BEA AquaLogic Data Services Platform™

Concepts Guide

Note: Product documentation may be revised post-release and made available from the following BEA e-docs site:

http://e-docs.bea.com/aldsp/docs21/index.html

Version: 2.1
Document Date: June 2005
Revised: March 2006

Copyright

Restricted Rights Legend

Trademarks or Service Marks

March 16, 2006 1:26 pm

Contents

CHAPTER 1

Introducing BEA AquaLogic Data Services Platform

This chapter provides an overview of BEA AquaLogic Data Services Platform (DSP). It covers the following topics:

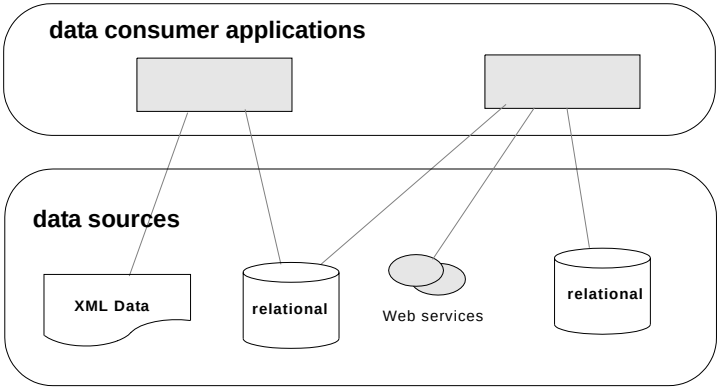
Note: Data Services Platform was formerly named Liquid Data. Some artifacts of the original name remain in the product, installation path, and components.

The Extraordinary Cost of Information Fragmentation

Information resources are often the most important assets of an enterprise. Availability of accurate and timely nformation 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.

It is a fact of life, however, that as enterprises evolve, their data resources tend to become increasingly fragmented. Corporate acquisitions and mergers, new technologies, and changing IT strategies all contribute to data lakes and puddles, in which pockets of information become isolated by physical and technical boundaries. In a typical enterprise information pools in databases and files, within applications, or externally, supplied on-the-fly by business partners, vendors, and clients as Web services or XML documents. Figure 1-1 illustrates this fragmentation of data.

Figure 1-1 Direct Data Access Applications



As a consequence, integrated information becomes so expensive to create that it is — except for the highest priority needs — effectively unavailable to the decision makers, employees, and customers when they need it. And as for those highest priority projects, developers must cope with a daunting variety of data access mechanisms and APIs. Resulting applications 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 the data access code. In sum, 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 access programming.

Other costs may be more difficult to gauge but are certainly as significant — from hampered decision making and poor responsiveness to the high cost of integration talent.

What is a Data Services Platform?

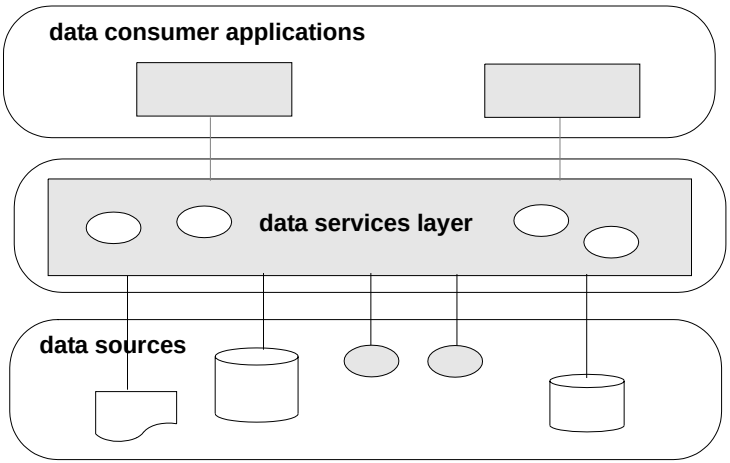
A data service is a query-based access layer that facilitates the ability of calling applications to access and manage disparate data.

Data Services Platform (DSP) is the AquaLogic component dedicated to developing and deploying integrated data services. Data services give consumers an easy-to-use, uniform model for accessing heterogeneous, distributed data.

DSP significantly simplifies the task of creating and deploying reusable data services. It provides tools and frameworks for rapidly generating physical data services based on existing data sources and for creating view-like logical data services.

