Red Hat Web Application Framework 6.0

WAF Developer’s Guide

Red Hat Web Application Framework 6.0: WAF Developer's Guide

Copyright © 2003 by Red Hat, Inc.

Red Hat, Inc.

1801 Varsity Drive
Raleigh NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

rhea-dg-waf-en(EN)-6.0-Print-RHI (2003-11-24T16:20)

Table of Contents

I. Introduction to the WAF Developer’s Guide

Welcome to the Red Hat Web Application Framework Developer’s Guide. The purpose of this guide is to provide an experienced developer with the information, tools, examples and tutorials needed to understand and implement solutions for WAF.

Table of Contents

Chapter 1.

Introduction to the WAF Developer's Guide

The Red Hat Web Application Framework is a platform for writing database-backed web applications in Sun's Java™. Applications leverage Web Application Framework APIs to enable the authoring of persistent structured data and to retrieve and display the data as content. The framework also integrates services such as search, versioning, and permissions into its basic objects, enabling applications to leverage framework services with little or no extra work.

The Web Application Framework domain layer models basic concepts such as users, groups, and permissions and has been proven and refined on hundreds of production deployments. A user interface (UI) framework, UI component library designed for the rapid development and reuse of web user interfaces, and a powerful object-relational persistence engine are also part of the framework.

Please enjoy this guide and report any bugs with the documentation to http://bugzilla.redhat.com/. Please report using the unique component name for this guide:

```
rhea-dg-waf-en(EN)-6.0-Print-RHI (2003-11-24T16:20)
```

1.1. Document Conventions

When you read this manual, you will see that certain words are represented in different fonts, typefaces, sizes, and weights. This highlighting is systematic; different words are represented in the same style to indicate their inclusion in a specific category. The types of words that are represented this way include the following:

`command`

Linux commands (and other operating system commands, when used) are represented this way. This style should indicate to you that you can type the word or phrase on the command line and press [Enter] to invoke a command. Sometimes a command contains words that would be displayed in a different style on their own (such as filenames). In these cases, they are considered to be part of the command, so the entire phrase will be displayed as a command. For example:

Use the `cat testfile` command to view the contents of a file, named `testfile`, in the current working directory.

`filename`

Filenames, directory names, paths, and RPM package names are represented this way. This style should indicate that a particular file or directory exists by that name on your system. Examples:

The `.bashrc` file in your home directory contains bash shell definitions and aliases for your own use.

The `/etc/fstab` file contains information about different system devices and filesystems.

Install the `webalizer` RPM if you want to use a Web server log file analysis program.

application

This style indicates that the program is an end-user application (as opposed to system software). For example:

Use **Mozilla** to browse the Web.

[key]

A key on the keyboard is shown in this style. For example:

To use [Tab] completion, type in a character and then press the [Tab] key. Your terminal will display the list of files in the directory that start with that letter.

[key]-[combination]

A combination of keystrokes is represented in this way. For example:

The [Ctrl]-[Alt]-[Backspace] key combination will exit your graphical session and return you to the graphical login screen or the console.

text found on a GUI interface

A title, word, or phrase found on a GUI interface screen or window will be shown in this style. When you see text shown in this style, it is being used to identify a particular GUI screen or an element on a GUI screen (such as text associated with a checkbox or field). Example:

Select the **Require Password** checkbox if you would like your screensaver to require a password before stopping.

top level of a menu on a GUI screen or window

When you see a word in this style, it indicates that the word is the top level of a pulldown menu. If you click on the word on the GUI screen, the rest of the menu should appear. For example:

Under **File** on a GNOME terminal, you will see the **New Tab** option that allows you to open multiple shell prompts in the same window.

If you need to type in a sequence of commands from a GUI menu, they will be shown like the following example:

Go to **Main Menu Button** (on the Panel) => **Programming** => **Emacs** to start the **Emacs** text editor.

button on a GUI screen or window

This style indicates that the text will be found on a clickable button on a GUI screen. For example:

Click on the **Back** button to return to the webpage you last viewed.

`computer output`

When you see text in this style, it indicates text displayed by the computer on the command line. You will see responses to commands you typed in, error messages, and interactive prompts for your input during scripts or programs shown this way. For example:

Use the `ls` command to display the contents of a directory:

```
$ ls
Desktop                about.html       logs          paulwesterberg.png
Mail                   backupfiles      mail          reports
```

The output returned in response to the command (in this case, the contents of the directory) is shown in this style.

`prompt`

A prompt, which is a computer's way of signifying that it is ready for you to input something, will be shown in this style. Examples:

```
$
```

```
#
```

```
[stephen@maturin stephen]$
```

```
leopard login:
```

`user input`

Text that the user has to type, either on the command line, or into a text box on a GUI screen, is displayed in this style. In the following example, **`text`** is displayed in this style:

To boot your system into the text based installation program, you will need to type in the **`text`** command at the `boot:` prompt.

Additionally, we use several different strategies to draw your attention to certain pieces of information. In order of how critical the information is to your system, these items will be marked as note, tip, important, caution, or a warning. For example:

Note

Remember that Linux is case sensitive. In other words, a rose is not a ROSE is not a rOsE.

Tip

The directory `/usr/share/doc` contains additional documentation for packages installed on your system.

Important

If you modify the DHCP configuration file, the changes will not take effect until you restart the DHCP daemon.

Caution

Do not perform routine tasks as root — use a regular user account unless you need to use the root account for system administration tasks.

Warning

If you choose not to partition manually, a server installation will remove all existing partitions on all installed hard drives. Do not choose this installation class unless you are sure you have no data you need to save.

1.2. Code Presentation Conventions

In addition to the standard document conventions covered in Section 1.1 *Document Conventions*, there are some additional conventions related specifically to discussing source code:

`classname`

This is the name of a class in an object-oriented (*OO*) programming language. For example, the class `com.arsdigita.categorization.CategoryTreeNode`.

`methodname`

This is the name of a method in an OO programming language, e.g. the method `getBaseDataObjectType`.

`function`

The name of a function or subroutine, as in a programming language. For example, the function `SecurityLogger.warn()`.

`varname`

The name of a variable. For example, the variable `BASE_DATA_OBJECT_TYPE`.

`option`

An option for a software command or `Method`. For example, a user has been granted `read` privileges on an object.

returnvalue

The value returned by a function. For example, a method returns null.

replaceable

Content that may, must or will be replaced by the user or a program. For example, the code is commented with `NOTE(`*n*`)`, where *n* is the number of the `NOTE`.

programlisting

A literal listing of all or part of a program:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.PermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;

OID acsObject = new OID("example.MyACSObject",
new BigDecimal(50));

OID party = new OID("com.arsdigita.kernel.Group", new BigDecimal(5));


PermissionDescriptor perm =
new PermissionDescriptor(PrivilegeDescriptor.READ,
acsObject, party);

