be one of the following:
  - block all access except as permitted by a more specific policy
  - permit access (even to unauthenticated users) except as blocked by a more specific policy.
- **Functions**. Secures specific functions in a data service. Function-level security allows you to specify differing access levels between read functions and submit functions. Since submit functions allow users to modify back-end data, they often require a more restrictively defined level of access control.
- **Data Elements**. Secures individual data items within a data service's return type. (In terms of database security, element level security corresponds to column-level security in databases.) For example, you can secure only the credit card number of a customer document.

  If a given user does not meet the security condition defined for an individual element, the element is redacted from the final result; that is, the customer information is provided with the credit card item missing. Element-level security applies across data service functions.
- **Document Instances**. An instance-level policy is data driven; it allows you to secure access to Liquid Data resources based on the value of the data being returned. Data-driven security is equivalent to row-level security for databases. It lets you, for example, block access to particular users if an order amount exceeds a specified value.

You can specify security policies that control access to the Liquid Data Console itself. The policies determine who can access particular pages in the console by their functional category, whether administration-based (for configuration and monitoring pages) or informational (for data service metadata pages).

Understanding Security Policies

A security policy determines whether a user can access a Liquid Data resource. With the WebLogic Authorization module, you can create policies based upon user identity, the user's group or role affiliation, time of day, development mode of the server, or any combination of these. Access policies can be used individually or together so that you can apply security in the manner that best matches your needs.

You can create a data-driven policy in the Liquid Data Console as an XQuery function. The function can perform any evaluation and processing steps desired, given the identity of the user making the request and the value of the requested data. To permit access, the function simply returns true or false to block it.