Most data in an organization exists as queryable data or application data. From the perspective of the data consumer, DSP presents a sensible, uniform model for getting and using information. The application simply retrieves or updates its data by calling one of the data service's public APIs. The backend data may come from legacy applications, relational databases, Web services, and so forth. Thus data services can be viewed as a layer integrated between the data consumer and the data sources, as illustrated in Figure 1-2.

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



DSP is a virtual data layer in the sense that, except for caching, data is not kept in any DSP 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 enterprise data model. Like a Web service, it exposes an API 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 complex datatype. The function uses XQuery to acquire, transform, and aggregate data from external sources.

Returned data is poured into an XML data shape defined as the XML type of the data service. Because an individual data service represents a relatively small unit of information, response time is generally superior. Other features — such as caching and query optimization — help to ensure the optimal performance of the data services layer.

DSP 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, DSP gives client applications a unified interface for updating data. DSP 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 DSP.

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 DSP update framework. For relational sources, DSP can propagate changes to the data source automatically. For other sources or to customize relational updates, you can use the DSP 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.

Data Services and a Service-Oriented Architecture

A Service-Oriented Architecture (SOA) is an IT environment architectural style. 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.

To appreciate SOA it is helpful to consider that, 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 the monolithic model 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 modular, self-contained, callable processes. That way other systems (such as Human Resource services or applications) can easily leverage financial computing resources for its current or future needs. In this way SOA can be viewed either as breaking down barriers between applications or eliminating the need for them entirely.

With SOA the IT infrastructure operates in an integrated, organic fashion like a virtual application that spans the organization.

Role of Data Services

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. New services can be exposed and used without requiring extensive changes to existing applications. The result is a service layer that is highly adaptive and change tolerant.

Data services fit perfectly into the SOA picture by providing a data abstraction layer between data users and the underlying data sources. Thus the application developer can count on working with a uniform, well-defined SOA-based data access interface. The need for data access specialists capable of establishing connections to back-end data is greatly reduced. The application simply calls a public function and good things happen.

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

Other benefits of DSP 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, DSP makes it easy to implement a data services layer that lets developers use simple function calls instead of tedious custom code.
- **Consolidate data management**. DSP 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

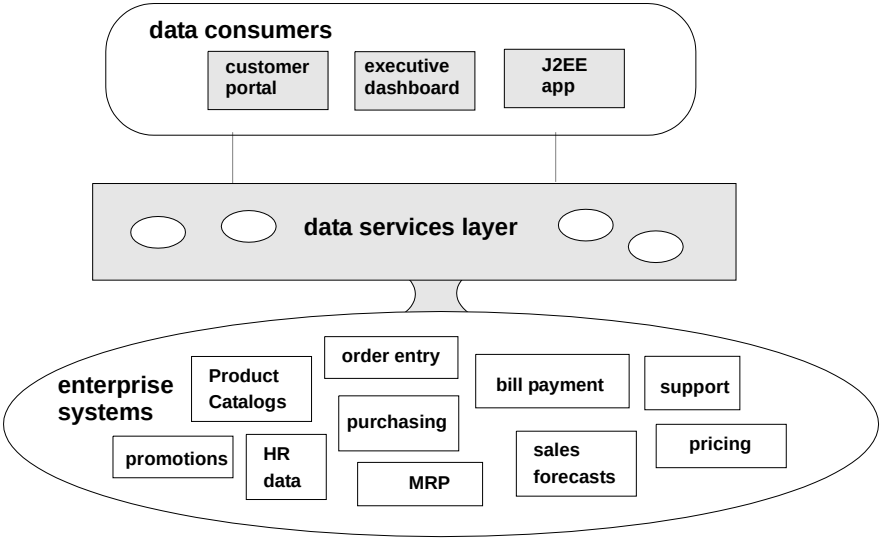
A data model typically organizes data in a way that represents business entities. With Data Services Platform, you can define the data model that makes sense for your organization. Business entities common to most organizations include, for example, products, orders, and customers. A data service Customer 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 underlying data sources for the elements that make up the Customer object are only of interest wheth the underlying structure or availability of that data changes. The data service can also specify security, caching, and other policies.

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, Data Services Platform 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 data service policies, such as security and caching, on information used across applications and services.

Figure 1-3 Anatomy of a 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 DSP deployment is captured by metadata. In DSP, 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. For example, you can determine how changes to the underlying data layer will the data services layer. When changes occur, as is inevitable, tools for synchronizing source metadata ease the difficulty of applying changes to data services.