PermissionService.grantPermission(perm);
```

firstterm

The first occurrence of a term, such as the first time we introduce a *bulletin-board* and note its abbreviated form, *bboard*.

1.3. Assumptions About WAF Developers

This manual assumes that the reader is familiar with the Java programming language, HTML, and relational databases. Familiarity with the J2EE Servlet and JSP specifications, XML, and XSLT are also helpful. An understanding of the UML and basic object-relational mapping concepts will help the reader understand the persistence system. For more information, see Section 8.1 *Developer Education*.

II. WAF Concepts

This section covers the concepts of WAF. The intention is to provide both a very high-level architectural view and a closer review of the individual components.

Table of Contents

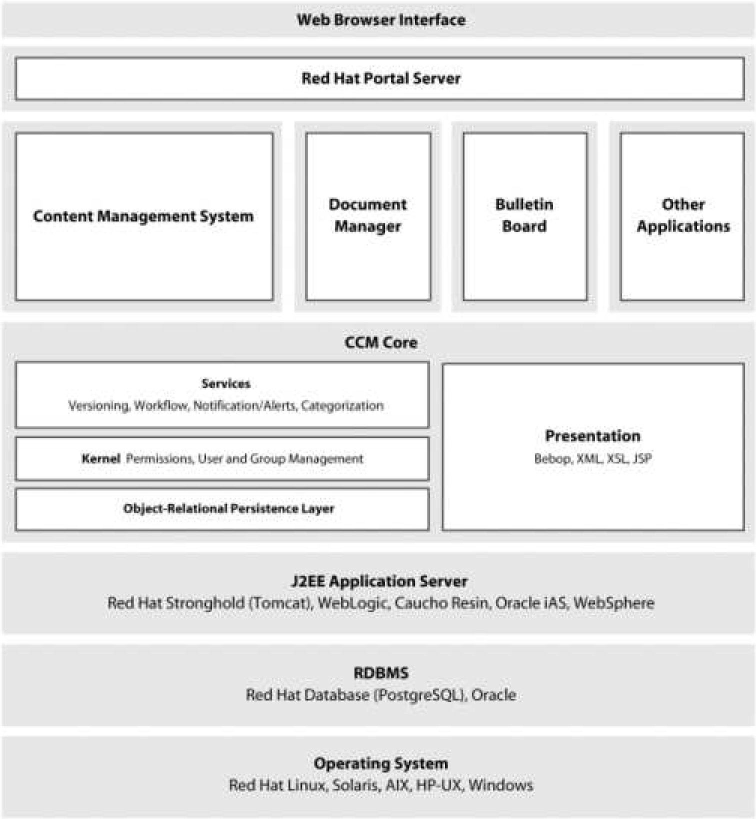
Chapter 2.

WAF Overview

This chapter is an overview of the Web Application Framework architecture. This high-level viewpoint is especially useful for gaining a good understanding of how WAF works. It is written with both the technical developer and the technical manager/team leader in mind.

WAF is a *web application development framework*. Some of the web applications that have been developed using WAF include Red Hat Content Management System and Red Hat Portal Server. WAF runs in any standards-compliant servlet container. For more details about system requirements, see the *Red Hat Web Application Framework Installation Guide*.

Figure 2-1 describes the architecture of WAF from a high-level perspective.



Web Browser Interface
Red Hat Portal Server
Content Management System
Document Manager
Bulletin Board
Other Applications
CCM Core
Services
Versioning, Workflow, Notification/Alerts, Categorization
Kernel Permissions, User and Group Management
Object-Relational Persistence Layer
Presentation
Bebop, XML, XSL, JSP
J2EE Application Server
Red Hat Stronghold (Tomcat), WebLogic, Caucho Resin, Oracle iAS, WebSphere
RDBMS
Red Hat Database (PostgreSQL), Oracle
Operating System
Red Hat Linux, Solaris, AIX, HP-UX, Windows

Figure 2-1. Basic Configuration

2.1. General Architecture

The WAF architecture described in Figure 2-1 follows the standard n-tier design pattern, with separate presentation, domain (business logic), data, and data model layers. Web applications built on WAF also follow the same n-tier design patter, leveraging the infrastructure provided by WAF.

2.1.1. The Layers

The four layers in the WAF architecture are:

- Presentation Layer (UI) — presents information to the user
- Domain Layer (Business Logic) — encapsulates business logic
- Data Layer — stores and retrieves data
- Data Model — stores data in a structured, format in a durable fashion

These layers are shown in Figure 2-2:

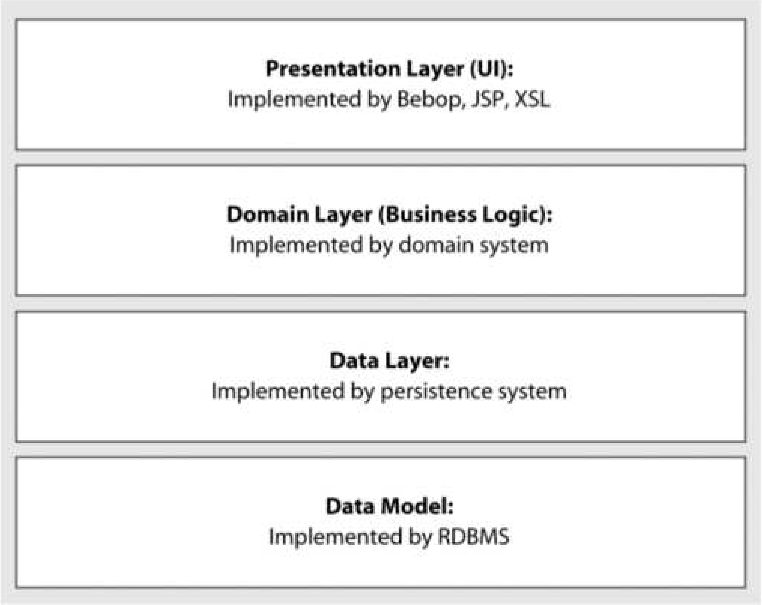

Presentation Layer (UI):
Implemented by Bebop, JSP, XSL
Domain Layer (Business Logic):
Implemented by domain system
Data Layer:
Implemented by persistence system
Data Model:
Implemented by RDBMS

Figure 2-2. Basic Configuration

2.1.1.1. Presentation Layer

The Presentation Layer is responsible for presenting information to the end user. The presentation layer accepts processed, structured data from the domain layer and is responsible for styling the data appropriately and delivering the content in a format appropriate for the end user.

2.1.1.2. Domain Layer

The Domain Layer contains *Domain Objects*, which are abstractions of entities that exist in the business domain, for example, Party, Person, Group, Company, Department, Team, Product, Order, Line Item.

2.1.1.3. Data Layer

The Data Layer contains two very important components:

Data Objects

> Provide read and/or write access to the persistent properties of Domain Objects.

Persistence Metadata

> Describes a Domain Object e.g., its name, its properties, and how each property is mapped into the Data Storage layer. This description is written using the *Persistence Definition Language* (PDL) format. PDL was designed to be used specifically in WAF. PDL is discussed in more detail in Section 3.3 *Persistence Definition Language (PDL)*.

2.1.1.4. Data Storage Layer

The data storage layer Contains the mechanism(s) employed for storing data persistently. This is typically a relational database (RDBMS) such as Oracle9i™ database or PostgreSQL, combined with a data model. It may also include other mechanisms, such as an LDAP directory or filesystem.

2.2. Features

In his seminal book *Analysis Patterns*, Martin Fowler writes that a framework "... should be applicable across a large domain and be based on an effective conceptual model of that domain" (p. 11). Accordingly, the Web Application Framework defines a set of Domain Objects that are encountered in the problem domain of most WAF applications. This object model further subdivides into two categories: *kernel* and *services*.

In addition to kernel and services, WAF includes other features that facilitate building database-backed web applications: infrastructure, persistence, presentation, and web.

2.2.1. Kernel

The kernel provides all the business logic provided by WAF, namely, business logic that is essential to building a web applications. Kernel provides domain objects that represent users, groups, and permissions.

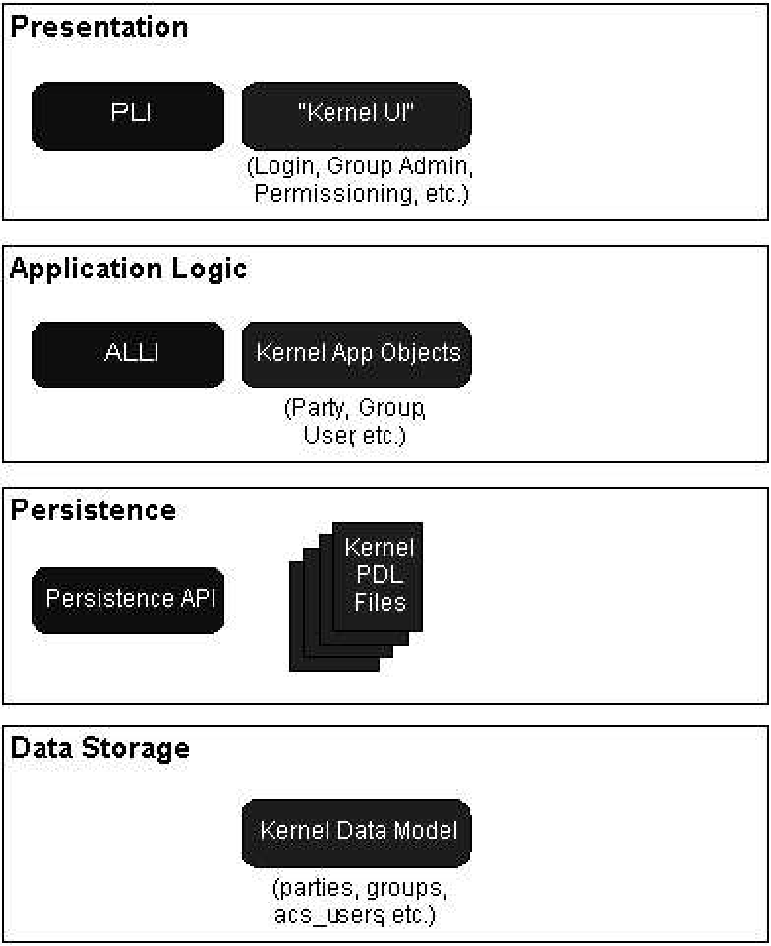


Figure 2-3. Basic Configuration

2.2.2. Services

Services are building blocks that address generic requirements common to most WAF applications. Each requirement defines a set of related Domain Objects, for example, *Versioning*, *Workflow*, and *Categorization*:

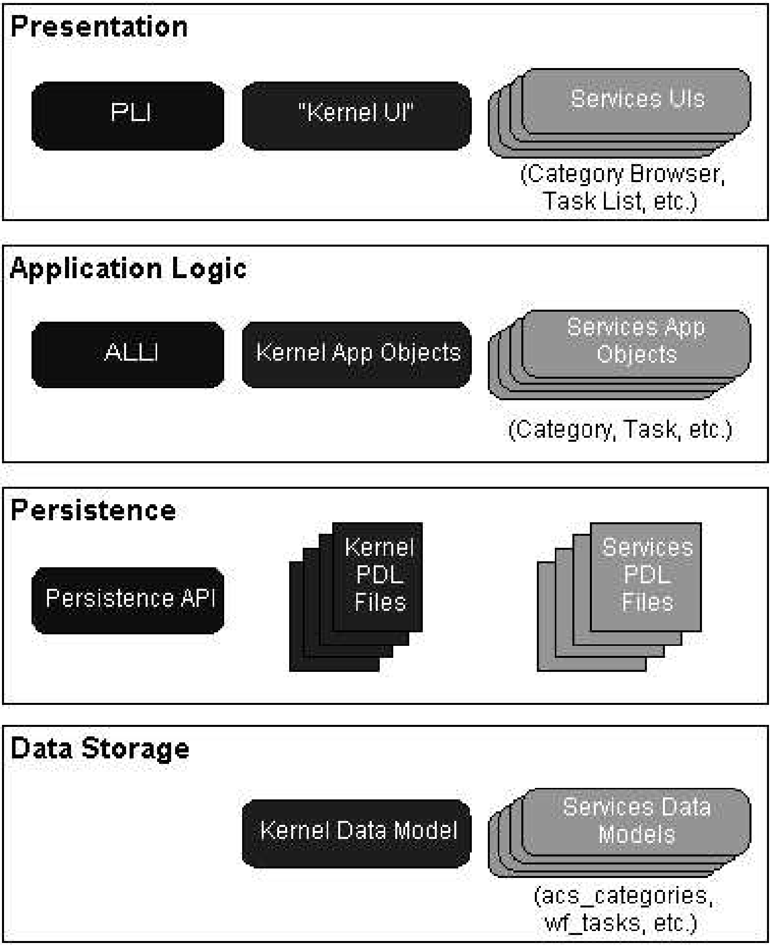


Figure 2-4. Basic Configuration

As shown in Figure 2-4, all services follow the n-tier design pattern discussed in Section 2.1 *General Architecture*, providing:

- A user interface for interacting with the Framework's Domain Objects.
- Domain logic.
- The metadata (PDL) required for persistence of the Framework's Domain Objects.
- Data storage as appropriate.

Services are discussed in more detail in Chapter 5 *WAF Component: Services*.

2.2.3. Infrastructure

Infrastructure contains software to support the mechanics of application building at each layer of the architecture (for example, serving page requests, styling the user interface, logging, specifying metadata, storing data, etc.).

Note

This infrastructure exists independently from any specific problem domain and generally does not depend on other WAF systems.

2.2.4. Persistence

Persistence handles the storage and retrieval of all information in WAF applications via Data Objects. Data Objects are implemented as a Java class library that supports *CRUD* (create, read, update, and delete) operations for any type of Data Object. This is done through a set of generic interfaces: `DataObject`, `DataCollection`, and `DataAssociation`. Persistence is discussed in detail in Chapter 3 *WAF Component: Persistence*.

2.2.5. Presentation

Presentation is responsible for presenting data in a structured format to the end user. WAF provides three basic systems for presentation: Bebop, a web user interface component framework modeled after Java Swing; Java Server Pages (JSP); and eXtensible Stylesheet Language (XSL). Presentation is discussed in detail in Chapter 6 *WAF Component: Presentation*.

2.2.6. Web

The Web component of WAF makes the persistent data and domain logic of your application available to others over protocols such as HTTP. It integrates the Java Servlet API and the kernel and persistence components of WAF. For more information, see Chapter 7 *WAF Component: Web*.

2.3. Applications

Each WAF application adds code and other assets (stylesheets and PDL files) to each layer of the architecture. The result is a complete application:

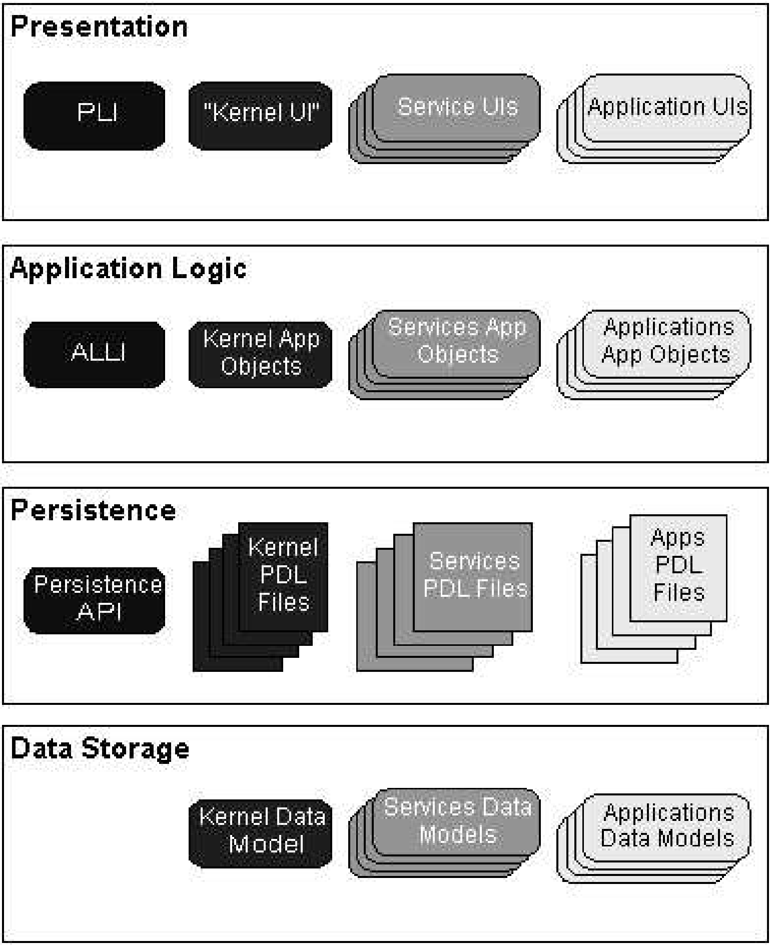


Figure 2-5. Basic Configuration

Again, each application follows the same n-tier design pattern, and builds upon the kernel and services provided by WAF.

Chapter 3.
WAF Component: Persistence

This chapter discusses the persistence layer in the overall Web Application Framework. This was originally discussed in Section 2.2.4 *Persistence*. You can find persistence tutorials in Chapter 9 *Persistence Tutorial*.

3.1. Persistence Overview

The storage and retrieval of persistent data is a common requirement of business applications. WAF provides a *persistence* layer as a generic solution to this requirement.

The WAF persistence layer allows naturally written Java classes to be persisted to and queried from a relational database. This is more than simply a method to save Java classes within the database. Developers can use the full range of standard object-oriented modeling techniques, such as inheritance, interfaces, polymorphism, associations, and composition when writing their Java classes.

WAF persistence is an example of an *object-relational mapping layer*. This chapter will discuss some of the concepts required to understand and use WAF persistence. Examples and tutorials are detailed in Chapter 9 *Persistence Tutorial*.

3.2. Object-Relational Mapping

An object-relational mapping layer allows developers to use both relational and object modeling in the development of their applications.

- Relational Modeling — a powerful tool for modeling *knowledge*. Modern relational databases utilized with well designed relational schemas provide guaranteed data integrity, support for multiple concurrent transactions, and support for fast and flexible querying.
- Object modeling — a powerful tool for modeling *behavior*. Object modeling is useful when coupled with object-oriented (OO) languages and design patterns. OO languages are a common choice for rapid application development and maintenance.

  An object-relational mapping layer can automatically translate operations on an object model into operations on a relational model and vice versa. This is accomplished through the use of mapping metadata that relates persistent attributes and associations in the object model with tables and columns in the relational model.

3.3. Persistence Definition Language (PDL)

Because of the flexibility of both relational and object models, it is possible to map a given object model to many different relational models and to map a single relational model to many different object models.

WAF persistence uses a modeling and mapping language called *Persistence Definition Language* (*PDL*) to allow developers to describe their object model, their relational model, and how the two are mapped.

The metadata described in PDL allows the relational engine to efficiently persist changes to the object model and to construct efficient SQL queries for performing object-level reads. PDL also allows developers to define their own custom queries and SQL operations to do specialized querying and updating of the relational model.

For a listing of PDL terms, see Appendix D *PDL Syntax*. Examples of PDL usage can be found in Chapter 9 *Persistence Tutorial*.

3.4. Persistence and Domain APIs (DataObject, DataAssociation, DataCollection, DomainObjects)

Once an object model is described in PDL, the persistent state associated with it may be directly accessed and manipulated by Java code through use of the `DataObject`, `DataAssociation`, and `DataCollection` classes that are part of the WAF persistence API.

Developers can then build upon this API in order to add the behavior required to implement their application. This is generally done by extending `DomainObject`, an abstract base class that encapsulates a `DataObject` and uses it to provide persistence capabilities to any derived classes.

3.5. Session and Transaction Management

Domain classes, once written, can be used similarly to any other Java code, with one exception — code that involves domain objects must be executed within the context of both a session and a transaction.

The session identifies which database that changes are persisted to and queried from, and the transaction allows atomic updates of the database. Setup of the current session and transaction is a simple process that can usually be centrally handled in an application.

Within WAF, session and transaction management is automatically handled by the `BaseServlet` class. The transaction model used by WAF persistence is a simple extension of common relational database transaction semantics:

- Transactions cannot be nested.
- Once a transaction begins it must either be committed or rolled back.
- Transactions must be rolled back after an error occurs.

Chapter 4.
WAF Component: Kernel

This chapter discusses the kernel layer, which provides services used by several parts of the WAF system. This component was initially explained in Section 2.2.1 *Kernel*. This discussion focuses on the various parts of the kernel component which a developer will need in building for WAF.

For information on how to utilize the kernel in WAF applications, see Chapter 10 *Kernel Tutorial*.

4.1. Users and Groups

Applications have *users*. A user is a person who uses an application to accomplish some purpose. In order to serve the needs of users, applications store information about the users. This information is used to personalize content, check the permission of a user initiated operation, and to provide information about a user to other users.

Users are organized into *groups*. The users in a groups are said to be members of the group. Groups can also be members of groups. A group exists so that several users and groups can be collectively identified as an entity.

Groups and users are the two types of *parties*. This parties system is one of the pieces of the kernel. The ability to refer to an entity that may be a group or a user provides flexibility to application authors in writing data models. An example is the definition of a group itself: 1 or more parties. This definition is recursive, but that recursion reflects the flexibility of having a party type.

4.2. ACSObject

`Party`, `Group`, and `User` are all subtypes of `ACSObject`. Its purpose is to serve as a base class for use in common object-level services, such as categorization and permissioning. As such, subclasses are often types that users directly interact with such as `Workflow` or `Category`. The `ACSObject` object type also has generic attributes such as a unique id and a displayName.

The `ACSObject` object type and class are abstract. In order to differentiate among subtypes there are attributes objectType and defaultDomainClass. These attributes are used to instantiate the appropriate Java domain class to wrap data objects that are instances of subtypes of `ACSObject`.

4.3. Permissions

The goal of the permissions system is to provide generic means to both programmers and site administrators to check, grant, or revoke permissions via a consistent interface. For example, an application developer might decide that viewing a certain set of pages within the application is an operation to be individually granted or revoked from a user. It's expected that the permissions system will be heavily used in production - almost every page will make at least one permissions API call, and some will make several.

The permissions systems deals with three kinds of objects: `ACSObject`, `Party`, `Privilege`. The system maintains a set of grants of the form `Party p` has `Privilege priv` on `ACSObject obj`. The privileges represent actions that are performed on objects.

There are generic privileges that are always defined: `read`, `write`, `edit`, `admin`, `create`, and `delete`. In addition, applications can define custom privileges, for example `categorize`.

The permissions system is used to answer questions such as:

- What parties have a particular privilege on a particular object?
- What privileges does a particular party have on a particular object?
- On what objects does a particular party have a particular privilege?

Much of the power of the permissions system comes from the flexibility of the model. The assertions described above are not the only input. Each privilege can imply some set of privileges. In addition, each `ACSObject` can have a security context from which it inherits privilege grants. This security context is just another `ACSObject`. Groups provide a way of aggregating users and the permissions system pays attention to group membership.

The power of the permissions system comes from the interaction of these 3 hierarchies: group membership, security context, and privilege implication. Taking these 3 together, the assertion that `p` has privilege `priv` on object `obj` means that *p* is a party or a member of a party (at any depth) that was granted `priv` or a privilege implying privilege *priv* on object *obj* or any parent of *obj* in the security context hierarchy.

4.4. Kernel Resources

`Resource` is a base class in the kernel that represents a user-accessible resource that may serve as a data container. Resources are organized into a parent-child hierarchy. By default, resources set their security context to their parent resource.

Currently, the two subclasses of resource are `Application` (which is web accessible) and `Portlet` (which is accessible through Red Hat Portal Server). All `Resources` have a `ResourceType` which is uniquely determined by the specific object type of the `Resource`. `ResourceType` has metadata that is used by some generic navigation and administrative interfaces.

Chapter 5.
WAF Component: Services

WAF provides a number of generic *services* that can be used in developing a WAF application. Services leverage the business logic provided by the kernel, and provide specialized domain APIs to solve specific, common business problems. Examples of services include notification, workflow, and versioning.

Services were originally discussed in Section 2.2.2 *Services*. Implementation examples are covered in Chapter 11 *Services Tutorials*.

5.1. Auditing Service

The **auditing** service can track when and by whom an object was created and last modified for an `ACSObject`. This tracking is optional. The `AuditedACSObject` class provides methods for accessing the auditing information.

5.2. Categorization Service

One common WAF application implementation is providing online access to large amounts of information. To make it easier for users to find relevant information, the **categorization** service is often employed. It allows various pieces of information such as articles, forum postings, and so on, to be categorized.

Categories are organized in a tree-like structure that is often called *taxonomy* or *ontology*. See Example 5-1.

- Entertainment
  - Music
  - Movies
  - Television Shows

- Education
  - Literacy
  - Testing
  - Home Schooling
  - Theories

- Sports
  - Scores
  - Venues
  - Kinds of sports
    - Basketball
    - Chess
    - Soccer

Example 5-1. Sample Taxonomy

The **categorization** service provides:

- The ability to set up multiple independent taxonomies.
- A system for importing predefined taxonomies from XML files.
- A user interface for managing categories.
- An API for categorizing objects.

5.3. Formbuilder Service

Most applications built with WAF allow users to enter information into the system through a web browser. A user fills out a number of fields in a form and clicks the **Submit** button. The user's input is validated and stored for later retrieval and display.

Building complex applications with this sort of user interaction usually requires programmer involvement. For example, if you are building a web browser based email client, you need to write code that knows how to send and retrieve email messages, how to organize messages in folders, and so forth.

There are another class of applications that do not require any special handling of the input data. What is required is the ability to enter the data in a structured way, and retrieve it later for display, reporting, printing and so forth..

For example, a group of users wants to input and store data about competing products — they want to record the product name, the company's name, a short product description, and the date it was first introduced to the market. Rather than requiring programming for this straightforward functionality, WAF provides the **formbuilder** service.

The **formbuilder** service enables non-technical users to build such applications via a web interface, without any programmer intervention. The **formbuilder** service provides a variety of *widgets* to choose from, such as a date widget which makes the part of the HTML form that allows you to specify a date. Validation code is automatically attached to the date widget to ensure that only valid dates are entered. Various widgets are provided for entering simple strings, numbers, etc.

Once the user has defined the required fields for their custom new document type, the system automatically allocates structured storage in the database. From this point on, other users may enter documents that require the specified fields to be filled in. Such documents are displayed in the tabular format:

Product name	gizmo
Company name	Acme, Inc.
Description	This gizmo slices, dices, and juliennes.
Date introduced	July 22, 1999

Table 5-1. Sample structured document

5.4. Globalization Service

A globalized application is one that tries to equally accommodate readers of different languages and dialects. Elements of the application's user interface are shown in the user's preferred language, if it is supported. For example, a link to the categories page is displayed as **Categories** for English speakers and **Kategorien** for German speakers.

A partial list of globalization and localization facilities supported by the Java language includes:

- The notion of *locale* (http://java.sun.com/j2se/1.3/docs/api/java/util/Locale.html).
- *Resource bundles* (http://java.sun.com/j2se/1.3/docs/api/java/util/ResourceBundle.html) allow you to organize and maintain locale-specific objects, such as strings to be shown as button labels, e.g. **Categories** and **Kategorien**.
- Locale-specific date format (http://java.sun.com/j2se/1.3/docs/api/java/text/DateFormat.html\ #getDateInstance(int,java.util.Locale) — "\" has been inserted to show where the line was broken for printing purposes; you will need to put both halves together again in e.g. your Web browser URL window).
- Locale-specific *collation* (http://java.sun.com/j2se/1.3/docs/api/java/text/Collator.html) of sequences of strings.
- Miscellaneous other facilities.

The WAF **globalization** service provides classes and methods that can be roughly divided into the following three categories.

1. Convenient wrappers around most-frequently used Java globalization APIs, such as *resource bundles*.
2. Means of configuring the list of supported languages, locales, and *character sets*.
3. A mechanism for locale and character set negotiation.

   Most modern browsers allow users to select their preferred language or a ranked sequence of preferred languages. The **globalization** service looks at the user's preferred language and locale, as reported by the user's browser, and compares them to the list of languages and locales that the system is configured to support.

For example, a user can configure the preferred first language to be German ("de") and the preferred second language to be British English ("en_GB"). If the system is configured to support both of the languages, then the **globalization** service will choose "de" as the preferred locale. If the system has been configured to support British English, but not German, then the **globalization** service will choose "en_GB" as the preferred locale. If the system does not support either of the configured languages, then the service falls back to a default locale which could be "en_US" (US English).

Once the desired locale is configured, a supported character set must be selected. For example, pages in German can be displayed using the ISO-8859-1 [1], UTF-8, or UTF-16 character sets. The user may express a preference by configuring their browser. The user's preference must then be reconciled with the list of character sets that the system has been configured to support.

This example of locale and character set negotiation is simplified for this discussion, but it illustrates the fact that correct negotiation is non-trivial. The **globalization** service relieves the developer of the burden of negotiating the preferred locale and character set correctly.

The WAF **globalization** service provides a set of high-level APIs based on the standard Java globalization facilities that handle issues such as language negotiation and localization of static strings. These APIs complement the basic globalization infrastructure provided by Java (i.e., `ResourceBundles` and property files are still used in WAF) and are specifically designed for use by web applications built with WAF.

5.5. Mail Service

The **mail** service acts as a **Mail Transport Agent** (**MTA**) for other WAF applications. It provides a simple mechanism to send plain or rich text messages to any entity with a valid email address. This service is not typically accessed directly, but rather is used as a building block for higher-level services such as **notification**. This service has no dependencies on WAF and exists primarily as a wrapper for the more cumbersome JavaMail API.

When should you use the **mail** service directly? If you are writing an application that sends simple plain-text alerts to users, and you do not require those alerts to be recorded in the database, then it makes sense to use the **mail** service. Also, if you are dealing with raw email addresses rather than WAF parties (users and groups), you probably want to use the **mail** service directly. Otherwise, you should consider using the higher-level **notification** service (see Section 5.7 *Notification Service*).

5.6. Messaging Service

The WAF **messaging** service is intended to be used as a building block for *messaging applications*, which are WAF applications that use messages to represent various types of communication between users.

Messages can represent an email from one user to another, or a bulletin-board post, or a comment on another object in the system. A message can store attachments such as images or other files which are relevant to the message. Each attachment can be of any arbitrary MIME type.

The implementation follows the RFC822 [2] standard, which has been deprecated by RFC2822 [3].

The `Message` class models a persistent text message with optional attachments:

[1] http://www.htmlhelp.com/reference/charset/
[2] (http://www.rfc-editor.org/rfc/rfc822.txt
[3] http://www.rfc-editor.org/rfc/rfc2822.txt

1. The message body should have a MIME type of `text/plain` or `text/html`.
2. Each attachment to the message can have an arbitrary MIME type and format. Messages can also refer to any `ACSObject` on the system.

`MessageParts` represent a message part (an attachment) in the sense of a multipart MIME message. Each part has some content represented as an arbitrary block of bytes and a MIME type that identifies the format of the content.

A `ThreadedMessage` is an extension of `Message` that allows messages to be organized into discussion threads with a tree structure. Messages at the first level are referred to as *root* messages.

Warning

> If you delete the root message in a `ThreadedMessage` tree, all of its children are deleted.

5.7. Notification Service

The **notification** service is built on top of the **mail** and **messaging** services. **Notification** provides persistent storage of messages, digest processing, automatic error recovery, and expansion of groups into individual recipients.

Tip

> When looking for a service to do alerting, consider the differences between the **notification** service and the lower-level **mail** service. The **mail** API is capable of handling simple alerts in plain-text. If you need added features, **notification** provides these.

5.8. Portal Service

The **portal** service provides basic domain classes that represent portlets and portals (a set of zero or more portlets). The **portal** service is used by the Red Hat Portal Server and any WAF application that provides portlets for displaying specific content to the end user.

For more information about utilizing **portal** services, see the *Red Hat Portal Server Developer's Guide*.

5.9. Search Service

The **search** service provides APIs for indexing and searching content. Two search engines are supported by WAF: Oracle *inter*Media™ and Lucene. *inter*Media requires an Oracle database, while Lucene is an open-source, database-independent search engine (see http://jakarta.apache.org/lucene/ for more information).

Objects that should be indexed register themselves with a `SearchableObserver`, which indexes the object whenever the object is persisted. If you extend WAF with custom content types, making the custom content searchable is a simple matter of implementing `com.arsdigita.lucene.Adapter`.

5.10. Workflow Service

Workflows allow specialized members of a group to collaborate using a standard process. Developers can define new workflows using the **workflow** service.

A workflow contains a set of tasks, each of which may depend on one or more other tasks. A task is a single unit of work. It can be enabled, disabled, or finished. A task may be assigned to a user or group.

Section 11.4.1 *Simple Workflow* describes the components of the simple **workflow** and how to create workflows and assign them to users and groups.

5.11. Versioning Service

The goal of the the **versioning** system is to provide facilities for versioning data objects. This involves recording information about changes made to the data in order to be able to:

1. Roll back to an earlier point in a data object's history.
2. Undelete a data object (corollary of ability to perform rollback).
3. Compute the difference between any two versions of the data object.

To illustrate the last point, suppose you want to know the difference between the Jan 12 and Apr 19 versions of an article. The **versioning** service can tell you that the article's title changed from *Ravioli is better than spaghetti* to *Macaroni is better than spaghetti*, and its by-line changed from Guy Lewis Steel to Guy Lewis Steele, Jr.

The service does *not* provide redundant storage. If you delete the versioned article in a context where the **versioning** service is not available (either because it is turned off, or because you are manipulating the database directly through JDBC or command-line client), there is no way for the **versioning** service to restore it. It is not a substitute for regular database backups.

The service is provided transparently. There is very little that a developer has to do in order to make their data versioned. To gain conceptual understanding of the **versioning** service, one must be familiar with the key concepts of persistence: *data objects* and *PDL* (see Section 3.4 *Persistence and Domain APIs (DataObject, DataAssociation, DataCollection, DomainObjects)* and Section 3.3 *Persistence Definition Language (PDL)*). For a more detailed explanation of **versioning**, please refer to Section 11.5 *Versioning Tutorial*.

Chapter 6. WAF Component: Presentation

Information in the database ultimately needs to be presented to the user for viewing and manipulating. The Web Application Framework provides a variety of systems to display and style information. Each system has its strengths and weaknesses, and is designed for specific situations.

The following sections discuss the various methods available for presentation under WAF. Each approach is designed to be complementary to the other approaches. Each approach is intended to be used in different situations; the following sections discuss when each system should be used.

Presentation in the context of the Web Application Framework was originally discussed in Section 2.2.5 *Presentation*. For information about implementing Bebop, the presentation method unique to Red Hat WAF, see Chapter 12 *Presentation (Bebop) Tutorial*. Further information and tutorials about implementing the open standards presentation methods in WAF can be found through Chapter 14 *References* and on the Web.

6.1. Overview of Presentation Standards

This section discusses the standards used in WAF for textual presentation of structured data. In addition, the standard Java APIs for parsing and manipulating structured data are reviewed. This information is provided to set a common understanding of which tools are incorporated into WAF and how they are used.

- *HyperText Markup Language* (HTML) is the predominant web standard for rendering documents. In WAF, HTML is produced either by JSPs or by transforming the XML generated by Bebop pages. WAF applications use HTML in conjunction with the following technologies to render and style an application's output.

  XHTML is the newest version of the HTML specification, adapted to conform to the standards of XML.

- *Cascading Style Sheets* (CSS) is a standard mechanism for adding presentation information (style and layout) to HTML documents. See Section 6.2 *CSS and XSLT* for more information. WAF uses CSS to style the HTML produced in JSPs and Bebop pages.
- *eXtensible Markup Language* (XML) is a standard format for representing arbitrary tree-structured data in textual format. XML documents appear similar to HTML because both are derived from *Standard Generalized Markup Language* (SGML). WAF uses XML in its Bebop component UI framework to describe a UI before it is styled.
- *Document Object Model* (DOM) is an API (available in both C++ and Java) for manipulating an XML document as a tree structure in memory. The heart of the DOM is the `Node` object, which represents an element or attribute; a `Document` is the top-level object that represents an entire XML document. WAF uses DOM to generate Bebop presentation XML. Bebop uses the J2EE *JAXP* API to manipulate DOM documents.
- *eXtensible Stylesheet Language Transformations* (XSLT) is a language for specifying rules to transform an XML document into some other kind of output. It is most often used for rendering XML-formatted data into an XML presentation format such as XHTML. WAF uses XSLT to convert Bebop source XML into HTML for the client browser. Bebop uses the J2EE *TrAX* API to plug into and use one of several available transformer implementations.

  See Section 6.2 *CSS and XSLT* for more information about XSLT.

- *JavaServer Pages* (JSP) is a J2EE standard for scripting dynamic web pages. JSPs typically produce HTML, though they can generate any XML markup. WAF supports the use of JSPs as a primary means of writing web UIs.

6.2. CSS and XSLT

WAF uses two technologies together, CSS and XSLT, to add style to logical (i.e. all form, no style) markup.

- CSS is used to control the style properties of HTML documents. CSS has broad browser support and fine-grained control of properties such as color, borders and padding, positioning, and typeface. Unlike XSLT, CSS cannot change the *structure* of the document it is styling. It cannot, for instance, turn a bulleted list into a table.

  Bebop uses CSS in the HTML that it produces when it transforms source Bebop XML. JSPs may also use CSS. Consistent use of CSS under both regimes will give your site a more consistent look and feel.

- XSLT is used to transform the structure of an input document into a new and different structure on output. This is very useful for creating modeling and then rendering UI concepts that do not exist in HTML. A tabbed pane, for instance, is not part of the HTML standard, but a block of HTML that functions as a tabbed pane can be produced by transforming source XML describing tabs into a concrete HTML rendering.

  Bebop, by design, produces XML that is not fit for direct consumption by a browser. Instead, it is transformed first using XSLT. JSPs, by contrast, typically produce HTML directly and so will not generally make use of XSLT. This is not, however, a rule. When it makes sense, a JSP author may wish to produce XML and transform it using XSLT.

6.2.1. Integrating XSLT with WAF

WAF integrates XSL stylesheets in the following way:

- It provides a default stylesheet as an integral part of each WAF package, so that each package ships with some way to style the content it generates.
- It allows the defaults for any package to be overridden when that package is invoked from a particular URL pattern. This allows for co-branding, etc.
- It allows for special-usage page scripts (for example, JSP/XSL pairs) which can import site-wide styling rules.

Note

> WAF infrastructure depends on the Xalan-J XSLT engine from the Apache XML project. In turn, this engine depends on the Xerces XML parser, also from Apache. Although it is theoretically possible to use a different XML parser with Xalan, this process hasn't been thoroughly tested. It is also possible to use a different XSLT engine, such as Saxon. WAF contains no explicit dependency on any vendor's XSLT engine.

This flexibility comes from the `PresentationManager` interface, whose one method `servePage` obtains a transformer for the current request, applies it to an input document, and serves the transformed output to the response output stream. A WAF application's dispatcher can use the provided

`BasePresentationManager` or swap it out with its own to use a different algorithm for choosing a stylesheet for the current request.

XSLT integration into WAF is accomplished by associating stylesheet documents with both WAF packages and *site nodes*. A default stylesheet is associated with each WAF package, and the package defaults can be overridden within a particular URL prefix's scope. A site node represents a node in the webapp's URL tree; each site node, combined with its ancestors, generates a URL prefix. For the purposes of this discussion, *subsite* is defined to mean any site node that contains child site nodes — that is, a directory that contains subdirectories.

Because XSLT templates are also XML documents, they can be manipulated and composed dynamically. A `PresentationManager` could compute a new XSLT stylesheet on the fly, taking fragments from files on disk or in the database.

For more information about using XSLT with WAF, see Section 12.1 *Calling XSLT from a WAF Application*.

6.2.2. WAF Templating Package

The WAF Templating package (com.arsdigita.templating) serves as a repository for all classes relating to the XSL templating.

6.2.2.1. Template Resolution

This section discusses the process of resolving the primary, top level stylesheet for transforming an application's XML DOM into an HTML page.

6.2.2.1.1. Background

The original WAF method for resolving top level stylesheets required the application programmer to register a default `com.arsdigita.kernel.StyleSheet` object against its `com.arsdigita.kernel.PackageType` object. Project integrators could override this default application stylesheet by registering a custom stylesheet against a `com.arsdigita.kernel.SiteNode` object.

These mappings were maintained in the database and the rules for querying them to discover the top level XSL template for an application were written into the standard presentation manager class (`com.arsdigita.sitenode.BasePresentationManager`). This resulted in an XSL templating architecture that was spread across multiple Java packages and left little scope for extending or altering the template resolution algorithms, without massive code replacement / duplication.

6.2.2.1.2. Stylesheet Resolver

A quick analysis of XSLT usage across projects and applications shows that there are a large number of variables that can come into play when deciding which stylesheet to apply to an application's DOM. Rather than attempt to standardise on a particular algorithm for resolving stylesheets, the WAF templating package introduces the `com.arsdigita.templating.StylesheetResolver` interface as a means to plug in an arbitrary template resolution algorithm. This interface contains a single method:

```
public URL resolve(HttpServletRequest sreq);
```

This method may use any available state information to locate a stylesheet, which transforms XML that has been generated during this supplied request object. The returned object must represent an absolute URL under any protocol supported by the `java.net.URL` class (typically, either file:// or http://). The `com.arsdigita.templating.LegacyStylesheetResolver` class provides a resolver compatible with the resolution method used in WAF releases older than 6.0. The com.arsdigita.templating.PatternStylesheetResolver class is the new preferred resolver and now the default setting.

6.2.2.2. Pattern Based Resolution

The primary implementation of the stylesheet resolver interface is the class com.arsdigita.templating.PatternStylesheetResolver. The core idea behind this approach is that there is a list of abstract paths containing placeholders, of the form ::key::. For example, a simplified version of the default list of paths looks like:

```
/__ccm__/apps/::application::/xsl/::url::-::locale::.xsl
/__ccm__/apps/::application::/xsl/::url::.xsl
```

When resolving which stylesheet to apply, the placholders are expanded based on the request state, to generate a list of real paths. Since a placeholder potentially has multiple valid values for a request, a single abstract path could expand to multiple real paths. Placeholder expansion proceeds left-to-right, to ensure a deterministic ordering of the list of real paths.

If we consider an example request for the content section admin pages at /content/admin/index.jsp, the placeholder expansions might be:

- ::application:: -> (content-section)
- ::url:: -> (admin, index)
- ::locale:: -> (en_US, en)

If we proceed left-to-right with this expansion, the first expansion is for ::application::, leading to:

```
/__ccm__/apps/content-section/xsl/::url::-::locale::.xsl
/__ccm__/apps/content-section/xsl/::url::.xsl
```

Next, the ::url:: placeholder is expanded:

```
/__ccm__/apps/content-section/xsl/admin-::locale::.xsl
/__ccm__/apps/content-section/xsl/admin.xsl

/__ccm__/apps/content-section/xsl/index-::locale::.xsl
/__ccm__/apps/content-section/xsl/index.xsl
```

Finally, the ::locale:: placeholder is expanded:

```
/__ccm__/apps/content-section/xsl/admin-en_US.xsl
/__ccm__/apps/content-section/xsl/admin-en.xsl
/__ccm__/apps/content-section/xsl/admin.xsl

/__ccm__/apps/content-section/xsl/index-en_US.xsl
/__ccm__/apps/content-section/xsl/index-en.xsl
/__ccm__/apps/content-section/xsl/index.xsl
```

Once all the placeholders have been expanded, these paths will be verified in order and the first that exists returned as the primary stylesheet for the request. In this example the first match will be /__ccm__/apps/content-section/xsl/admin.xsl.

6.2.2.3. Pattern Generators

The `com.arsdigita.templating.PatternGenerator` interface provides the mechanism for introducing new placeholders in the pattern based stylesheet resolver. This interface contains the single method:

```
  public String[] generateValues(String name,
                                HttpServletRequest request);
```

This method may use any available state information to generate a list of values for the placeholder. The elements in the array should be ordered in decreasing specificity. If there are no possible values for a placeholder, then an empty array can be returned, causing the entire abstract path to be thrown away. As a hypothetical example, consider a pattern generator for switching based on the user’s browser:

```
public class BrowserPatternGenerator implements PatternGenerator {
    public final static String GENERIC = "generic";

    public String[] generateValues(String key,
                                   HttpServletRequest req) {
        String useragent = req.getHeader("user-agent");
        if (useragent != null &&
            useragent.indexOf("MSIE") > -1) {
            return new String[] { "ie", GENERIC };
        } elseif (useragent != null &&
                  useragent.indexOf("Mozilla") > -1) {
            return new String[] { "mozilla", GENERIC };
        } else {
            return new String[] { GENERIC };
        }
    }
}
```

Example 6-1. Web Broswer Triggered Pattern Generator

Once the new pattern generator class has been implemented, it needs to be registered with the resolver by calling the static `registerPatternGenerator` method in `com.arsdigita.templating.PatternStylesheetResolver`, supplying the name of the placeholder key. This registration is best done in the static initializer block of a core class, such as a servlet.

```
static {
  PatternStylesheetResolver.registerPatternGenerator(
       "browser",
       new BrowserPatternGenerator()
  );
}
```

Available Patterns

`application`

This pattern generator expands to the current application package key. ie, the value returned by a call to `com.arsdigita.web.Web.getConfig().getApplication().getKey()`.

`host`

This pattern generator expands to the hostname and port number for the current servlet container, as returned by `com.arsdigita.web.Web.getConfig().getHost()`.

`locale`

This pattern generator expands to the current kernel execution context locale. ie the value returned by a call to `com.arsdigita.kernel.Kernel.getContext().getLocale();`.

`outputtype`

If the `outputType` request parameter is set, this expands to one of `text-plain`, `text-javascript`, or `text-html` if the parameter value is `text/plain`, `text/javascript` or `text/html` respectively. In all other cases no values are generated.

`prefix`

If the current request has passed through an instance of the `com.arsdigita.web.InternalRedirectServlet`, then this pattern generator expands to the value of the `prefix` init parameter in the web.xml servlet declaration. In all other cases no values are generated.

`url`

This pattern generator expands to multiple values based on the URL fragment remaining after the application's mount point. The set of values are generated by progressively stripping off trailing path components. The values are then normalized as follows:

1. The file extension is removed.
2. Trailing occurrences of `/index` are replaced with `/`.
3. If the path is the empty string, it is replaces with `index`.
4. Occurrances of `/` are replaced with `-`.

Some examples are:

- /index.jsp -> { "index" }
- / -> { "index" }
- /admin/index.jsp -> { "admin", "index" }
- /admin/ -> { "admin", "index" }
- /admin/item.jsp -> { "admin-item", "admin", "index" }

6.2.2.4. Configuration

The choice of which stylesheet resolver to use is controlled by the waf.templating.stylesheet_resolver property, and defaults to `com.arsdigita.templating.PatternStylesheetResolver`.

The location of the paths file to configure the pattern based stylesheet resolver is controlled by the waf.templating.stylesheet_paths property, defaulting to `/WEB-INF/resources/stylesheet-paths.txt`

To enable an application to operate with the default pattern stylesheet resolver configuration, its primary top level stylesheet should be named `/__ccm__/apps/[`*`application key`*`]/xsl/index.xsl`.

6.3. JavaServer Pages (JSP)

JSP technology is a J2EE standard for presentation. JSP pages have full access to WAF APIs. For documentation and tutorials on how to write JSPs, see http://java.sun.com/products/jsp/docs.html.

The strength of JSP is the ease with which a developer can rapidly develop and modify a single page. JSPs can be recompiled on the fly, and changing layout is straightforward, since a JSP is patterned after a standard HTML file. JSPs are recommended for projects where user interface specifications rapidly change and where the need for UI reuse is limited.

JSPs that wish to use the services of the WAF framework must extend the base JSP servlet provided by the WAF framework. See Section 7.2 `BaseServlet`.

6.4. Bebop - Reusable Web UI Components

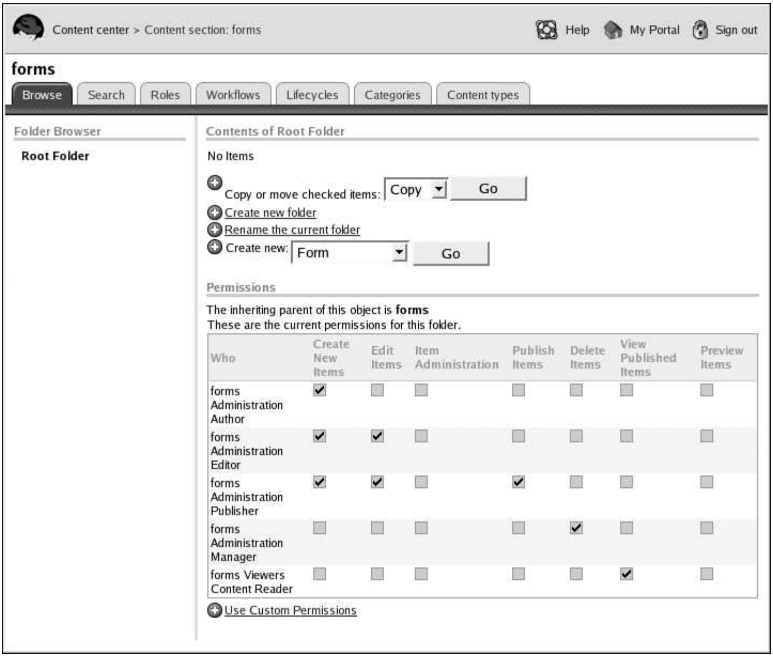
Bebop is a web-based UI component framework. It is named after Swing™, the Java UI toolkit from Sun.

Calling Bebop a *web UI component framework* has a very specific meaning:

- Bebop as a *framework* is intended to establish a standard mechanism and pattern for defining UIs. It is *not* a library of ready-to-wear UI elements, though it happens to include some.
- Bebop as a *component framework* is used to define UIs, with the resulting components useful as UI building blocks.
- Bebop as a *Web UI component framework* — Bebop is specifically designed for the conventions and constraints of the Web.

A component-based system is designed for reuse in varying contexts, bundles useful behaviors with useful state, and operates in and responds to a service-rich environment. Bebop is intended to turn a developer into a deployer by providing components which need only be told what to do, rather than how to do it.

In Figure 6-1, Bebop components provide the tabbed pane. This tabbed pane component remembers which are the currently visible components, and it knows internally how to generate the correct URLs to other tabs. This functionality is gained just in using the component.



No Items

Copy or move checked items: Copy Go

Create new folder

Rename the current folder

Create new: Form Go

Permissions

The inheriting parent of this object is **forms**
These are the current permissions for this folder.

Who	Create New Items	Edit Items	Item Administration	Publish Items	Delete Items	View Published Items	Preview Items
forms Administration Author	☑	☐	☐	☐	☐	☐	☐
forms Administration Editor	☑	☑	☐	☐	☐	☐	☐
forms Administration Publisher	☑	☑	☐	☑	☐	☐	☐
forms Administration Manager	☐	☐	☐	☐	☑	☐	☐
forms Viewers Content Reader	☐	☐	☐	☐	☐	☑	☐

Use Custom Permissions

Figure 6-1. Tabbed Pane Using Bebop Components

6.4.1. Working With Bebop

Chapter 12 *Presentation (Bebop) Tutorial* discusses specifics of implementing Bebop. This section is a more abstract look into the design and usage of Bebop.

Bebop components follow these guidelines:

- Provide a tree of components; components are also containers.
- Consolidate parameter validation logic and support for custom validators.
- Lock data structures for reuse across requests.
- Automate state preservation and services to allow components to control their own state.
- Multiplex Swing-like events onto the HTTP request.
- For a given request, one component is selected and has control.
- Transforms XML for global style; no HTML in Java strings.

The code in Example 6-2 gives an idea of how Bebop is used.