The metadata browser, a component of the Data Services Platform Console, provides an easy-to-use 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 application or process 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.

Logic is encapsulated in a data service in the form of queries written in the XML Query (XQuery) language. XQuery 1.0, an emerging W3C 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, freeing developers from having to learn extensive rules and techniques for writing optimized query functions.

Those familiar with SQL should find XQuery easy to learn since they have many features in common. DSP provides an XQuery Editor that allows you to create XQueries through graphical gestures using an easy-to-use IDE tool.

Data Consumers

Once the DSP application has been deployed to a WebLogic Server, clients can use it to access real-time data. DSP supports several data clients:

- Java
- Web services
- Data service control
- JDBC/SQL
- ADO

In all cases Data Services Platform supports data access through Java interfaces (DSP Mediator API and Workshop Data Service control) or through Web services that act as wrappers for the data services.

As an additional option, the Data Services Platform 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 DSP 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 Data Service-Oriented Development Project

The following are the basic steps for managing a data service in a SOA environment:

1. **Establish project requirements.** This entails identifying available data resources, business logic requirements, security needs, 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 profiles for them. Importing metadata for the sources automatically creates the physical data services that are the building blocks of a 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 can help you plan and document your data services implementation. In Data Services Platform such services can be created and modified directly from a model diagram.
4. **Implement logical data services.** Logical data services can transform, filter, and aggregate disparate data, as well as prescribe constraints on the data, security, caching and other properties. For more information 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 customize update routines provided with Data Services Platform.
6. **Deploy the data services**. The query functions are deployed to a server for universal secured availability.
7. **Configure cache and security settings**.
8. **Maintain and monitor the data services layer**. DSP includes tools for evaluating the impact of changes to the data source layer and for synchronizing data services with the modifications.

Data Service Development Roles

In a SOA environment 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 are 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 application developers.
- *Application Developers* create client applications that use data service functions and procedures to access and manage real-time information.
- *System Administrators* Administrator perform the following types of tasks:
  - Deploy the data service to a server such as a WebLogic domains.
  - Configure access to data sources.
  - Set up and configure caching.
  - Implement security such as configuring users, groups, and defining security roles.
  - Monitor operation, tune performance, and provide support.

Additional Technical and Product Information

This section provides a compendium of links to technical and product information related to BEA AquaLogic Data Services Platform. These documents are subject to revision on a frequent basis and may reflect earlier versions of the product in some aspects.

- **Programmatic interfaces.** AquaLogic Data Services Platform supports several different programmatic interfaces enabling enterprises to build a single data services layer that can be consumed by a wide variety of applications in the enterprise. This technical note covers the different programmatic interfaces supported by AquaLogic Data Services Platform along with typical usage scenarios.
- **Caching.** AquaLogic Data Services Platform provides a flexible mechanism to manage caching of data services. This tech note covers the caching features provided by the platform together with their benefits and the common scenarios in which caching is used.
- **Metadata Management.** Metadata management capabilities provided by AquaLogic Data Services Platform significantly lowers the cost of maintaining and evolving a data services layer. This technical note covers the metadata management capabilities provided by the platform.
- **Modeling.** Modeling is an important first step in any software development project. It becomes increasingly important in distributed environments in which organizations have multiple information sources and domain expertise spread across multiple geographically dispersed groups. This technical note covers the modeling capabilities and key benefits provided by the AquaLogic Data Services Platform.

- **Security.** BEA AquaLogic Data Services Platform provides a rich security framework that enables data service architects to capture and manage the security policies in a single place. This tech note briefly describes the security features provided by BEA AquaLogic Data Services Platform, along with its benefits and the common usage scenarios to which security can be applied.
- **Update and SDO.** AquaLogic Data Services Platform provides a rich, disconnected programming model in the form of Service Data Objects (SDOs). The SDO specification defines a Java-based programming API for data access and update independent of the underlying physical sources. This technical note covers that and all of the platform's update features along with their associated benefits.

CHAPTER 2

Architecture

This chapter describes the architecture of BEA AquaLogic Data Services Platform (DSP). The following topics are covered:

- WebLogic Platform and Data Services Platform
- Data Services Platform Architecture
- Data Services Platform Components

WebLogic Platform and Data Services Platform

DSP 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 is WebLogic Server. DSP leverages WebLogic Server technologies 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 using DSP services need only establish an initial context to the WebLogic Server where DSP is deployed. This context — a JNDI mechanism for identifying a resource on a network — is similar to a database connection in that it identifies the server location 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 WebLogic Server documentation at the following URL:

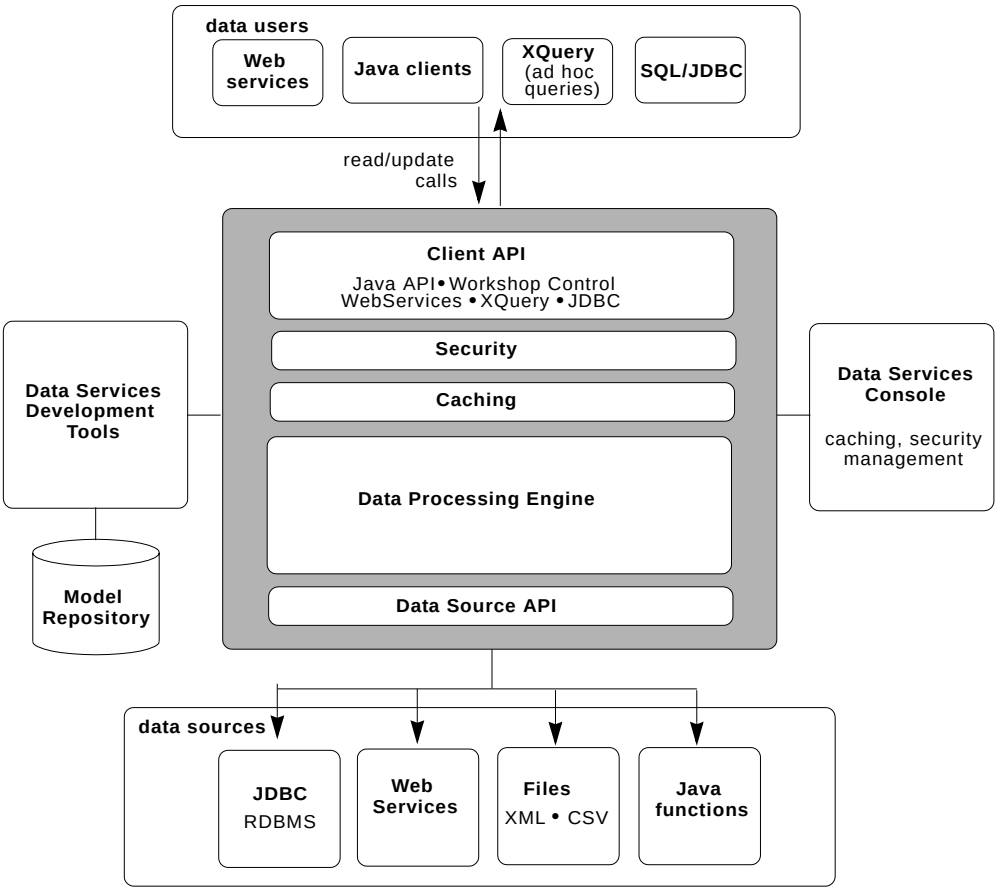
`http://edocs.bea.com/wls/docs81/index.html`

Data Services Platform Architecture

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

The core of the Data Services Platform runtime component is the data processing engine. It is a distributed query processor that divides user requests into optimized sub-queries which, as 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 Data Services Platform Components Architecture



Data Services Platform Components

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

- **Data Processing Engine**. The XQuery data processing engine is optimized for distributed data access to databases, Web services, and files.
- **Cache**. By caching frequently accessed data to a database, you can improve response time and reduce the load on back-end resources. For more information, see Chapter 6, "Performance and Caching."
- **Security**. In addition to taking advantage of security features of the underlying WebLogic server, DSP lets you set read and update permissions for functions. For more information, see Chapter 7, "Securing Enterprise Data."
- **Client API**. Application developers have several options for accessing data. The Service Data Objects (SDO) API allows client applications to read and update the data through a typed or untyped interface. Query-oriented access is supported through an ad hoc query mechanism. In addition, the DSP JDBC driver enables JDBC clients, including SQL tools, to access information provided by DSP services.
- **Data source API**. DSP supports many data source types, including:
  - **Relational sources**
  - **Web services**
  - **XML files**
  - **Delimited files**
  - **Custom Java functions**
- Data Services Platform Console**.** The DSP Console allows you to manage function caching and security access, as well as internal server behavior such as thread usage and memory. It provides usage statistics regarding data services, as well as auditing and logging of the runtime engine.
- **Design Tools**. DSP includes tools for creating the data services layer, including developing data services, modeling, 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 BEA AquaLogic Data Services Platform 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?