```
public void service(HttpServletRequest sreq, HttpServletResponse sresp) {
    Page page = new Page("Content Section");

    page.add(new Label("Content Section"));
    TabbedPane tabs = new TabbedPane();
    page.add(tabs);

    tabs.addTab("Browse", new BrowsePane());
    tabs.addTab("Search", new SearchPane());
    tabs.addTab("Roles", new RoleAdminPane());

    page.lock();

    Document doc = page.buildDocument(sreq, sresp);

    transform(sreq, sresp, doc);
}
```

Example 6-2. Basic Bebop Page

6.4.1.1. Bebop Lifecycles

Bebop components used in a page design have a regular lifecycle.

UI Lifecycle

1. Register each component.

   a. Each component adds any state parameters and event listeners it needs.

   b. Each component is added to the page and given a unique key so that any state carried in the query string is safe from collisions.

2. Lock the component tree.

   a. Components can no longer be added or removed. The component instances are now safe for reuse across requests.

3. Service requests.

Request Lifecycle

1. Client sends an HTTP request.

2. A new page state object is created.

   Using the servlet request and the parameter models defined by the developer, the page builds an object representing the state of the current request.

   In addition to component state parameters, the selected component and the visibility state are recovered from the servlet request state.

3. Fire the request event.

   Any request listeners added when each component registered itself run at this time. Note that this runs before the page request state is validated; in fact, the request event runs before much of any work is done.

4. Validate page state.

   This is when custom validators, e.g. a zip-code validation listener, are run. Parameters are also typed at this step.

   If there are errors, they are saved so that the component whose state is invalid may choose how to present the issues.

5. Fire the control event.

   When a client sends a request to a Bebop page (e.g. a mouse-click on a tab); only one component receives the request. This triggers the respond method of that component only; no other components response methods are involved.

   If the request state is deemed valid, the selected component has the opportunity to respond, perhaps changing the request state and thus the outcome of further processing.

   A control event handles form submission, control links, and has influence on the state and visibility before any output is produced.

6. Fire the action event.

   The action event is an opportunity for any component to run code before the response is committed and after the controlling component has responded.

   Any component may listen and edit state and visibility before any output is produced. This is the last opportunity to edit before XML is written out.

7. Generate XML.

   The components generate the XML. They write to the document created in the first step. Each component generates its own semantic XML (such as *XUL*, Mozilla's presentation-based cross-platform markup language) and also delegates to its children. A DOM is built.

8. Transform the XML.

   The XML document is transformed with an XSLT stylesheet and sent back to the client in the form of HTML and CSS.

   By following this method, it is easy to define and maintain a style and layout uniformity. A developer or designer can drop a different XSLT into the last step in a page request cycle.

9. The client receives the transformed results of the page request.

6.4.2. JSP Integration with Bebop

JSP is integrated with Bebop components (XML sources) and XSLT by using a JSP tag library to allow the use of Bebop components in an otherwise standard JSP. The JSP tag libraries accomplish this by performing transformations on the XML document produced by a Bebop page. This JSP tag library also directs the generation of output by rendering the resulting XML document through XSLT, so that the included components displayed in a JSP are styled with the standard template rules that are used on the rest of the site. See Appendix A *Bebop Tag Library Reference*.

Java developers can also construct their Bebop pages in JSP. The tag library for declaring Bebop pages is separate from the one for displaying Bebop components, but they are intended to work together. This is different from using a Bebop component inside of JSP. In this instance, the specific Bebop JSP tags takeover JSP similar to writing and controlling a servlet.

There are several motivating factors for integrating JSP with Bebop. First, laying out pages using JSP integration will allow web developers who are not Java programmers to use third-party web publishing tools (Dreamweaver, etc.) to alter the layout of pages, add new components to an existing page, edit form field labels or prompts, etc. It would also be possible to make completely new pages with JSP that display components from existing Bebop pages.

Second, JSP is a convenience for Java developers creating Bebop pages, because JSP shortens the development cycle for individual pages by eliminating the need for a manual recompile and server restart when a page is changed.

The overall request pipeline for constructing a page with Bebop and displaying it with JSP follows this sequence:

1. The requested JSP obtains an XML document from a Bebop page object and the current request state.
2. The tags in the requested JSP construct a new XML document, copying pieces from the Bebop page where needed.
3. The resulting XML is passed as the XML input to an XSLT transformation step, using global stylesheets.
4. The final result is sent to the user's browser.

6.4.3. Relationship Between Static and Dynamic Pages

The JSP tag libraries for Bebop are designed to work together with Bebop's model of static, shared `Page` and `Component` objects, while providing familiar semantics to JSP authors. The division in Bebop between static Page objects that produce dynamic output on each request is reflected by the division of tags into two separate libraries:

A library of `define:...` tags.

> Used for creating Bebop components.
>
> The `define:...` tag library defines a Bebop page and its components with JSP tags. Since the page structure may be static, JSP code in the `define:...` tag library is not guaranteed to run on each request. Care must be taken to not put any runtime expressions or scriptlets into a `<define:page>` block that are expected to run on each request.

A library of `show:...` tags.

> Used for displaying the output form Bebop components inside a JSP page.
>
> The `show:....` tag library is dynamic. Any Java code inside a `<show:page>` block is guaranteed to run on each request, even if the components used are static. This is because the `show:...` tags manipulate the XML output produced by Bebop components on each request, and not the components themselves. Any valid JSP runtime expressions, tags, or scriptlets may be used in conjunction with `show:...` tags and they should yield the same results they would in any other JSP.

Caching in `<define:page>` can be suppressed with the JSP tag attribute *`cache="no"`*. This will cause the JSP `define:page` tag, along with the code and all the other tags inside it, to execute on each request; the entire `Page` object will be re-created and garbage-collected on each request.

Warning

> Although it seems there could be some benefit to changing the structure of a `Page` on each request, it is not recommended.
>
> Variations on the structure of a page will interfere with Bebop's state maintenance, because the URL and form variables that preserve state for individual components are linked to the component's position in the overall page structure. Runtime errors or unpredictable results may occur if a page's structure differs between subsequent requests.

Certain types of non-structural page variations between requests are permissible. Most importantly, the individual options in option groups (radio groups, select widgets, checkbox groups) are not components themselves, so the following code is permissible:

```
<define:radioGroup>
   <% loop { %>
```

```
      <define:option label="<%= rtexpr %>" value="<%= expr %>"/>
   <% } %>
</define:radioGroup>
```

This is especially true if the options in the group do not change frequently. Also, the contents of stateless components (Labels, Links, Images, etc.) may change between requests, though the position of the component in the page should not.

Further examples can be found at Appendix A *Bebop Tag Library Reference*.

Chapter 7.
WAF Component: Web

The Web component of WAF makes the persistent data and domain logic of your application available to others over protocols such as HTTP. It integrates the Java Servlet API and the kernel and persistence components of WAF, as originally discussed in context in Section 2.2.6 *Web*.

7.1. Applications

WAF supports deploying multiple instances of an application type. A typical WAF installation may have three forum instances, each with its own data and configuration at distinct locations on the site. A WAF **Application** is the base on which developers build such applications in the WAF framework.

In the following, we distinguish between an application, here meaning the concrete application you wish to develop, and **Application**, the WAF base class used to define such new applications.

7.1.1. Application Properties

Applications have a number of specific properties.

- **Applications** are persistent — An **Application** has persistent properties describing its location, name, etc. It is also a container of persistent data, often user-entered data, that a concrete application uses to do its work. A forum, for instance, will contain database-stored message threads and posts.
- **Applications** work with the dispatcher — **Applications** are designed to work together with the WAF dispatcher to convert user-friendly URL paths into an application servlet and an application instance containing its specific data. The dispatcher uses the location and parent properties of an **Application** to look it up using the request URL.
- **Applications** have a Servlet — Once the dispatcher has routed a request to an **Application**, the **Application** responds using a Java Servlet.
- **Applications** are associated with Stylesheets — The WAF presentation layer uses XSLT stylesheets to implement the look-and-feel of its applications. The **Application** abstraction carries with it a stylesheet property that the presentation layer uses to fetch the appropriate stylesheet.
- **Applications** extend Resources — **Applications** leverage a feature of the WAF kernel API, Resources. Resources model containment, resource-partitioned user and configuration data, and a data-backed type system. See Section 4.4 *Kernel Resources*. For example, **Application** uses the containment behavior of Resource to implement its security.

7.2. `BaseServlet`

The WAF `BaseServlet` implements a Java Servlet and integrates it with the WAF kernel and persistence layers. The `BaseServlet` performs two functions:

- Transaction management — The `BaseServlet` is responsible for starting and stopping the request's database transaction. If any errors occur, the servlet will roll back the transaction. If application code requests a redirect, the `BaseServlet` ensures that the transaction is finished before proceeding.

- Request context — The BaseServlet packages certain facts about the request, such as the current user, current application, etc., and loads them into a context object whose data is available to any code running inside the servlet.

The BaseServlet is used in two principal ways in WAF:

- New applications will sometimes use custom servlets that extend the BaseServlet and use the Servlet API to implement their UI.
- Applications that use JSPs can take advantage of the WAF platform by having their JSP pages extend the BaseJSP class. Any code inside of the JSP will then be able to use the transaction and request context setup by the BaseServlet.

7.3. Dispatcher

The WAF dispatcher connects user requests to **Applications**. A forum instance may live at /apps/sales/forum/ while another is at /apps/engineering/forum/. The dispatcher maps requests to the distinct Servlet and the distinct **Application** instance corresponding to each.

The design of the dispatcher is such that the URL-to-application mapping is reversible. Just as it is possible to navigate from a URL to a specific data partition, it is possible to navigate from a data partition back to a URL. This is done so that, given an object and its container, the WAF platform can generate an address to it through the dispatcher.

A typical path through the dispatcher takes the following steps:

1. A request for the engineering forum at /apps/engineering/forum/index.html comes in.
2. The dispatcher is mapped to the pattern /apps/*, so the servlet container sends the request to the dispatcher servlet.
3. The dispatcher servlet searches the database of applications using the path, minus the /apps prefix and minus the filename suffix. The path /engineering/forum matches an Application instance and the dispatcher servlet loads it.
4. The dispatcher servlet then fetches the path to the servlet of the application. The forum servlet is mapped to /__ccm__/servlet/forum. The dispatcher forwards the request, now in the form /__ccm__/servlet/forum/index.html.
5. The forum servlet receives the request. It uses application ID set on the request by the dispatcher to load the data for the engineering forum. It uses the path info (in this case /index.html) part of the request to display the appropriate page in the forum UI.
6. The engineering forum is served.

The WAF dispatcher is not responsible for serving static resources. Instead, your servlet container in its default configuration performs the work of serving images, CSS and XSL files, and JSPs. No special interaction with the dispatcher is necessary to use this feature of servlet containers.

III. Equipping Developers

This section provides information needed to develop on the Web Application Framework. Included is a chapter on setting up a development environment and a number of tutorials related to the WAF components.

Table of Contents

Chapter 8.
Developing with WAF

This chapter introduces prerequisites to working with WAF as well as the tools that are available to the developer.

It is presumed that anyone developing for WAF has sufficient knowledge in his or her area of expertise and can perform accordingly. For this reason, both high-level concepts, such as database design and programming theory, and basic concepts, such as command line usage and shell scripting, are not addressed.

8.1. Developer Education

A developer undertaking additions and modifications to this complex system needs a sufficient level of appropriate knowledge. In particular, a developer should be knowledgeable and skillful in the following areas:

- Java — the programming language used in WAF.
- Servlets and JSP — the J2EE standards used in WAF.
- SQL — used for directly accessing the relational databases in cases where the object-relational mapping provided by the persistence engine is deemed insufficient.
- UML — parts of the Unified Modeling Language's basic vocabulary, especially those related to class diagram, are used by the persistence framework.
- XML and XSLT — key standards used in the WAF presentation system.
- CSS — used to style WAF pages.
- Concepts of Object Oriented (OO) programming.
- Relational database design and object-relational mapping concepts.

In addition, a knowledge of Perl is necessary for extending the build system. PL/SQL and PL/pgSQL are used to maintain certain denormalizations in the database.

A developer either needs to understand how to install, configure and administrate the chosen database or have an experienced DBA performing those functions.

Finally, anyone who is involved with the installation, configuration and administration of the server(s) and development environments for the WAF implementation requires sufficient skill in the target installation OS, e.g. Red Hat Enterprise Linux AS.

Chapter 14 *References* contains references to additional resources that can help in understanding WAF.

8.2. Third-Party Development Tools

Tools are very straightforward. Because the WAF system is designed around open standards and open source solutions, a developer is given a wide range of development tools and environments.

Programming is done primarily in Java, for which numerous *Integrated Development Environments* (IDE) exist. One useful open source IDE is **Eclipse**. Integration of **Eclipse** with WAF is covered in Section 8.4 *Setting Up Eclipse and WAF*. A developer can also use any text editor such as **vi** or **Emacs**.

For access to build systems and configuration files, either console or remote access to servers is required. The best choice for remote access is **SSH**, a protocol suite of network connectivity tools which includes a secure telnet replacement and a *secure file-copy* utility (**scp**). Both open source and proprietary implementations of **SSH** exist for virtually every OS. An open source implementation of **SSH** for Unix-like systems can be found at http://www.openssh.org. The site also provides pointers for various alternatives to **OpenSSH**.

The **OpenSSH** client is often packaged with the following Unix and Unix-like systems:

- Linux
- *BSD (FreeBSD, OpenBSD, NetBSD)
- Apple OSX
- Legacy Unices (Solaris, AIX, HP-UX, SCO, Irix, etc.)

For Win32 systems, a good open source application is **PuTTY**, found at http://www.chiark.greenend.org.uk/~sgtatham/putty/. It is a **Telnet/SSH** client which also supports **sftp** (*secure ftp*) and **scp** on a Win32 platform.

The ultimate output of WAF is HTML pages. These can be viewed in any modern Web browser. For the most part, Web browsers will "just work", although each browser may implement standards differently.

For actual development efforts, it is highly recommended to use a version control system, also called *software control management* (SCM). Several mature systems exist. On the open source side, there is **CVS** (http://www.cvshome.org). The Red Hat Applications development teams use **Perforce** for version control. If your code might integrate back into the main development tree, you may want to look into **Perforce** at http://www.perforce.com.

8.3. Developer Support

WAF provides the **Developer Support** application to give developers high-level information about each server request. **Developer Support** shows data such as request duration, number of queries, and information about each query executed. In addition, **Developer Support** allows developers to adjust their server logging level at runtime.

8.3.1. Enabling Developer Support

To enable **Developer Support**, edit your `enterprise.init` file and set the following initializer to `active`:

```
init com.arsdigita.webdevsupport.Initializer {
  active = true;
}
```

To disable **Developer Support**, modify your `enterprise.init` file and modify the *`webdevsupport`* Initializer so that `active = false`.

Once you have enabled **Developer Support** in your `enterprise.init` file, you must restart the servlet container in order to use it. E.g. if you are using Tomcat, `service tomcat restart` is sufficient.

Warning

> You should always make sure that **Developer Support** is disabled in a production system. **Developer Support** imposes a significant performance penalty on systems as it collects and stores request debugging information.

It is possible to have **Developer Support** enabled and not running. This still provides a measurable performance hit because it registers a listener. Once enabled in `enterprise.init` and recognized by the servlet container, you can turn **Developer Support** on and off via the administrator page. See Section 8.3.2.2 *Developer Support Features* for more information on controlling **Developer Support** via the administrator page.

8.3.2. Using Developer Support

8.3.2.1. Accessing Developer Support

To access **Developer Support**, login as a site-wide administrator and then navigate to `/ds/` on your server. For example, if your server is running on your localhost and your Dispatcher Servlet Path is `/ccm`, then you would access **Developer Support** at `http://localhost/ccm/ds/`.

8.3.2.2. Developer Support Features

Once you have accessed **Developer Support**, you will find an index page listing up to the last 100 requests to the server as well as several options for configuring your server:

Request Information

> **Developer Support** stores information about the last 100 requests to the server, accessible through the index page. For each request, information such as the duration, the number of database operations executed, the requesting IP address, and the URL requested is available. This information is especially useful for performance-tuning. For example, if you are looking for a slow or expensive page in a certain use case, look for a request with an unusually long duration or large number of queries.
>
> Click on a **Request URL** to find out detailed information about that individual request. The **Request Information** page provides details such as the amount of time for various stages of the request, and the text and duration of each SQL operation.
>
> Click on a **SQL Operation** to find detailed information about that specific operation. The **Query Information** page provides a stack trace showing the code that led to the SQL operation's execution. Also, if the server is running on an appropriately configured **Oracle** database, the page will provide a link to show the **explain plan** of the query.

Query Log

> On the **Developer Support** index page, next to each request, is a link to a **Query Log**. A request's **Query Log** page presents a text page of all the SQL operations for that request, formatted for easy cut-and-paste into a database client such as **SQLPlus** or **PSQL**.

Log4j Logger Adjuster

> The **Developer Support** index page provides a link to a **Log4j Logger Adjuster**. This **Log4j** adjuster allows the developer to change the server's logging level at runtime. See also Section 8.5.1 *Log4J*

Disable Request Logging

> The **Disable Request Logging** link on the **Developer Support** index page allows you to toggle at runtime whether **Developer Support** actively stores information about each request. If you disable request logging, **Developer Support** will clear all previously stored request information as well as stop recording information about new requests.

Show Hits To Developer Support

The **Show hits to developer support** link on the **Developer Support** index page toggles the tracking of requests against the **Developer Support** application itself. As most developers will not be interested in this information, this information is usually left hidden.

8.4. Setting Up Eclipse and WAF

The following sections describes how to set up a WAF development and debugging environment within **Eclipse**, an open source *Integrated Development Environment* (IDE).

8.4.1. Installing Eclipse

1. Install WAF — install as normal, using the RPM's and WAF Autobuild system. For more information, see the *Red Hat Web Application Framework Installation Guide:*.
2. Install **Eclipse** — **Eclipse** can be obtained from http://www.eclipse.org. Install using RPM, if possible.

   In order to take advantage of the anti-aliased fonts within **Eclipse**, use a build of **Eclipse** designed for the GTK in Red Hat Linux 8.0 and later.
3. You may also wish to install the **Perforce**, **Resin**, and **JavaCC** plugins for **Eclipse**, which can be found at:
   - http://sourceforge.net/projects/p4eclipse
   - http://www.improve-technologies.com/alpha/resin
   - http://sourceforge.net/projects/eclipse-javacc
4. To install the Perforce and Resin plugins, use the **Eclipse Update Manager** by selecting **Help** => **Software Updates** => **Update Manager**. Next, add bookmarks for the download URLs listed by the plugins' Web sites. The Update Manager can then automatically download and install the Perforce and **Resin** plugins.

   To install the JavaCC plugin, download the plugin and unzip it in **Eclipse**'s plugin directory. Then, follow the instructions in the plugin's README file.
5. Install **Resin** — if you are using the **Resin** application server. You may download **Resin** at Caucho's website: http://caucho.com. You may also want to download the source code for **Resin** in order to trace through its code when debugging.

8.4.2. Eclipse Preferences Configuration

To set preferences within **Eclipse**, go to **Window** => **Preferences**. Noteworthy preferences include:

- **Workbench** => **Editors**. There is a key bindings option where you can select Emacs key bindings.
- **Java** => **Compiler** => **Errors and Warnings**. You probably only want to select **Error** for the options:
  - `Unreachable code`
  - `Resolvable import statements`

Tip

It is recommended to only select **Warning** for the options:

- `Methods overridden but not package visible`
- `Methods with a constructor name`
- `Hidden catch blocks`

Otherwise, your compilations will output hundreds of messages in your **Tasks** window.

8.4.3. Eclipse Project Configuration

This section covers adding and configuring projects to **Eclipse** for developing with WAF.

8.4.3.1. Adding Projects

WAF

1. In **Eclipse**, go to **File** => **New** => **Project**.
2. Select **Java Project**.
3. Name the project, e.g. *`webapp`* and select your existing WAF core-platform directory as the project directory.
4. In the source tab, make sure that **Eclipse** lists the WAF source directory, `src/`. If it is not there, click **Add Existing Folders...** to add it. You may also want to add the directory for the Junit test code, `test/src/`.

   After adding the existing source folders, click on **Create New Folder...** and name the folder `_generated`. This is where **Eclipse** will store its JavaCC-generated source files.
5. Select the **Libraries** tab once you are presented with the **Java Settings** dialog box.
6. Click on **Add JARs** and select all the JARs under your WAF Project (typically under `lib` and `etc/lib`).
7. Click on **Add External JARs** and add the following JARs:
   - `$ORACLE_HOME/jdbc/lib/classes12.zip`
   - `$RESIN_HOME/lib/jta-spec JAR`
   - `$RESIN_HOME/lib/resin.jar`
8. Click on the **Order and Export** tab. Select `All`.
9. Create a build folder for **Eclipse** on your file system. Then, enter the path to that build folder in the **Build output folder** input box.
10. Click on the **Finish** button.

CMS and Other Projects

To add CMS or other projects to **Eclipse**, follow the same steps as for adding WAF, with the following exceptions:

- For CMS and other projects that depend on WAF, you do not need to add external JARs. **Eclipse** will pick up these libraries from WAF.
- When you reach the **Java Settings** dialog box, select the **Projects** tab. For CMS, add a dependency upon WAF. For projects that use both WAF and CMS, add a dependency upon both WAF and CMS.
- Create build folders for the respective projects.

8.4.3.2. Configuring Eclipse for Perforce

For each project that you have added, go to the Package Explorer and right-click on it. Then, select **Team => Share Project**. Select the options for **Perforce**. Now, you should see a **Perforce** menu and **Perforce** options for your source files.

8.4.3.3. Building and Deploying with Eclipse

Eclipse uses its own build directories for compiling your source files. However, you will probably still want to use the WAF build system for building and deploying your application. Therefore, you should create a shell script that will do this for you. Then, you may call this shell script to build and deploy WAF from within **Eclipse** when you are ready to run your application.

Here is an example shell-script, which should be modified appropriately to fit the target environment.