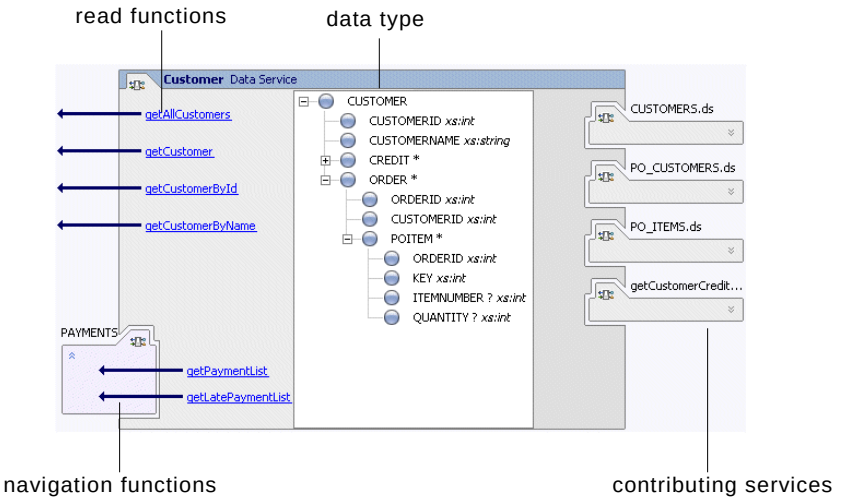
A data service gives data users access to any information in the enterprise through a single access layer. Typically, a data service models a unit of information, such as a customer, sales order, or product. The set of data services collectively comprise the data integration layer in an IT environment.

In concrete terms, a data service is a file with a `.ds` extension that contains XML Query Language instructions for querying and transforming data. In many ways, a data service is similar to a Web service. It is modular and reusable. It has a contract made up of public functions and procedures for accessing its services. It conceals the details of the service implementation — 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 DSP acquires and integrates data in response to a request, it is returned in the shape defined by the XML data type, it concrete terms, its schema.

A service consumer — a client application, process, or another data service — uses read functions defined in 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. Another facility available from data services are procedures. These routines can invoke external processes that have side effects rather than return data. A data service can have any number of read functions, procedures, and navigation functions.

Figure 3-1 Workshop Representation of a Data Service



In addition to forming returned data as prescribed by its target XML schema, a data service can operate on data in other ways as well. It can through build-in XQuery functions perform mathematical calculations on numeric data, for example, or concatenate text strings.

Understanding Data Service Functions

It is through data service functions that client applications, controls, and processes obtain enterprise data. The data service interface consists of the public functions of the data service, which can be of several types:

- **Read functions.** Returning data in the form of the data service XML type.
- **Navigate functions.** Returning data from related data services.
- **Procedures.** Executing side effecting functions in the enterprise.

- **A submit function.** Allowing authorized users to insert, delete, and change back-end data.

It is important to keep in mind that — in accord with the goals of SOA — data services in general (and DSP functions in particular) provide "course grain" access to data. Coarse grain access relieves clients from the druggery of identifying and aggregating information that collectively makes up a distinct business entity.

In addition to public interfaces, 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.

There is also a facility for providing auxiliary functions across multiple data services. These are called XQuery Function Library (`.xfl`) files.

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 architect to decide whether to create a set of 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.

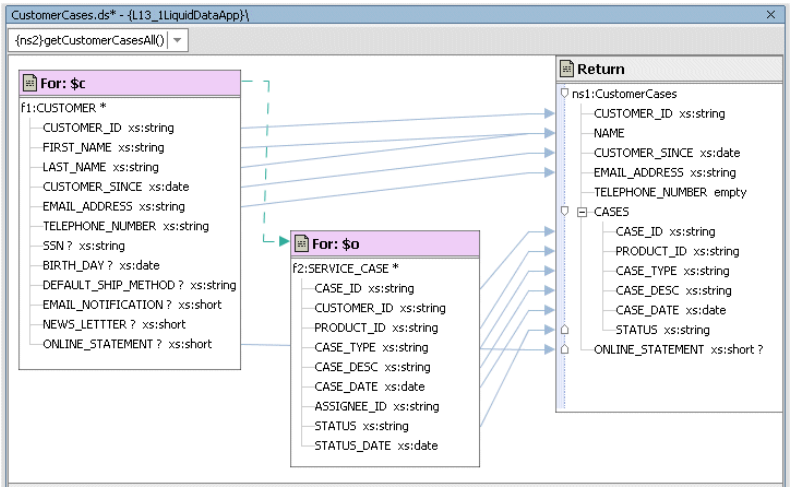
Data Service Transformations

Between acquiring data from back-end systems and presenting that data to clients, a data service often 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 transformational 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 returned in response to service requests.

Figure 3-2 Graphical XQuery Transformation Designer



The Role of XQuery

The internal language of a data service, XQuery (short for XML Query Language), is the emerging standard 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, reliable manner. If potential users of a data source do not know it exists, the data is effectively non-existent. 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.

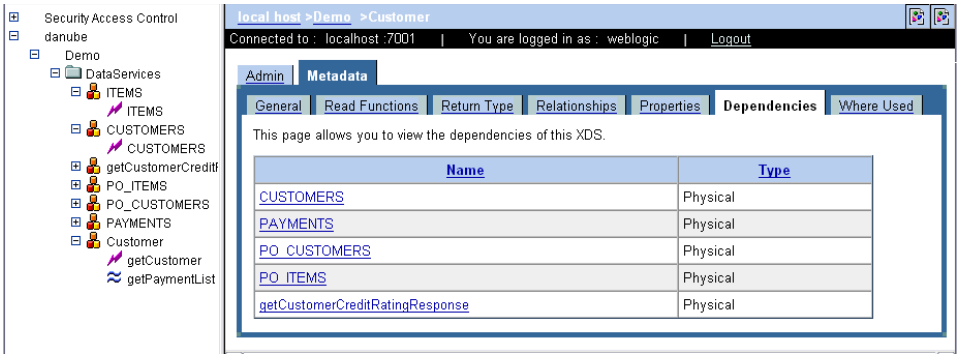
DSP uses a set of data service descriptors (or metadata) to supply enterprise-wide information on data services and their component functions and procedures. The metadata describes the data services — what information they provide and where the information comes from (that is, its lineage).

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. Also, a Metadata API provides programmatic access to the same information.

Figure 3-3 Metadata Browser



CHAPTER 4

Modeling and a Service-Oriented Architecture

This section describes the role of modeling in building a data services layer for your enterprise. The following topics are covered:

- The Role of the Model
- Modeling Data Using XML Types
- Modeling Data Services

The Role of the Model

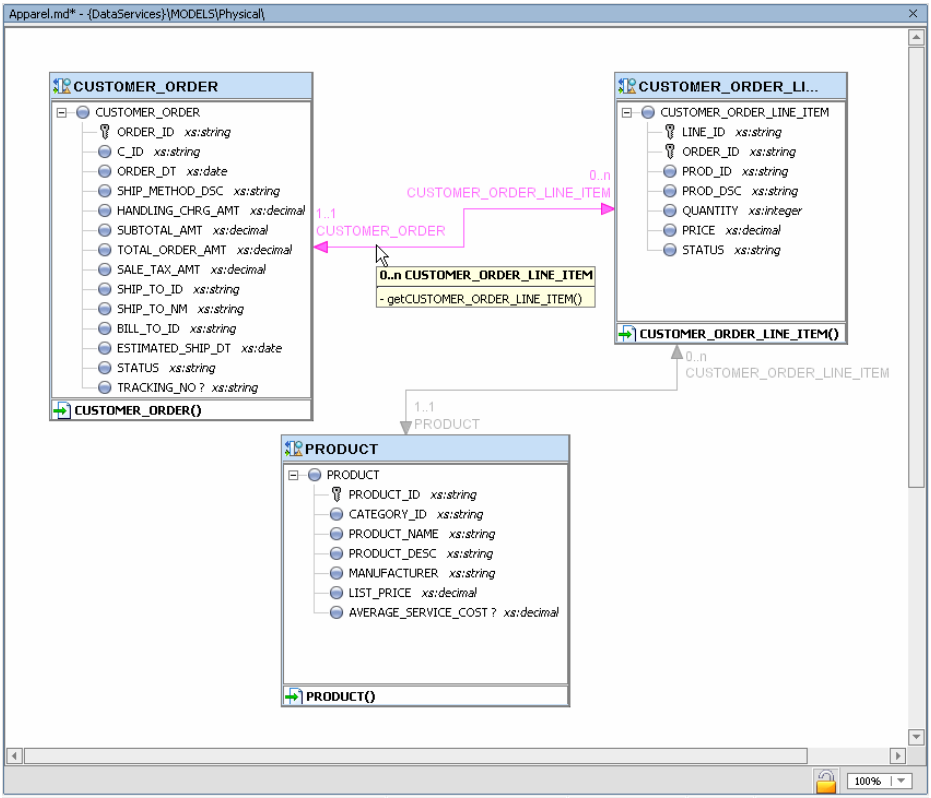
Although never required, modeling can help mitigate complexity of large, intricate data systems. The BEA AquaLogic Data Services Platform provides loosely-coupled modeling services through a visual, representational view of data resources. It can present a high-level view of a complex system or it can break a complex problem into smaller pieces.