```
#!/bin/bash

#setup environment
. /etc/profile.d/ccm-config.sh
. /etc/profile.d/ccm-devel.sh
. /etc/profile.d/ccm-scripts.sh

CCM_HOME=/var/ccm-devel/dev/user/cms_dev
ORACLE_HOME="/opt/oracle/product/9.2.0"
JAVA_HOME=/opt/IBMJava2-131

CLASSPATH=$CCM_HOME/core-platform/lib/jaas.jar
CLASSPATH=$CLASSPATH:$CCM_HOME/core-platform/lib/jce.jar
CLASSPATH=$CLASSPATH:$CCM_HOME/core-platform/lib/sunjce_provider.jar
CLASSPATH=$CLASSPATH:$CLASSPATH:$ORACLE_HOME/jdbc/lib/classes12.zip
CLASSPATH=$CLASSPATH:$CCM_HOME/core-platform/etc/lib/iDoclet.jar
export CLASSPATH

export ANT_OPTS="-Xms128m -Xmx128m"

##########
#build
##########

#uncomment to make enterprise.init
#cd $CCM_HOME
#ant make-config
#ant make-init
```

```
#clean
cd $CCM_HOME
ant clean

#build
cd $CCM_HOME
ant deploy

#javadoc
cd $CCM_HOME
ant javadoc
ant deploy-api
```

Example 8-1. Shell-script to build and deploy from within Eclipse

In your `ant.properties` file, make sure that you have set `compile.debug=on`. Once you have created this shell script, in **Eclipse** go to **Run** => **External Tools** => **Configure**. A dialog box will appear for configuring external tools. Select **New** and in the **Tool Location** input box, select your shell script for deploying WAF. Select the options:

- `Block until tool terminates`
- `Show execution log on console`

Then, select **OK**. Now, in **Eclipse**, your shell script will appear under **Run** => **External Tools**. Use that to build and deploy WAF from within **Eclipse**.

8.4.4. Eclipse Debugging Configuration

To do runtime debugging of WAF within **Eclipse**, you will need to add a new **Resin** launch configuration for running WAF and a new **Remote Java Application** debug configuration for attaching to WAF.

WAF Execution Setup

To be able to run WAF from **Eclipse**, select **Run** => **Run**. Select **Resin** and then click on **New**. Then, enter information for the following tabs:

- **Main** Tab

  To set your project, select the highest-level project, which is the project no other projects are dependent on. This is where you select your `resin.conf`.

- **Arguments** Tab

  In the VM arguments text box, enter the following, all in one line — this line is broken for printing purposes with a "\":

  ```
  -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=\
  dt_socket, server=y, suspend=n,address=8000 -Xms128m -Xmx128m
  ```

  You may allocate more memory than 128 MB to the server if you wish.

- **Classpath** Tab

  Add all the external JARs under `$RESIN_HOME/lib` to your user classes.

- **Common** Tab

  At the bottom of the dialog box, select **Display in favorites menu: Run**.

  You are now finished and may close the dialog box.

WAF Debugging Setup

In **Eclipse**, select **Run** => **Debug**. In the dialog box, select **Remote Java Application** and then click on **New**. Then enter information for the following tabs:

- **Connect** Tab

  Enter a name for your debug configuration, such as *webapp-debug*. Select the same project previously selected as the main project. Enter the host name (`localhost` should be fine). Use `8000` as your connection port. Also, select **Allow termination of remote VM**.

- **Source** Tab

  Add all the external JARs under `$RESIN_HOME/lib` to your source lookup path.

- **Common** Tab

  At the bottom of the dialog box, select **Display in favorites menu: Debug**.

You are now finished and may close the dialog box.

8.4.5. Eclipse Setup Validation

Your **Eclipse** setup should now be complete. To test this:

1. Select **Run** => **External Tools** => **[*`Deploy Script`*]**.
2. Select **Run =>** **Run** and pick the WAF configuration. The server's log output should become visible in the **Console**. Wait for the server to finish initializing.
3. Select **Run** => **Debug** and pick the debug configuration. This opens the **Debug** perspective. Open a source file, set a breakpoint, and request a page from the server.

If everything is setup correctly, you can now develop with **Eclipse** and use it to step through your WAF code.

8.5. Using logging for debugging

When code breaks, there are two ways to figure out what went wrong: launch a debugger and step through the program, or add logging statements to see what is going on. In its crudest form, logging means adding a few `System.err.println()` calls here and there. What are the benefits of logging over using a debugger?

> *The Practice of Programming* by Brian W. Kernighan and Rob Pike
>
> As a personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that it is easy to get lost in details of complicated data structures and control flow; we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places. Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugger sessions are transient.

Another important use of logging is to provide an audit trail: who did what and when. The predominant pattern in WAF is to write such auditing information to the database rather than the filesystem. The logging framework in WAF is intended to be used primarily for troubleshooting and debugging.

8.5.1. Log4J

Using `System.out.println()` for logging has numerous drawbacks.

1. It degrades performance. There is no easy way to conditionally turn off logging statements system-wide, without resorting to error-prone and maintenance-intensive workarounds such as:
   ```
   if (DEBUG) {
       System.err.println("foo=" + foo);
   }
   ```
   where *`DEBUG`* is a system-wide boolean parameter that you can easily set to true or false.
2. Even with the above workaround, you don't get a fine-grained degree of control. The logging is either on, or off across the board. You could, of course, introduce more than one system-wide flag, like so:
   ```
   if (WARN) {
       System.out.println("warn: bar=" + bar);
   }
   ```
3. You are constrained to outputting only to the standard out and error devices. What if you need to output to rotated log files? Additional custom code is required to support this requirement.

Going down this path slowly but surely, you may end up developing a general-purpose logging library. Rather than reinventing the wheel, WAF relies on the **Log4J** library from the Jakarta project of the Apache Foundation. If you are not familiar with it, please skim through the online documentation at http://jakarta.apache.org/log4j/docs/. The rest of this document tries to highlight rather than explain the most salient **Log4J** features.

The key abstractions that **Log4J** provides are:

1. Ability to log to multiple devices. See the Javadoc for the `Appender`[1] interface and its implementing subclasses. There are many types of appenders included in **Log4J**: file appender, rolling file appender, remote syslog daemon appender, asynchronous appender, etc. Appenders are fairly easy to extend for special purposes. For instance, you could write a database appender. (In fact, several have already been written.) See Section 8.5.4 *Custom Appenders* for more discussion on this.
2. Ability to easily configure the format of the message. For example, each message can automatically include the timestamp, thread name, location (file name and line number), full or abbreviated class and method name from the which the message originated, etc. See the Javadoc for `Layout` (at http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/Layout.html) and its subclasses.
3. API for logging errors and exceptions. See Section 8.5.2 *Always log the Throwable*.
4. Different logging levels. The minimum set of possible levels are `off`, `fatal`, `error`, `warn`, `info`, `debug`, and `all`. Extending the list of levels is possible but discouraged. You can change the requested level of logging either by altering your configuration files before system startup, or at runtime. More on this in Section 8.5.3 *Runtime logging level manipulation*. See also the Javadoc for `Level` (http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/Level.html).
5. Fine-grained hierarchy of loggers. A logger is the central class that performs all logging operations. Loggers are named with dot-delimited names like `foo.bar.baz` or `foo.bar.quux`. Loggers form a hierarchy based on their names. You can configure and manipulate loggers in bulk by referring to their parent names. For example, both `foo.bar.baz` or `foo.bar.quux` are children of `foo.bar`.

   Each logger has an associated logging level. By setting the logger's level, you can control the amount of information logged. Log statements are enabled only if the level of statement is

[1] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/Appender.html

greater than or equal to the current level of the logger. For example, if you set the `foo.bar.baz` logger’s level to `warn`, it will only log message whose level is `warn`, `error`, or `fatal`. By setting the level for the `foo.bar` logger, you are automatically setting it for `foo.bar.baz` and `foo.bar.quux`, unless overridden on a more specific level.

6. Miscellaneous other items such as *filters* (http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/spi/Filter.html) and *flexible configuration*.

The next sections examine some of these benefits in more detail.

8.5.2. Always log the Throwable

In general, if you catch an exception and decide not to rethrow it for whatever reason, you should log it. In doing so, avoid the following anti-pattern:

```
try {
    doSomething();
} catch (FooException ex) {
    ex.printStackTrace();
    s_log.warn("foo occurred: " + ex.getMessage());
}
```

Example 8-2. Exception logging anti-pattern

The `printStackTrace()` method should not be used, because its output goes to the standard error device `System.err` rather than the devices managed by **Log4J**.

The `getMessage()` method should not be used, because it does not include the stack trace. Stack traces are invaluable for tracking down the source of error conditions.

The following pattern should be used instead.

```
try {
    doSomething();
} catch (FooException ex) {
    s_log.debug("foo occurred", ex); // or
    s_log.info("foo occurred", ex);  // or
    s_log.warn("foo occurred", ex);  // or
    s_log.log(Level.WARN, "foo occurred", ex);
}
```

Example 8-3. Log the Throwable

Note that the various logging methods provided by the `Logger` [2] class all take a Throwable parameter, i.e. an Error or Exception. Given a throwable, **Log4J** will print *both* the error message obtained via `getMessage()` and the stack trace.

However, be careful not to fall into this trap:

[2] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/Logger.html

```
try {
    doSomething();
} catch (FooException ex) {
    s_log.debug(ex);
}
```

Example 8-4. Swallowed stack trace anti-pattern

There is no `debug(Throwable throwable)` method — there is only `debug(Object msg)`. Same goes for warn, info, etc. The above statement is essentially equivalent to the following:

```
s_log.debug(String.valueOf(ex));
```

This is probably not the intended result.

What if you want to log the current stack trace? The easiest way to do this is to log a new `Throwable`:

```
s_log.debug("got here", new Throwable());
```

Example 8-5. Logging the current stack trace

A variation of this technique is to add a public `Throwable` field to a class.

```
public class Frob {
    // add the following line temporarily
    public final Throwable STACK = new Throwable();
}
```

Example 8-6. Tracking the origin of an object

Once this has been done, you can write code such as this:

```
class Foobar {

    void doStuff(Frob frob) {
        if ( !checksOutOK(frob) ) {
            s_log.error("Got a sketchy frob", frob.STACK);
        }
    }
}
```

This allows you to figure the origin of the `frob` object at a distant point further down the execution path.

8.5.3. Runtime logging level manipulation

Logging levels are usually configured at system startup via the configuration file. It is possible to change the logger's level at runtime. One way to do this is via the **Developer Support** interfaces. (See Section 8.3 *Developer Support*.) From the programmatic point of view, runtime manipulation of the logging level boils down to something as simple as this:

```
// assume s_log is instance of Logger
s_log.setLevel(Level.WARN);
```

This sets the logging level for the `s_log` logger to `warn`. This means that messages output via, say, `debug(Object msg, Throwable throwable)` will be suppressed, because `warn` is stricter than `debug`.

The standard pattern for instantiating loggers that is used throughout WAF is as follows:

```
package foo.bar;

import org.apache.log4j.Logger;

public class Baz {
    private final static Logger s_log = Logger.getLogger(Baz.class);
}
```

Example 8-7. Standard way of instantiating loggers

This instantiates a singleton logger named `foo.bar.Baz`. Note, however, that assigning the logger to a private static variable is merely a matter of syntactic convenience. If you ever need to, you can obtain the same logger from anywhere else in your code.

```
package foo.frob;

import org.apache.log4j.Logger;

class Nosey {

    void doStuff() {
        Logger logger = Logger.getLogger("foo.bar.Baz");
        logger.debug("got the logger used by the Baz class.");
    }
}
```

Example 8-8. Random access to loggers

There is a couple reasons why this might be useful.

1. This allows you to adjust the logging level of any logger at runtime. If you have a production system that starts exhibiting erratic behavior, you can turn up the logging level in selected classes without shutting down and restarting the system. This is what **Developer Support** allows you to do.
2. There are times when you want to see what goes on in the class `Foo` when you call a method reliant on `Foo` elsewhere in your code.

To elaborate on the last point: Consider the logging output generated by the `com.arsdigita.db.PreparedStatement` logger. If the logging level for this logger is set to `info`, it will log every executed query and its bind variable values. If you are interested in seeing the query your code generates, you can set this logger's level to `info` in your config file. Note, however, that this will result in *all* queries being logged system-wide. That may be many more than you require. The alternative is to adjust the logging level at runtime:

```
// assume all the necessary imports

class Frobnivator {

    void frobnivate() {
        Logger logger = Logger.getLogger("com.arsdigita.db.PreparedStatement");
        Level old = logger.getLevel();
        logger.setLevel(Level.INFO);

        doSomething();

        logger.setLevel(old);
    }
}
```

Example 8-9. Adjusting logging level temporarily

In the above example, you end up enabling the `PreparedStatement` logger only for the duration of the `doSomething()` method. This may significantly cut down the number of queries logged, thus making it easier for you to see what is going on.

The obvious caveat here is that extraneous queries may still end up getting logged in the presence of multiple threads. This is because the logger's level cannot be adjusted on a per-thread basis - it can only be changed globally.

8.5.4. Custom Appenders

Out of the box, WAF provides two appenders: `ConsoleAppender` [3], wired to your standard output device (usually a TTY), and `RollingFileAppender` [4] tied to a file whose location is configurable.

You can also use other appenders. One of the interesting possibilities is the `SocketAppender` [5]. It expects a log viewer (or some other log processing application) to listen on the specified TCP port, possibly on a different machine. There are a number of GUI log viewers that work with the socket appender. These viewers give you fine-grained control over your logging output. See, for example, **Chainsaw** [6] and **Lumbermill** [7]. Both of these log viewers allow you to filter the output by level, logger, thread name, etc.

8.5.5. Beware of Buffered Streams

The Java language provides two abstractions for doing output: *output streams* [8] for working with sequences of bytes, and *writers* [9] for working with sequences of characters. An output stream or a writer can be buffered or unbuffered. For more information, read about `BufferedOutputStream` [10] and about `BufferedWriter` [11].

When I/O is buffered, it means that bytes or characters that you write do not necessarily go to their intended output device immediately. They are cached in an intermediate buffer. When the buffer fills up, it is flushed — an actual write to the underlying physical device occurs.

Why should you care about this? If you log a message to a buffered output immediately before the system crashes, you may not see the message logged anywhere. The system didn't have a chance to flush the buffered stream or writer. This situation is fairly rare, but when it does happen, it may stump the unwary troubleshooter. If you want to be absolutely sure you are not losing any logging statements, you must use unbuffered output.

For example, the standard output `System.out` may be buffered, while the standard error device `System.err` is usually not. Therefore, `System.err` is preferable when you want to make sure no logging is lost.

[3] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/ConsoleAppender.html
[4] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/RollingFileAppender.html
[5] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/net/SocketAppender.html
[6] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/chainsaw/package-summary.html
[7] http://traxel.com/lumbermill/
[8] http://java.sun.com/j2se/1.3/docs/api/java/io/OutputStream.html
[9] http://java.sun.com/j2se/1.3/docs/api/java/io/Writer.html
[10] http://java.sun.com/j2se/1.3/docs/api/java/io/BufferedOutputStream.html
[11] http://java.sun.com/j2se/1.3/docs/api/java/io/BufferedWriter.html.

What kind of I/O does log4j use: buffered or unbuffered? It depends on how the particular appender is configured. In the case of `FileAppender` [12], you have the option of configuring it either way. See the methods `getBufferedIO()` [13] and `setBufferedIO(boolean)` [14].

For maximum performance, you should use buffered I/O. Only switch to unbuffered I/O when you suspect logged information may be getting lost. By default, the file appender is buffered.

8.5.6. Performance Considerations

For maximum performance, follow these rules:

1. Turn off all unnecessary logging. Set the threshold to `error`, `fatal`, or `off` to minimize the amount of logging produced.
2. As has been pointed out in Section 8.5.5 *Beware of Buffered Streams*, make sure you are using buffered I/O for logging.
3. If logging statements appear in a performance-critical section of code, they should be wrapped inside an `if` statement that checks the current level of the logger. The "\" character has been inserted where the lines were artificially broken for printing purposes:

   ```
   if ( s_log.isDebugEnbled() ) {
       Object expensiveMsg = "parameter foo " + foo + " expected by the \
   object " +
           frob + " was not supplied. Oh, and the value of baz was " + \
   baz);
       s_log.debug(expensiveMsg);
   }
   ```

   By making the above logging call conditional, we avoid the high cost of constructing the `expensiveMsg` object, if the logging level is to stricter than `debug`.

[12] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/FileAppender.html
[13] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/FileAppender.html#getBufferedIO()
[14] http://jakarta.apache.org/log4j/docs/api/org/apache/log4j/FileAppender.html#setBufferedIO(boolean)

Chapter 9. Persistence Tutorial

This chapter presents a tutorial for using the persistence system. It presumes you are familiar with the concepts covered in Chapter 3 *WAF Component: Persistence*.

This tutorial will first discuss using Data Objects, which is followed by a tutorial on using the features of PDL. Referenced throughout this tutorial is the *Persistence Glossary*.

9.1. Data Objects Tutorial

This section has been assembled to help familiarize you with the concepts and functionality of the persistence layer. This section assumes a familiarity with the concepts discussed in Chapter 3 *WAF Component: Persistence* and in *Persistence Glossary*. It also assumes that the code is being executed within a standard WAF installation.

This tutorial will walk you through the use of object persistence in the WAF. It begins by showing you how to ensure that your server is set up correctly so that the PDL will compile. It then continues with a simple example of how to create a PDL file and how to write the corresponding Java code to interact with the database.

After covering the basics of using standalone Data Objects, the tutorial describes how to associate objects to each other. It discusses several examples, including how to retrieve Data Objects through arbitrary queries. The tutorial concludes with a list of common mistakes that are made by developers.

9.2. Beginning With Data Objects

This section starts with the basic steps that are required to access information in the database using the persistence layer. It begins by discussing PDL and Data Objects. It then discusses how to create the database schema and how it can be represented in PDL. Finally, it covers using the properties defined in the PDL to access the database in your Java code.

9.2.1. Data Objects and PDL: How are they related?

The persistence layer can be looked at as a way to access objects within the database. Examples of objects that can be stored in the database are Users, Groups, Articles, Images, and Email Addresses. In order to abstract out the information regarding object storage within the database, the persistence layer has implemented the concept of a single *Data Object*. Data Objects are used by Java classes to handle all interaction with the database. Because this class is provided, other classes do not need to know how to create, retrieve, update, or delete a given object. Data Objects are defined by their *Attributes* and *Object Key*.

Every Data Object is associated with a single Object Type. Every object type, in turn, is associated with events and operations that specify how information is stored in the database and how information in the database is mapped to Java variables. The role of the PDL file (see Section 9.2.3.1 *The PDL File*) is to provide developers with a mechanism to easily specify that object type.

9.2.2. Setting up the Schema

When defining your persistence layer, you need to work through two aspects of your design:

- A UML model of your Data Objects.

- A database schema to handle the storage of the Data Objects.

Some designers may feel more comfortable starting with a database design and then designing the objects that use the schema, or vice versa. You may begin with either one, but both should be designed as part of the persistence layer. The UML model is typically designed for the Data Objects using a UML modeling tool.

This tutorial will use a hypothetical database schema composed of publications, magazines, articles, paragraphs, and authors. This goal of this tutorial is to help you better understand how this technology can be used with WAF. In practice, the entire data model below will be automatically generated by the persistence layer using the metadata defined in the PDL file. Therefore, none of the data model below will have to be created by the developer.

The tutorial will begin by using two simple objects: a Publication and a Magazine. The schema will be expanded throughout the tutorial. To see the full schema that is used throughout the tutorial, please see Appendix D *PDL Syntax*.