Modeling helps you to organize and document data services in all or a portion of your enterprise. 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 often the first step in developing a data services layer. It provides a roadmap for the development of logical data services.

The DSP Modeller (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 Sample Model Diagram



Modeling Data Services

Just as it does for database design, data modeling helps to ensure the optimal design of your data services layer. Modeling helps you analyze and develop data services efficiently. Often it can help turn 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. This data service, in turn, can be immediately represented in one or several models.

Modeling not only represents the data services in your DSP implementation, it also captures the relationships between data services. A relationship is a logical connection between two data services,

such as between a Customers data service and an Orders data service, joined by a customer ID primary key.

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* (or vice-versa).
- Two way, in which data service *a* can navigate to *b* and *b* can navigate to *a*.

Modeling Data Using XML Types

DSP uses the XML Schema language to model data. XML Schema (a standard, XML-based schema definition language) gives DSP 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, called an XML type, with a data shape defined in XML Schema. The schema defines the structure of the data service; that is, its elements, attributes, and rules. For example the following code defines CUSTOMER as having a sequence containing two elements: CUSTOMERID of type integer and CUSTOMERNAME of type string:

```
<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 automatically through the XQuery Editor, or using a schema editor built into the Modeller and Design View. Schemas can also be created externally through tools such as XMLSpy. In any case, a data service cannot be valid without an associated XML schema definition.

CHAPTER 5

Using Service Data Objects (SDO)

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

- Introducing Service Data Objects
- Getting Data Objects
- An Initial Look at SDO Programming
- Data Updates

Introducing Service Data Objects

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 specialized mediators for particular source types, such as for databases. DSP 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 Data Services Platform 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, DSP releases the connections. The client works on a local copy of the data, disconnected from the data source.

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 data 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

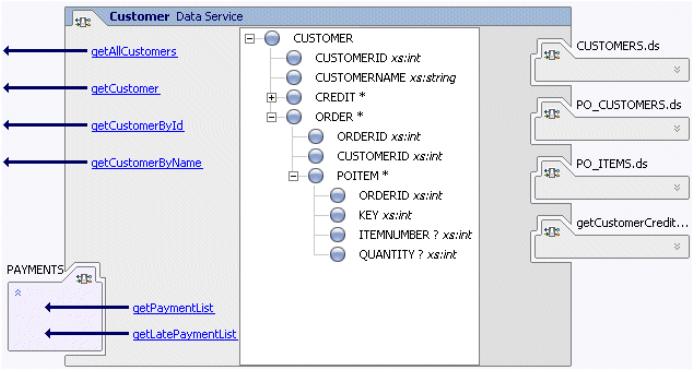
Data Services Platform uses SDO as its client-side programming model. Simply put, this means that when a Java client invokes a data service's read function — either through the DSP Mediator API or through the Workshop Data Service control — it gets a return value in the form of an SDO data object. A 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.

An Initial Look at SDO Programming

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 5-1.

Figure 5-1 Data Service Design View



The data type for the Customer data service is CUSTOMER. A data type is a structured XML document known as a schema. In this example the data type is composed of properties such as customer ID, name, and orders. The data for a CUSTOMER instance comes from four other data services: CUSTOMERS, PO_CUSTOMERS, PO_ITEMS, and getCustomerCredit. Physically each of these is an XQuery document with a `.ds` extension (example: `PO_CUSTOMERS.ds`).

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, readable and easy to maintain, as shown in Listing 5-1.

Listing 5-1 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 four different data sources.
- Modifies a value (customer name).
- Submits the change, which propagates from Data Services Platform through SDO to 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 DSP framework.

Data Updates

As it does for reading data, SDO gives client applications a unified interface for updating data. With DSP, 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.

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 as needed.)