```
create table publications (
    publication_id     integer
                       constraint publications_pub_id_nn
                       not null
                       constraint publications_pub_id_pk
                       primary key,
    type               varchar(400)
                       constraint publications_pub_type_nn
                       not null,
    name               varchar(400)
                       constraint publications_pub_name_nn
                       not null,
    constraint publications_pub_name_un unique(type, name)
);


create table magazines (
    magazine_id       integer
                      constraint magazines_magazine_id_fk
                      references publications
                      constraint magazines_magazine_id_pk
                      primary key,
    issue_number      varchar(30)
);
```

9.2.3. Defining an Object Type in PDL

9.2.3.1. The PDL File

A PDL file is a text file that has a `.pdl` extension and can be parsed using the PDL grammar (Appendix D *PDL Syntax*). It is made up of a model declaration followed by block declarations. These block declarations can be object type definitions, association blocks, Dara Query blocks, and Data Operation blocks.

The only required item is the Model declaration, as this declaration informs the compiler of which namespace to use. It is common to have one for all `DataQuery` and `DataOperation` definitions and one file for each Object Type. However, it is possible to combine all Object Type definitions into a single file. It is also possible to include `DataQuery` and `DataOperations` in their own files or in-line with Object Type Definitions. Data Associations are normally found with one of the Object Type definitions used within the association.

You can find a list of PDL reserved words, as well as the PDL grammar, in Appendix D *PDL Syntax*.

9.2.3.2. Model and Object Type

When creating a PDL file, the first line of the file must be the name of the model that defines the namespace for the block definitions in the PDL file. This is similar to the name of the package in a Java class and is necessary to avoid name collisions of object types that have the same name and which exist in different PDL files. This example will use the `tutorial` model.

The second block of lines within a PDL file can either be a list of namespaces to import (similar to Java's import statement) or the declaration of the object type itself. See Section 9.2.6 *Object Type Inheritance* for more information about importing.

The object type definition follows the import statement. The definition of an object type may include attributes and event definitions. The Publication object type defined below has two attributes defined. One of these attributes, `id`, is also part of the object key, which means that a Publication is uniquely identified by the `id` attribute value.

The first block of code within the object type definition is a list of persistence attributes and mappings of those attributes to database columns for the given object. Note that the attribute names do not need to be the same as the column names. The attributes are a list of Java variables that a given Data Object class can access. Finally, notices that in addition to the java data type at the beginning of the attribute mapping the SQL data type is also present. This SQL data type allows for DDL generator to generate the correct SQL. For a complete list of supported Java types, see Section D.2.2 *PDL Attribute Types*

```
// declare the namespace as "tutorial" using the "model" keyword
model tutorial;

// there are not any import statements because Publication does not
// extend anything.

// next comes the "object type" keyword followed by the name of
// the object type
object type Publication {

    // the first block of code within the object type is a set of
    // mappings from the Attribute Java type and Attribute name to
    // the database column to which they correspond.
    BigDecimal id = publications.publication_id INTEGER;
    String name = publications.name VARCHAR(400);
}
```

9.2.3.3. Object Key

For each object type, you must define an object identifier for the purposes of uniquely identifying object instances at run time. If the object type does not have a super type, the `object key` syntax is used. If the object has a super type, a `reference key` must be declared to indicate how this object joins with the supertype. (See Section 9.2.6 *Object Type Inheritance* for more details.) The following example indicates how to designate that the Publication's `id` attribute is the object identifier or key:

```
model tutorial;

object type Publication {
    BigDecimal id = publications.publication_id INTEGER;
    String name = publications.name VARCHAR(400);
    String type = publications.name VARCHAR(400);

    object key (id);
}
```

This object type definition almost models the SQL that we have defined above but it is missing any mention of the unique constraint. To allow for this and to allow the DDL generator to correctly generate unique constraints the persistence layer allows you to specify either a single property or a set of properties as being unique. The syntax for this is as follows:

```
SINGLE UNIQUE PROPERTY:
object type User {
    BigDecimal id = users.id INTEGER;
    unique String[0..1] screenName = users.screen_name VARCHAR(100);
    ...
}

SET OF UNIQUE PROPERTIES:
object type Node {
    BigDecimal id = nodes.id INTEGER;
    Node[0..1] parent = join nodes.parent_id to nodes.id;
    String[1..1] name = nodes.name VARCHAR(100);

    object key (id);
    unique (parent, name);
}
```

Therefore, in order to correctly model the publications, we need to add the constraint. The correct, full publications object type defintion is below:

```
model tutorial;

object type Publication {
    BigDecimal id = publications.publication_id INTEGER;
    String name = publications.name VARCHAR(400);
    String type = publications.name VARCHAR(400);

    object key (id);
    unique (name, type);
}
```

Note

Attributes cannot have the same name as a pdl reserved word without special quoting. For a list of reserved words and escaping techniques, see Section D.2 *PDL Reserved Words*.

9.2.4. Getting Started with the API

Now that we know how to specify an Object Type using PDL, you can learn how to access the information from Java. The first step is to examine the public API that is provided by the persistence system.

The persistence layer in WAF is implemented in the Java package (http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/package-summary.html). This package contains a set of classes and interfaces for working with persistent objects that store themselves in a relational database. Some of the URLs listed contain a visual break with the "\" character for printing formatting.

- `SessionManager` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ SessionManager.html.

  This class is responsible for initializing the `Session` class. Users of persistence will use it to get a pointer to the `Session` class through `SessionManager.getSession()` (http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ SessionManager.html#getSession()).

- `Session` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/Session.html

  All persistence operations take place within the context of a session. The `Session` object contains methods for creating and retrieving data objects, data queries, and data operations.

- `DataObject` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/DataObject.html

  This interface defines the public methods for Data Objects; see *Data Object*. Data Objects are normally accessed through the use of `DomainObjects` (http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/domain/DomainObject.html).

- `OID` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/OID.html

  This class represents a unique identifier for a given Data Object. It typically holds the information in the object key (*Object Key*) of the object type. For the WAF Object type, the `OID` is the combination of the "acs object type" (e.g., `com.arsdigita.User`) and the "object id". This is typically used to retrieve specific objects from the database.

- `DataQuery` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/DataQuery.html

  This class represents an interface for retrieving arbitrary information from the database. It is the Java equivalent of the PDL data query described in Section 9.4.1 *Data Queries*.

- `DataOperation` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataOperation.html

  This class represents an interface for executing arbitrary DML within the database. It is the Java equivalent of the PDL data operation described in Section 9.4.2 *Data Operations*.

- `DataCollection` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataCollection.html

  This class is used to represent a collection of data objects. `DataCollections` can be used to efficiently iterate over a large set of `DataObjects` and access the values of their properties. These can be filtered in a manner similar to DataQueries.

- `DataAssociation` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataAssociation.html

  This class is similar to the Java Collection interface, in that it provides methods that act on the entire set of associations, such as `add(DataObject object)` (http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataAssociation.html#add(com.arsdigita.persistence.DataObject)), and methods that return other objects to access the associations, such as `cursor()` (http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataAssociation.html#cursor()).

- `DataAssociationCursor` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataAssociationCursor.html

  Like a Java Iterator, this is used to loop over a set of associations. Developers can also add `Filters` to `DataAssociationCursors`, thereby restricting the objects returned to the desired subset.

- `Filter` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/Filter.html

  This class is used to filter the results of associations and collections. For instance, if a developer only wants the associated objects with a name starting with "l" then a filter is used.

- `PersistenceException` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\PersistenceException.html

  This is an unchecked exception that is thrown whenever there is an error within the persistence system. It typically contains an error message as well as some debugging information.

9.2.5. Creating and Retrieving Objects in Java

Now that you have written your PDL files and your database is loaded, the next step is to create Data Objects to access the information in the database. The main access point to Data Objects is through the `Session` class. `Session` objects allow you to instantiate data objects that are part of a logical client session. Specifically, there are three methods that allow you to create and retrieve objects as well as a method allowing you to delete an object.

Suppose you have defined a Publication and want to create a new DataObject so that you can store information about it. The first step is to use the `Session.create(String typeName)` method. You can pass in the fully qualified name of the Object Type and get a Data Object back that corresponds to the defined events. For instance, to create a new Publication, you can do the following (remove the "\" and make it all one line):

```
DataObject publication = SessionManager.getSession().create\
("tutorial.Publication");
```

If you want to retrieve an object that already exists (and you have the `OID`), you can use the `Session.retrieve(OID oid)` method. This is similar to the creation method, except that you are accessing a row that already exists in the database instead of creating a new row (or rows).

If you have an `OID` and you want to delete the object from the database, `Session` provides a `delete(OID oid)` method that allows you to perform the deletion. For most cases, however, you will have the DataObject and can just call the local `delete()` method.

The above methods show how to retrieve or create a single object within the system. The system also provides the ability to retrieve all objects of a given object type. This capability is provided through the method `retrieve(String typeName)`, located in the `Session` class.

Suppose you want to print out the names of all of the Publications in the system whose ID is less than 100. One way to do this would be to loop from 1 to 100, create the corresponding `OID`, and look up each Publication. Another way would be to use a Data Query. However, the simplest way is to take advantage of the `retrieve all` event that is generated for the `Magazine` object type. The following example demonstrates this (remove the "\" and make it all one line):

```
// get all of the Publications in the system.  Calling "retrieve" triggers
// the "retrieve all" event defined in the PDL file.  Note that
// the string passed in to the method is the model name (tutorial)
// followed by the object type name (Publication) separated by a dot (.).
DataCollection pub = SessionManager.getSession().retrieve\
("tutorial.Publication");

// now we want to filter on the ID of the publication
pub.addFilter(pub.getFilterFactory().addLessThan("id", new Integer(100), \
false));

// finally, we can loop through the publications and print out its name
//
// the query to get the information out of the database is executed
// when next() is called for the first time
while (pub.next()) {
  System.out.println(pub.get("name"));
}
```

`DataCollections` are also useful when it is necessary to perform certain tasks on a large number of Data Objects. Suppose you want to add the publications to a java.util.Collection that can then be passed around. The following code will create the java.util.Collection (remove the "\" and make it all one line):

```
DataCollection publication = SessionManager.getSession().retrieve\
("tutorial.Publication");
Collection collection = new ArrayList();

pub.addFilter(pub.getFilterFactory().addLessThan("id", new Integer(100), \
false));

while (pub.next()) {
  collection.add(pub.getDataObject());
}
```

Note

Please note the following:

- Most creation of Data Objects should be handled through the use of Domain Objects. All `ACSObjects` are also DomainObjects and therefore can use the `DomainObject` API to retrieve the correct information. For more information, please see the Section 10.2 *Domain Objects Tutorial*.
- The `DataCollection` class does *NOT* extend the java.util.Collection interface, nor does it extend the java.util.Iterator interface.

9.2.6. Object Type Inheritance

Now that you know how to create and access information for a specific object type, the next logical step in Object-Oriented programming is to create an object type that extends another object type. To address this problem, PDL allows object types to inherit attributes from other object types, similar to inheritance in most Object-Oriented programming languages.

The following example shows a "Magazine" object type extending the "Publication" object type. The definition for the "Magazine" is very similar to that of "Publication" in that it has a block of attributes. It is different, however, in that it does not have an `object key` definition. Rather, it has a `reference key` which indicates how the table containing the Magazine information can be joined to the table containing the Publication information. If the `Magazine` tries to define an `object key`, an error will be thrown.

```
model tutorial.magazine;

// we do not have to import the tutorial model because both object types
// are in the same model.  However, we show the import statement here as
// an example.
import tutorial.*;

object type Magazine extends Publication {
    // we need to specify the size of the String attribute so we know
    // whether it is actually a String or if it is really a Clob
    String issueNumber = magazines.issue_number VARCHAR(30);

    // notice that because it extends Publication, there is not an
    // explicitly "object key" declaration.  Rather, there is
    // a "reference key" declaration and "id" is not defined
    // as one of the attributes
    reference key (magazines.magazine_id);
}
```

9.3. Associations

So far, the documentation has discussed how to create simple Data Objects to access the database. These features, while sufficient to build many systems, lack the ability to relate object types to each other. To address these needs, the persistence system contains the concept of associations.

This document discusses how the persistence layer allows developers to associate Data Objects and how these associations can be saved in the database without needing to involve the Java code in how the associations are actually stored. More concretely, the document first discusses how an association is structured within PDL. It then defines the PDL for `Articles`, `Paragraphs`, and `Authors` and show hows to relate them through *Two-Way Associations*, *Composite Associations*, and *One Way Associations*.

9.3.1. Association Blocks

Most commonly you associate objects to each other when both objects need to know to which objects they are associated. Magazines and Articles, Users and Groups, and Employees and Offices are all examples of this type of association. In the PDL below, an `Article` Object Type and an "association block" are defined to associate the `Articles` to `Magazines`. Association Block definitions are similar to Object Type definitions in that they both have attribute definitions. Instead of the Attribute being set as equal to a single column within the database, the Attribute is set as equal to a *Join Path*.

```
object type Article {
    BigDecimal id = articles.title;
    String title = articles.title;

    object key (articles.article_id);
}

// this is an "association block" associating "articles" and "magazines"
association {
   // note that the Attribute Type is an Object Type (Article)
   // and not a standard Java Type.  Also notice the order of the
   // join path and see the note below.
   Article[0..n] articles = join magazines.magazine_id
                              to magazine_article_map.magazine_id,
                            join magazine_article_map.article_id
                              to articles.article_id;
   Magazine[0..n] magazines = join articles.article_id
                                to magazine_article_map.article_id,
                              join magazine_article_map.magazine_id
                                to magazines.magazine_id;
}
```

Note

The order of the `join path` is important. The information that the developer has must come first. That is, when the developer needs to retreive `articles`, the information he has is the `Magazine` (he needs to know for which magazine to get the articles). Therefore, the first line of the `join path` specifies how to join the magzines table to the mapping table. From then on, it should be in order so that the last section of the `join element` uses the same table as the first section of the next `join element`.

9.3.2. Composite Associations

Composite relationships are a special type of Association. Composite relationships are useful for modeling relationships between objects where a contained object cannot exist outside its container object. The main difference between a Composition and a standard Association is that within a Composition, one of the objects cannot exist without the other.

In the following example, a `Paragraph` is a component of the `Article` (they therefore have a composite relationship). That is, it does not make sense for a Paragraph to exist outside of an article. There are many different ways to signify a relationship as composite but the easiest way is to add the `component` keyword before the component Object Type (in this case, the paragraph is a component of the article)

```
object type Paragraph {
    BigDecimal id = paragraphs.paragraph_id INTEGER;
    String text = paragraphs.text VARCHAR(4000);

    object key (paragraphs.paragraph_id);
}

association {
   Article[1..1] articles = join paragraphs.article_id
                             to articles.article_id;
   // notice the component keyword indicates that if the article does
   // not exist then the paragraph also does not exist
   component Paragraph[0..n] paragraphs = join articles.article_id
                                            to paragraphs.article_id;
}
```

Another way to make the same association is to use the composite keyword on the role that points toward the composite end of an association in order to indicate that the association is a composition. For example:

```
association {
   composite Article[1..1] articles = join paragraphs.article_id
                                        to articles.article_id;
   Paragraph[0..n] paragraphs = join articles.article_id
                                  to paragraphs.article_id;
}
```

The final way to signify the relationship is to use keywords for both object types. This displays the same behavior as the two examples above but it also valid.

```
association {
   composite Article[1..1] articles = join paragraphs.article_id
                                        to articles.article_id;
   component Paragraph[0..n] paragraphs = join articles.article_id
                                            to paragraphs.article_id;
}
```

9.3.3. Role References

Developers ofen only need to be able to obtain associated information in a single direction. For instance, if authors have screen names that are used and can be shared, it is useful to be able to look up the screen name for a given author. However, it may not be as important to look up the author that corresponds to a given screen name. In this case, developers should use a *Role Reference*. In the following example, the developer wants to be able to easily look up a given screen name for an author.

The PDL below can be used to create Object Types for both `ScreenName` and `Author`. Notice that `Author` contains a role reference to a `ScreenName`.

```
model tutorial;
object type ScreenName {
   BigDecimal id = screen_names.name_id INTEGER;
   String screenName = screen_names.screen_name VARCHAR(200);
   Blob screenIcon = screen_names.screen_icon BLOB;

   object key (id);
}

object type Author {
    BigInteger[1..1] id = authors.author_id INTEGER;
    String[1..1] firstName = author.first_name VARCHAR(700);
    String[1..1] lastName = author.last_name VARCHAR(700);
    Blob[0..1] portrait = authors.portrait BLOB;

    // the following line is the role reference.  Notice that it
    // appears in the definition just like an Attribute.  The only
    // difference is that instead of pointing to a column in the
    ScreenName[0..1] screenName =
                     join authors.screen_name_id to screen_names.name_id;

    object key (id);
}
```

9.3.4. Link Attributes

One final feature that is immensely useful for associating objects is the idea of *Link Attributes*. Often, some sort of relationship is needed for associations. For instance, for Magazines, it is useful to include the page number with the Article. The concept of having `Articles` associated with `Magazines` is covered by standard associations but in order to capture a page number with the association, Link Attributes are needed.

```
// this is an "association block" associating "articles" and "magazines"
association {
   Article[0..n] articles = join magazines.magazine_id
                              to magazine_article_map.magazine_id,
                            join magazine_article_map.article_id
                              to articles.article_id;
   Magazine[0..n] magazines = join articles.article_id
                                to magazine_article_map.article_id,
                              join magazine_article_map.magazine_id
                                to magazines.magazine_id;
   // the next line is the Link Attribute.  Note that it also specifies
   // the SQL type of INTEGER so that the DDL generator can correctly
   // create the mapping table with the page_number column.
   BigDecimal pageNumber = magazine_article_map.page_number INTEGER;
}
```

For more information about Link Attributes and their use, see Section 9.8 *Link Attributes*.

9.3.5. Using Java to Access Associations

Now that you have seen how to declare associations within PDL, you can learn the different ways to access the information from Java. In Java, Associations are accessed with two classes: `DataAssociation`, similar to a Java `Collection`, and `DataAssociationCursor`, similar to a Java `Iterator`.

9.3.5.1. Using Standard Associations

```
public Collection getArticles(DataObject obj) {
    LinkedList articles = new LinkedList();
    DataAssociationCursor cursor =
        ((DataAssociation) obj.get("articles")).cursor();

    while (cursor.next()) {
        articles.add(cursor.getDataObject());
    }

    return articles;
}
```

The next example shows how to associate one item with another. In this case, you are associating an `Article` with a `Magazine` by adding the article to the "articles" association. By calling save(), you are signalling for the data object to fire the appropriate insert and update association events (remove the "\" and make it all one line).

```
public void addArticle(DataObject magazine, DataObject article) {
    DataAssociation association = (DataAssociation) magazine.get\
("articles");
    association.add(article);
    magazine.save();
}
```

Note

There are two important things to realize when dealing with adding items to `Associations` and iterating through them:

- Unlike `DataCollection`, `DataAssociation` has been separated into two distinct entities. If you want to loop through the items in the association, or filter or order the association, use a `DataAssociationCursor`. If you want to add or remove items from the association, use the `DataAssociation` object. The `DataAssociation` is a property of the DataObject and is shared by all code accessing the data object. The `DataAssociation`Cursor is essentially a local copy of the association that can be filtered, ordered, and iteratred through without any external consequences.
- While this next item is actually a feature of Domain Objects, it is important to mention here as well. When adding items to associations using the DomainObject (and therefore any ACSObject), there are three choices of which method to use. If the association has an association of 0..n (or any upper bound > 1), developers should use the `add(String propertyName, DataObject dataObject)` method or the `add(String propertyName, DomainObject dobj)` method. If the

association has a Multiplicity of 0..1 or 1..1 (or any upper bound = 1) then developers should use the `setAssociation(String attr, DomainObject dobj)` method.

9.3.5.2. Using Role References

Role References can be treated in exactly the same way as standard associations. The only practical difference between Role References and standard associations is that Role References are one-way associations and standard associations are two-way associations. Thus, everything outlined in Section 9.3.5 *Using Java to Access Associations* also applies to Role References.

9.3.5.3. Using Copmposite Associations

Composite Associations are also similar to standard associations. The main difference is that in a composite association, if one item is deleted, the other does not have any real meaning (e.g., if you delete an Article, the Paragraph is meaningless).

Again, these can be accessed exactly as associations except for one significant difference: when the association between an object and its component is deleted, the component is also deleted. For example, if the association between an Article and a Paragraph were deleted, the Paragraph would be deleted. Also, when the Article is deleted, the Paragraph is deleted.

9.4. Named SQL Events

So far, we have outlined how to interact with the database in a controlled, structured fashion. While creating standard data objects and associations handles most developing needs, sometimes a developer needs to perform database queries that do not fit within the standard realm of objects and associations. It is also often the case that developers are able to perform operations in a single operation that would normally take the system multiple operations. This situation has been handled in two separate ways through the introduction of Data Queries (for selects) and Data Operations (for DML).

9.4.1. Data Queries

Developers often come across situations in which they need information from the database and the persistence layer does not quite do what is needed. Therefore, the system has the ability to execute arbitrary queries through the use of a `DataQuery`.

9.4.1.1. Retrieving Information from the Database

Executing arbitrary queries through `DataQueries` is easy. You can retrieve them in the same way that you retrieve an existing data object, and you can execute the query and loop through the results in the same way that you use a `DataAssociationCursor`.

To begin, you retrieve a query through the `Session` object using its model and name. Then, you can use `next()` to loop through the results.

For example, if you want all paragraphs that show up in the magazine with issue number `5A`. The first step is to define the query within the PDL file. A `DataQuery` definition has four sections. It begins with the declaration of the name of the query followed by data type mappings for each returned attribute. It concludes with two code blocks. The first block, the `DO` block, contains the actual SQL that will be executed. The second block, the `MAP` block, allows the developer to map database columns to attribute names. The attributes are the values that can be accessed from the Java code.

To accomplish the task of retrieving the paragraphs as mentioned above, you could declare the following `DataQuery` in your PDL file:

```
model tutorial;

// the first line indicates that it is a query and the name of the query
query paragraphMagazines {
   // the next section maps the attributes to the java type so that the
   // same type is returned regardless of which database driver is used.
   BigDecimal magazineID;
   BigDecimal paragraphID;
   String issueNumber;
   String text;
   do {
       select m.magazine_id, p.paragraph_id, issue_number, text
       from magazines m, a, magazine_article_map ma, paragraphs p
       where ma.magazine_id = m.magazine_id
       and p.article_id = ma.article_id
   } map {
       magazineID = m.magazine_id;
       paragraphID = p.paragraph_id;
       issueNumber = m.issue_number;
       text = p.text;
   }
}
```

With this PDL definition, it should be easy to see how the following code does what is desired (remove the "\" and make it all one line).

```
DataQuery query = SessionManager.getSession().retrieveQuery\
("tutorial.paragraphMagazines");
query.addEqualsFilter("issueNumber", "5A");
while (query.next()) {
     System.out.println((String)query.get("text"));
}
```

9.4.1.2. Creating Data Objects

The method discussed for retrieving arbitrary information from the database is sufficient to do most of what is needed. However, it is not very convenient since most Java code is written around using `DataObjects`. Therefore, most developers want to be able to retrieve `DataObjects` directly from the `DataQuery`. One way to do this is to create a new DataObject for each row returned by the query and then populate that DataObject with the information retrieved. While this works, it is inefficient and inelegant.

To solve this problem, the `DataQuery` allows the developer to create `DataObjects` directly from the query. The objects can be defined within the `query` statement in a fashion similar to Attribute declarations.

Suppose you want to get all magazines that have authors of articles whose last name starts with a given sequence of characters. This is not a standard association because you are actually going through two separate mapping tables. This could be done by getting all articles with authors that match the criteria and then getting all magazines that contain the articles. However, this option would require two separate database hits. Another option is to perform a query and then for every row create the corresponding data object. A third option is to have the persistence layer create the data objects for you. The following example shows how you can allow the persistence layer to perform the work for you.

The PDL is simply a Data Query with extra Attribute definitions (remove the "\" and make it all one line).

```
query MagazineToAuthorMapping {
    // the next two lines are declaring that objects will be returned
    Magazine magazine;
 Author author;

    do {
        select publications.name, issue_number, publication_id,
               authors.first_name, authors.last_name, author_id
          from magazines, publications, articles, authors,
               magazine_article_map, article_author_map
         where publications.publication_id = magazines.magazine_id
           and magazine_article_map.magazine_id = magazines.magazine_id
           and maagazine_article_map.article_id = article_author_map.\
article_id
           and article_author_map.author_id = authors.author_id
    } map {
        // here we map the attributes of the objects to columns returned
        // by the query.
        magazine.name = publications.name;
        magazine.issueNumber = magazines.issue_number;
        magazine.id = publications.publication_id;
        author.authorID = authors.author_id;
        author.firstName = authors.first_name;
        author.lastName = authors.last_name;
    }
}
```

This can then be accessed with the Java like most other queries. The \ marks where the line has been wrapped for printing purposes.

```
DataQuery query = SessionManager.getSession().retrieveQuery\
("tutorial.MagazineToAuthorMapping");

// the next line adds the filter so that we only get author's whose last
// name begins with the letter "s".  Note that we are using a stanard filter
// because we need to perform a function on the column and we are positive
// that the value is not null.
Filter filter = query.addFilter("lower(lastName) like \
'%' || :lastNamePrefix");
filter.set("lastNamePrefix", "s");

while (query.next()) {
    DataObject myAuthor = query.get("author");
    DataObject myMagazine = query.get("magazine");
    System.out.println("the author I retrieved is " + myAuthor);
    System.out.println("the magazine I retrieved is " + myMagazine);
}
```

9.4.2. Data Operations

As mentioned previously, developers often need to be able to execute arbitrary DML statements that do not fit nicely into the realm of data objects and data associations. To accommodate this need, the system contains the concept of a `DataOperation` that can be used to execute arbitrary DML statements or PL/SQL functions and procedures.

9.4.2.1. Executing Arbitrary DML

Data Operations are similiar to DataQueries in both structure and use. However, while they are retrieved in a fashion similar to DataQueries, they are executed differently. After the query is retrieved, the program can set bind variables, after which it is executed. Suppose you want to create a magazine with ID 4 using all articles in the system that are not yet currently in a magazine. To do this, you could create a new Magazine DataObject, give it an ID of 4, use a `DataQuery` to get all articles not already in a magazine, add those articles to the magazine through the use of associations, and then save the magazine. Alternately, you can use a `DataOperation` and execute a single query.

The `DataOperation` to execute the above query is structured in almost the same way as a `DataQuery`. In fact, it can even have an `OPTIONS` block (although it does not yet have any valid values for the options block). However, since it does not return many different rows of results, it does not allow attribute mappings before the first `do` block. The PDL can be defined as follows:

```
data operation createMagazine {
   do {
      insert into magazine_article_map (magazine_id, article_id)
      select :magazineID, article_id from articles where not exists
      (select 1 from magazine_article_map
      where magazine_article_map.article_id = articles.article_id)
   }
}
```

Now that the operation is defined, you can set the value of the bind variable `magazineID` to the correct value and then execute the operation. This can be done with code such as the following (the \ marks where the line has been wrapped for printing purposes):

```
DataOperation operation = getSession().retrieveDataOperation\
("tutorial.createMagazine");
// we have to pass in an Integer instead of an int so that JDBC can
// handle it correctly
operation.setParameter("magazineID", new Integer(4));
operation.execute();
```

9.4.2.2. Executing PL/SQL

Developers often need to execute PL/SQL procedures and functions. Therefore, it is possible to execute both using a `DataOperation` with additional syntax. Arguments can be passed to functions ans procedures using Parameter Binding.

Note

> The methods described below do not allow users to return cursors from their PL/SQL functions. If this is required, the recommended workaround is to use a `CallableStatement` directly and bypass the persistence layer entirely.

9.4.2.2.1. PL/SQL Procedures

Suppose you want to execute the following PL/SQL procedure:

```
create or replace function myPLSQLProc(v_priority in integer)
as
begin
   insert into magazines (magazine_id, title)
   select nvl(max(magazine_id), 0) + 1, :title from magazine_id;
end;
/
show errors
```

To do this, first include the above statement in your SQL file, so that it will be defined in the database when your package is installed. Next, declare it in your PDL file using a DataOperation:

```
data operation DataOperationWithPLSQLAndArgs {
    do {
          select myPLSQLProc(:title) from dual
    }
}
```

You can then execute this data operation just like any other data operation, after binding the title variable.

It is also possible to use OUT parameters within the DataOperation. To do this, the only additional requirement is for the developer to specify the JDBC type of all of the parameters within the query.

Suppose you want to copy the article with the highest ID into a new row with the ID that you pass into the procedure, and you want back the ID for the row that was copied. You can declare a PL/SQL procedure such as the following:

```
create or replace procedure DataOperationProcWithInOut(
       v_new_id IN Integer,
       v_copied_id OUT Integer)
as
begin
   select max(article_id) into v_copied_id from articles;
   insert into articles (article_id, title)
       select v_new_id, title from articles where article_id = v_copied_id;
   insert into article_author_map (article_id, author_id)
       select v_new_id, author_id from article_author_map
       where article_id = v_copied_id;
end;
/
show errors
```

Adding the JDBC type, the PDL to access this procedure appears as follows:

```
data operation DataOperationProcWithInOut {
    do call {
        DataOperationProcWithInOut(:newID, :copiedID)
    } map {
        newID : INTEGER;
        copiedID : INTEGER;
    }
}
```

To execute this in Java, you simply need to bind the variable and then retrieve the variable using the get(String) method in a fashion similar to retrieving a value from a DataQuery. For instance, to print out the value of the copiedID variable, the following code can be executed:

```
DataOperation operation = getSession().retrieveDataOperation
                                   ("tutorial.DataOperationProcWithInOut");
operation.set("newID", new Integer(4));
operation.execute();
Integer copiedID = (Integer)operation.get("copiedID");
System.out.println("The copied ID was [" + copiedID.toString() + "]");
```

Note

> The `do call` and `OUT` parameters are not available for Postgres because Postgres has not yet implemented `CallableStatement`s or `OUT` parameters.

9.4.2.2.2. PL/SQL Functions

Retrieving a single value back from a function is almost identical to using OUT parameters for procedures. First, declare your PL/SQL in your SQL file. For example, you may define the following:

```
create or replace function DataQueryPLSQLFunction(v_article_id in integer)
return number
is
   v_title varchar(700);
begin
   select title into v_title from articles
    where article_id = v_article_id;
   return v_title;
end;
/
show errors
```

Next, you can define the function as a DataOperation within your PDL file, as follows:

```
data operation DataOperationWithPLSQLAndArgsAndReturnInPDL {
    do call {
       :title = DataQueryPLSQLFunction(:articleID)
    } map {
        title : VARCHAR(700);
        articleID : Integer;
    }
}
```

Finally, you can retrieve the value for `title` just like any normal data query, after binding the `:articleID` variable.

It is necessary to declare the types for each variable within the function whether or not it is an OUT parameter.

9.5. Filtering, Ordering, and Binding Parameters

When retrieving information from the database, developers almost always need to be able to filter and order the results that are returned. Therefore, `DataQuery`, `DataCollection`, and `DataAssociationCursor` objects allow for ordering and filtering. DataQuery and `DataOperation` also allow developers to set arbitrary bind variables within the queries and DML statements. This document discusses how these features are implemented and how using the `Filter` can be overridden to use any arbitrary filtering scheme.

9.5.1. Filtering

9.5.1.1. Overview

The filtering system is complex, in that it allows developers to create complex expressions by combining filters, yet simple in that it provides convenience methods for developers with simple needs.

It is important to realize that by default, `Filters` simply filter the resulting data from the query. For instance, if you have the following:

```
query myDataQuery {
    BigDecimal articleID;
    do {
       select max(article_id) from articles
    } map {
       articleID = articles.article_id;
    }
}
```

and then add the filter "lower(title) like ’b%’", the new query will be:

```
select *
from (select max(article_id) from articles) results
where lower(title) like ’b%’
```

and not:

```
select max(article_id) from articles where lower(title) like ’b%’
```

This can clearly lead to different results.

9.5.1.2. Simple `Filters`

For simple filters, you should use the `addEqualsFilter(String attributeName, Object value)` or the `addNotEqualsFilter(String attributeName, Object value)` methods to filter a `DataQuery`, `DataOperation`, `DataCollection`, or a `DataAssociationCursor` object. These methods take the name of the attribute and its value, create the correct SQL fragment, and bind the variable. If the system is using Oracle or Postgres and the value is null, the system will create the correct `is null` or `is not null` syntax.

In order to specify the filter, you must use the name of the Java Attribute, not the name of the database column. The persistence layer automatically converts the property names to column names using the property mappings defined in the PDL. This layer of abstraction is one of the features that allows developers to change column names without having to update Java code. For example, see the following `DataQuery` defined in PDL (remove the "\" and make it all one line):

```
query UsersGroups {
String firstName;
String lastName;
String groupName;
do{
      select *
      from users, groups, membership
      where users.user_id = membership.member_id and membership.group_id \
= groups.group_id
} map {
      firstName=users.first_name;
      lastName=users.last_name;
      groupName=groups.group_name;
}
```

To retrieve all users whose last name is "Smith", do the following:

```
DataQuery query = session.retrieveQuery("UsersGroups");
query.addEqualsFilter("lastName", "Smith")
while (query.next()) {
   System.out.println("First name = " + query.get("firstName") +
                      "; Last name = " + query.get("lastName") +
                      "; Group = " + query.get("groupName"));
}
```

To get all users whose last name starts with "S", use the addFilter method:

```
DataQuery query = session.retrieveQuery("UsersGroups");
// FilterFactory is explained below
query.addFilter(query.getFilterFactory().startsWith("lastName", "S", false));
while (query.next()) {
   System.out.println("First name = " + query.get("firstName") +
                      "; Last name = " + query.get("lastName") +
                      "; Group = " + query.get("groupName"));
}
```

9.5.1.3. Complex `Filters`

For more complex queries, it is helpful to understand the role of each interface that deals with `Filters`.

- `Filter` — This class represents a single expression for part of a "where" clause. For instance, a Filter could be "foo = :bar" with a value associated with "bar" (e.g. "foo = 3").
- `CompoundFilter` — This class extends `Filter` and provides the ability to add filters together using the `AND` and `OR` keywords.
- `FilterFactory` — This class is responsible for handing out filters. It has methods such as "simple", "equals", "notEquals", "lessThan", "greaterThan", "startsWith", "contains", and "endsWith". If a user is using Oracle or Postgres, all these methods check whether the value is null, and if so, act correctly (e.g., use "foo is null" instead of "foo = null").
- `DataQuery` — This class allows you to add filters as well as get a reference to the FilterFactory. If you need a FilterFactory but do not have a `DataQuery`, use `Session.getFilterFactory()`.

If you want to filter the query based on certain conditions, you can incrementally build up your query as follows:

```
DataQuery query = session.retrieveQuery("UsersGroups");
FilterFactory factory = query.getFilterFactory();

if (beginLetter != null) {
    query.addFilter(factory.lessThan("firstLetter", beginLetter, true));
}

if (lastLetter != null) {
    query.addFilter(factory.greaterThan("firstLetter", beginLetter, true));
}

while (query.next()) {
   System.out.println("First name = " + query.get("firstName") +
                      "; Last name = " + query.get("lastName") +
                      "; Group = " + query.get("groupName"));
}
```

Now suppose you want to get all users with a last name that is the same as the variable `lName` or is `Smith`, and with a first name that matches the variable `fName` or is `John`. You could do this as follows:

```
DataQuery query = session.retrieveQuery("UsersGroups");
FilterFactory factory = query.getFilterFactory();

Filter filter1 = factory.or().addFilter(factory.equals("lastName", lName))
                             .addFilter(factory.equals("lastName", "Smith"));
Filter filter2 = factory.or().addFilter(factory.equals("firstName", fName))
                             .addFilter(factory.equals("firstName", "John"));

query.addFilter(factory.and().addFitler(filter1).addFilter(filter2));
while (query.next()) {
   System.out.println("First name = " + query.get("firstName") +
                      "; Last name = " + query.get("lastName") +
                      "; Group = " + query.get("groupName"));
}
```

These could also have been chained together in order to avoid creating any Filter variables, but this was not done here for clarity.

Finally, if you want to add a bunch of "foo = :foo" statements to a query, you can use the convenience methods provided by `DataQuery`. These methods delegate to FilterFactory and therefore handle the Oracle null problem. For instance:

```
DataQuery query = session.retrieveQuery("UsersGroups");
if (includeFirstName) {
    query.addEqualsFilter("firstName", fName);
}

if (includeLastName) {
    query.addEqualsFilter("firstName", fName);
}

while (query.next()) {
   System.out.println("First name = " + query.get("firstName") +
                      "; Last name = " + query.get("lastName") +
                      "; Group = " + query.get("groupName"));
}
```

Note

When setting dates, java.util.Date objects should be used instead of java.sql.Date objects because java.sql.Dates do not have hours, minutes, or seconds.

When filtering a query that returns data objects (as opposed to simple java types), you must prepend the name of the object attribute. For instance, if you want to retrieve all uses using a query, you would follow the example below. In practice, you want to retrieve a `DataCollection` from the `Session` but we are using a query here for demonstration purposes.