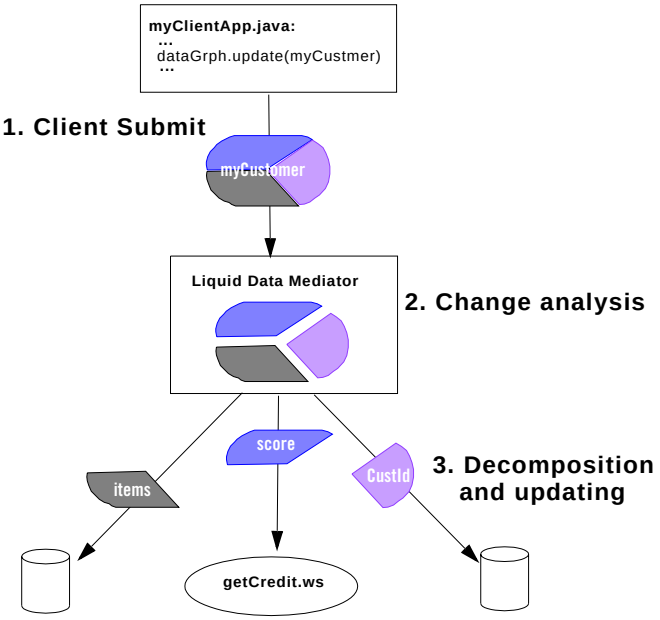
From the data service implementor's point of view, the task of building a library of update-capable data services is considerably eased by the DSP update framework. For relational sources, DSP can often propagate changes to the data sources automatically. For other sources, or to customize relational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

As shown in the Figure 5-2, updates occur through a process in which the requested change is first analyzed to determine, among other things, the lineage of the data. The DSP 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 Data Services Platform *Application Developer's Guide*.

Figure 5-2 Data Services Platform Source Update Sequence



Performance and Caching

This chapter describes Data Services Platform (DSP) performance features. It covers these topics:

Overview

Poor performance can outweigh many of the advantages that an otherwise carefully designed data services deployment provides.

DSP includes a number of configurable settings and internal features to facilitate good performance. With caching, data response times can actually improves upon those provided by native access mechanisms.

Query Optimization

As a declarative language, XQuery also 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 Data Services Platform 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.

The types of query optimization DSP performs include:

- When possible, DSP pushes predicate evaluation, joins, aggregation, and other data manipulation functions to the underlying SQL databases. This minimizes the number of database calls required and reduces the volume of data returned from those calls.
- DSP handles layered data services using a process called *view unfolding*. Instead of retrieving the full data set encompassed by a multi-level function call, DSP retrieves only the data relevant to the actual request.
- DSP 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, DSP stores the results returned from a data service function in a local relational database. When a function call is made again with the same parameters, DSP can respond with the cached copy of the data, thereby avoiding repeated calls to the back-end data sources.

Caching with DSP 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 suits the type of information. If the information is apt to be long-standing without change, the cache can hold that data longer. If information changes frequently, you can have the cache expire as frequently as appropriate.

You can manage the cache by setting time-to-live values and purging the cache through the Data Services Platform Console.

If a client application wants to be sure that it acquires the latest information, it can use the GET_CURRENT_DATA attribute to retrieve information directly from the data source. This operation also automatically updates the function cache, if any.

For information about caching see "Configuring the Results Query Cache" in the Data Services Platform *Administration Guide*.

CHAPTER 7

Securing Enterprise Data

This chapter discusses Data Services Platform (DSP) security features. It covers the following topics:

- Ensuring Data Security
- Securing Data Services Platform Resources
- Understanding Security Policies

Ensuring Data Security

Integrating enterprise data with DSP does not require any compromise in 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, DSP 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 DSP deployment.

The WebLogic Platform provides the security framework that handles authorization based upon information in the context of the user request. By default, DSP 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 Data Service control API, JDBC, or a web service.

Securing Data Services Platform Resources

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

Specifically, secureable resources in DSP include:

- **Applications**. An application-level policy applies to all data services in a DSP 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 DSP 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 DSP 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 Data Services Platform resource. With the WebLogic Authorization module, you can create policies based upon the user's 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 Data Services Platform 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.