```
query retrieveAllUsers {
   User myUser;
   do {
      select user_id, first_name, last_name from users
   } map {
      user.id = users.user_id;
      user.firstName = users.first_name;
      user.lastName = users.last_name;
   }
}

DataQuery query = session.retrieveQuery("retrieveAllUsers");
// notice how the attribute name corresponds directly to what
// in the map" section of the query and is actully
// the <object type>.<attribute name>
query.addEqualsFilter("user.firstName", fName);

while (query.next()) {
   DataObject user = query.get("user");
   System.out.println("First name = " + user.get("firstName") +
                      "; Last name = " + user.get("lastName") +
                      "; Group = " + user.get("groupName"));
}
```

9.5.1.4. Restricting the number of rows returned

One common feature that is requested of queries, collections, and associations is to be able to restrict the number of rows that are returned. Specifically, in order to create any sort of pagination or to break up a large set of results in to a series of smaller sets it is necessary to restrict the number of rows returned.

Restricting the number of rows a query returns is easy. To do so, you can simple call `setRange(Integer, Integer)` on the data query is question. For instance, if I want results 10-19 for a qury, I can do the following:

```
DataQuery query = session.retrieveQuery("retrieveAllUsers");
query.setRange(new Integer(10), new Integer(20));
while (query.next()) {
   DataObject user = query.get("user");
   System.out.println("First name = " + user.get("firstName") +
                      "; Last name = " + user.get("lastName") +
                      "; Group = " + user.get("groupName"));
}
```

9.5.1.5. Filtering Using Subqueries

The filtering methods described so far handle most situations. However, sometimes developers need the ability to filter based on the results of a subquery. Therefore, the persistence layer provides a mechanism to allow developers to use named queries within filters. Specifically, this is useful within `IN` clauses and `EXISTS` clauses.

Suppose that you want to retrieve all articles written by authors whose last name begins with the letter "b". One easy way to avoid duplicates is to use an `IN` clause. To perform this operation, you can create the following two DataQueries:

```
query retrieveArticlesBasedOnAuthor {
      BigDecimal authorID;
      do {
          select article_id
```

```
            from authors, author_article_map
           where authors.author_id = author_article_map.author_id
             and lower(last_name) like :lastName || '%'
      } map {
          authorID = authors.author_id;
      }
}

query retrieveSelectedArticles {
      BigDecimal articleID;
      String title;
      do {
          select article_id, title from articles
      } map {
          articleID = articles.article_id;
          title = articles.title;
      }
}
```

Next, simply retrieve one query and add the other query as part of the filter, as follows (remove the "\" and make it all one line):

```
Session session = SessionManager.getSession();
DataQuery query = session.retrieveQuery("tutorial.retrieveSelectedArticles");
Filter filter = query.addInSubqueryFilter("articleID", \
"tutorial.retrieveAuthorsWithParam");
// we have to set the value for "lastName" since it is a bind variable
// in the subquery we added.
filter.set("lastName", "b");
System.out.println("The following articles have at least one author whose " +
                   "last name starts with 'b'");
while (query.next()) {
    System.out.println(query.get("title"));
}
```

The code above will actually execute the following SQL:

```
select article_id, title from articles
where article_id in (select article_id
                       from authors, author_article_map
                      where authors.author_id = author_article_map.author_id
                        and lower(last_name) like ? || '%'

with ? = "b"
```

Note

While there are other, possibly better ways to obtain the same result, this example is used to demonstrate how the feature works, not as an authoritative example of writing queries.

9.5.1.6. Filtering Using Functions

The filtering methods discussed so far work well when the developer only needs to filter directly off of columns or values. However, they do not work well if the developer wants a case-insensitive filter or needs to use some other function to manipulate the data in the columns.

To meet this need and to appropriately handle null values, the system provides a method within `FilterFactory` named `compare` that allows developers to pass in two expressions and have the system compare them to each other.

The most common use case for this is a case-insensitive comparison where the developer wants to know whether a string exists in a column, but does not care about the case. For instance, suppose a developer wants to retrieve all articles titled "Disaster Strikes," but does not care about the capitalization. He could use use the following code to achieve this:

```
DataCollection pub = SessionManager.getSession().retrieve("tutorial.Articles");
Filter filter = pub.addFilter(pub.getFilterFactory().compare("upper(title)",
                                                     FilterFactory.EQUALS,
                                                     ":title"));
// we set the title to all upper case so that we do not make oracle do
// it for us which would be slower
filter.set("title", "DISASTER STRIKES");
```

If the developer actually wants all articles with either the word "Disaster" or "Strikes," he can do the following:

```
DataCollection pub = SessionManager.getSession().retrieve("tutorial.Articles");
FilterFactory factory = pub.getFilterFactory();
Filter disasterFilter = factory.compare("upper(title)", FilterFactory.CONTAINS,
                                        ":disasterTitle"));
filter.set("disasterTitle", "DISASTER");
Filter strikesFilter = factory.compare("upper(title)", FilterFactory.CONTAINS,
                                       ":strikesTitle"));
filter.set("disasterTitle", "STRIKES");
pub.addFilter(factory.or()
                     .addFilter(disasterFilter)
                     .addFilter(strikesFitler));
```

The important thing to realize is that it this method will handle problem with null values faced by Oracle and Postgres, whereas using a standard `simple` filter will not.

9.5.2. Ordering

Use the `addOrder(String order)` method to order a `DataQuery`, `DataCollection`, and a `DataAssociationCursor` object. The `addOrder` method takes a *`String`* as its only parameter and the format of the string is the optional object type following by a dot and then then required attribute name ([<*`object type name`*>.]<*`attribute`*>) you wish to order by.

The string parameter passed to the `addOrder(String order)` method is used in an `ORDER BY` clause, which is appended to the `SELECT` statement. The order is specified by constructing a string representing the `ORDER BY` clause, but instead of specifying column names, you specify attribute names. The persistence layer automatically converts the attribute names to column names using the property mappings defined in the PDL. For example, see the following `DataQuery` defined in PDL (remove the "\" and make it all one line):

```
query UsersGroups {
   String firstName;
   String lastName;
   String groupName;
   do{
      select *
      from users, groups, membership
      where users.user_id = membership.member_id and membership.group_id \
= groups.group_id
   } map {
      firstName=users.first_name;
      lastName=users.last_name;
      groupName=groups.group_name;
   }
}
```

You can order this query by the user's last name, as follows:

```
DataQuery query = session.retrieveQuery("UsersGroups");
query.addOrder("lastName asc");
while (query.next()) {
   System.out.println("First name = " + query.get("firstName") +
                      "; Last name = " + query.get("lastName") +
                      "; Group = " + query.get("groupName"));
}
```

Finally, you can build up the ORDER BY string in the same way that you build up a filter. For instance:

```
DataQuery query = session.retrieveQuery("UsersGroups");
query.addOrder("lastName asc")
if {careAboutGroupName} {
    query.addOrder("groupName");
}
while (query.next()) {
   System.out.println("First name = " + query.get("firstName") +
                      "; Last name = " + query.get("lastName") +
                      "; Group = " + query.get("groupName"));
}
```

The ORDER BY string is any valid ORDER BY clause, except that you specify property names, not column names.

9.5.3. Binding Parameters

Use the `setParameter(String parameterName, Object value)` method to bind an arbitrary variable within a `DataQuery`, `DataOperation`, `DataAssociationCursor`, or `DataCollection`. The method `getParameter(parameterName)` allows you to retrieve the value of a set parameter. The `setParameter` takes in a string that should match the string within the defined SQL. The Object it takes should specify the value.

This functionality is useful for complicated queries that involve embedding parameters. For example, see the following `DataQuery` defined in PDL (bind variables are in *bold*):

```
query CategoryFamily {
  Integer level;
  BigDecimal categoryID;
  String name;
  String description;
  Boolean isEnabled;
  do {
     select l, c.category_id, c.name, c.description, c.enabled_p
        from (select level l, related_category_id
                 from (select related_category_id, category_id
                          from cat_category_category_map
                          where relation_type = :relationType)
                 connect by prior related_category_id = category_id
                 start with category_id = :categoryID) m,
            cat_categories c
        where c.category_id = m.related_category_id
  } map {
     level = m.l;
     categoryID = c.category_id;
     name = c.name;
     description = c.description;
     isEnabled = c.enabled_p;
  }
}
```

This query first retrieves all mappings that are of a particular type (e.g., "child" mappings as opposed to "related" mappings). It then does a "connect by" to get all the parents (or all the related categories). Finally, it joins this result with the original categories table so that it can have the `name`, `categoryID`, `description`, and `isEnabled` for each of the selected categories. Without being able to set the `categoryID` and `relationType` variables, you could not perform this query. Creating a separate query for each kind of `relationType` does work, but there is no way to account for every possible `categoryID`.

After the query is defined, it can be used as follows:

```
DataQuery query = session.retrieveQuery("CategoryFamily");
query.setParameter("relationType", "child");
query.setParameter("categoryID", "3");
while (query.next()) {
  System.out.println("We retrieved Category " + query.get("name")) +
                     "with ID = " + query.get("categoryID"));
}
```

9.5.3.1. Binding Collection Parameters

Thus far, the document has discussed binding parameters that contain a single value, that is, parameters that are used in comparison clauses. While these types of parameters cover most cases, sometimes you will want to be able to have bind variables that represent many different values. Specifically, you may want to be able to get rows that meet specific criteria and are `IN` a given set of rows. To provide this functionality, the system has the ability to take a java.util.Collection as the value for a bind variable and then expand the Collection so that it works correctly.

For instance, if you want all Articles whose IDs were in a given collection, you can write a method such as the following (remove the "\" and make it all one line):

```
public DataCollection retrieveArticles(Collection articlesToRetrieve) {
    DataCollection articles = SessionManager.getSession().retrieve\
("tutorial.Articles");
    Filter filter = articles.addFilter("id in :articlesToRetrieve");
    filter.set("articlesToRetrieve", articlesToRetrieve);
    return articles;
}
```

9.5.4. How it Works

You may be wondering how to add filters and bind variables after retrieving the query. The persistence system does not actually execute the required SQL query until the first call to `next()`. Therefore, the `DataQuery` (or `DataCollection` or `DataAssociationCursor`) can be passed around and have `Filters` and ORDER BY statements added and variables bound before the first element is retrieved. Then, when the first element is retrieved, the query is executed and the code can no longer add or remove filters or ordering criteria.

9.6. Common Mistakes

9.6.1. Introduction

As developers use the persistence layer of the WAF, several types of errors are made by many developers. This document tries to highlight some of those errors to help with debugging tasks. Another document that may be of interest to developers having problems with persistence is the Section 9.10 *Frequently Asked Questions*.

9.6.2. Use of Semi-Colons; Invalid Character Error

One of the most common and most easily fixed errors is that of an invalid character. The error is typically something such as:

```
com.arsdigita.persistence.PersistenceException: ORA-00911: invalid character
```

This error is caused because the developer accidentally included a semi-colon at the end of a SQL statement. For instance, the following `retrieve` event will cause the above error because of the semi-colon (in *bold*) at the end:

```
retrieve {
   do {
      select publication_id, name
      from publications
      where publication_id = :id;
   } map {
      id = publications.publication_id;
      name = publications.name;
   }
}
```

To fix the error, simply remove the semi-colon so that the event definition appears as follows:

```
retrieve {
   do {
      select publication_id, name
      from publications
      where publication_id = :id
   } map {
      id = publications.publication_id;
      name = publications.name;
   }
}
```

This error occurs is because the query is fed directly to the database using the JDBC driver, which does not exptect a trailing semi-colon.

9.6.3. Incorrect Attribute Mappings

When you receive an error indicating that an attribute cannot be found or an attribute name is invalid, make sure that there are no typos. Simple typos cannot be caught by the compiler and cause many errors.

9.6.4. Problems with Transactions

Typically, two types of mistakes occur when dealing with Transactions within the WAF.

The first type of mistake is trying to access Data Objects (and hence the database) outside of a transaction. This typically happens when the developer tries to write a custom dispatcher that bypasses the standard WAF dispatcher and does not remember to include transaction management. To fix this, make sure that the custom dispatcher properly opens and closes transactions using appropriate WAF APIs.

The second type of mistake is the exact opposite of the first: trying to nest transactions. Unless you have written a custom dispatcher that bypasses the standard WAF dispatcher, you should never have to open or close a transaction. Trying to do so within the standard framework of the WAF will result in an error. In almost all cases, developers building on top of the WAF should not try to manage their own transactions.

For more information, see the document on Section 9.7 *Transaction Management*.

9.6.5. Join Paths

Common problems:

- Non-continuous join paths -- the second column's table in each join path element must match the table of the next join path element's first column. That is, the path must look somthing like `join A.B to C.D, join C.E to E.F` In this example, table `C` is the last table of the first half and the first table of the second half.
- Join paths are too long -- MDSQL only supports join paths composed of 1 or 2 join path elements. If paths are longer than that, it becomes impossible to generate events correctly given the current implementation.
- Join path does not start in the primary table -- a join path must always start in the same table as the reference key or object key. If it does not, there is a gap and the event cannot be generated.

9.6.6. Dynamic DDL Generation

Dyanmic DDL Generation works well when the PDL is specified correctly. However, it is easy to think that the PDL is written to represent the model correctly when in fact you are missing a simple item to two. Below are some symptoms that you may see with your generated DDL and some suggestions as to how you may be able to fix them.

Data Model Problem	Likely Cause in PDL
Missing columns	Properties were specified without a column or join path

Data Model Problem	Likely Cause in PDL
Missing foreign keys	Object Model doesn't use associations, but data model does, e.g. Message object has BigDecimal authorID property instead of User author property, in these cases an easy fix is to create the author association without removing the authorID attribute so that existing java code continues to work.
Missing not null	The property in PDL is [0..1] when it should be [1..1].
Extra not null	The property in PDL is [1..1] when it should be [0..1].
Missing on delete cascade	The property in PDL should be declare component or composite.
Missing unique constraint	The property(ies) in PDL should be declared unique.

9.6.7. Debugging Persistence

When developing code using persistence, you may have a difficult time tracking down the reason for a `PersistenceException`. Turning up the logging level can help you understand what is going on. The following Log4J loggers `com.redhat.persistence.Session` and `com.redhat.persistence.engine.rdbms` can be turned to the info log level to help see what is going on. The latter will log all SQL being sent to the database.

9.7. Transaction Management

Within WAF, the Servlet Dispatchers handle all transactions for the developer. This document clarifies the decisions that have been made regarding transactions within the WAF, and briefly discusses how to manage transactions outside of the WAF.

9.7.1. Transactions within the WAF

The WAF imposes several restrictions on the developer that are important to understand:

- Only one transaction may be open at a time for a given thread. More concretely, it is not possible to nest transactions.
- Writable Data Objects cannot be cached across transactions, because aborting a transaction requires some way to either rollback or invalidate all `DataObjects` using the transaction. However, it is possible to cache read-only data objects since they are loaded from the database a single time and then never modified and they never again need to access the database. However, when doing this, it is important that all read-only objects are removed from memory when a writable object is retrieved.

One feature to realize is that if a SQLException is thrown when using Postgres then the transaction is aborted and it is no longer possible to interact with the database. Therefore, it is necessary to check for all conditions that could cause an error to arise before actually making a call to the database.

When programming in the WAF, unless you are writing a dispatcher or initializer, you will most likely not need to deal with transactions. However, if you do, the process is simple and is outlined in the next section.

9.7.2. Manual Transaction Management

When you manage your transactions locally, your starting point is the `SessionManager` class. From there, you can get access to the `Session` class and then to the Transaction. Specifically, the `Session` provides access to the TransactionContext, which has methods for beginning a transaction, committing a transaction, and aborting a transaction. For example, the following code will open a transaction, execute the necessary code and then either commit or abort the transaction:

```
import com.arsdigita.persistence.Session;
import com.arsdigita.persistence.SessionManager;
import com.arsdigita.persistence.TransactionContext;

Session session = SessionManager.getSession();
TransactionContext txn = session.getTransactionContext();

try {

    txn.beginTxn();

    // do some work here

    txn.commitTxn();

} catch (Exception e) {
    txn.abortTxn();
}
```

Transactions cannot be nested. If you have to make sure that you are in a transaction, use the inTxn() method of TransactionContext, as in the following example:

```
import com.arsdigita.persistence.Session;
import com.arsdigita.persistence.SessionManager;
import com.arsdigita.persistence.TransactionContext;

Session session = SessionManager.getSession();
TransactionContext txn = session.getTransactionContext();

boolean openedTransaction = false;
try {

    if (!txn.inTxn()) {
       openedTransaction = true;
       txn.beginTxn();
    }

    // do some work here

    if (openedTransaction) {
       txn.commitTxn();
    }
} catch (Exception e) {
    if (openedTransaction) {
        txn.abortTxn();
    }
}
```

9.8. Link Attributes

When modeling many real-world problems, it is common to encounter situations that require associating data with a link between two objects. This document describes how to model these situations using the persistence system.

9.8.1. Introduction

When creating a model for use in a particular domain, there are usually cases in which a simple association between two object types will not suffice. A `Group` may contain `Users` through a members Association (see Section 9.3 *Associations*), and for each member that `Group` may want to record the date on which the `User` became a member of the `Group`. This is a case in which the link between two objects must carry more information than just that the two objects are associated. The UML terminology for this is a *qualified association* and it is described with the following model.

```
     ____________                           ____________
    |   Group    |                         |    User    |
    |------------|                         |------------|
    |            | n                     n |            |
    |            |------------------------>|            |
    |            |           |     members |            |
    |____________|           |             |____________|
                             |
                             |
                             |
                    _________|________
                   |                  |
                   |------------------|
                   |  membershipDate  |
                   |                  |
                   |__________________|
```

A model like the one depicted above can be created in sql with the following 3 tables.

```
create table groups (
      group_id     integer primary key;
      name         varchar(300);
      ....
);

create table users (
      user_id      integer primary key;
      name         varchar(300);
      email        varchar(100);
      ....
);

create table group_member_map (
      group_id     integer references groups;
      member_id    integer references users;
      membership_date date default sysdate;
      primary key (group_id, member_id);
);
```

This association can then easily be transformed to PDL through the use of the association keyword. When defining an association between two objects using the association block, it is possible to add extra properties that can store information associated with a particular link between two objects.

```
    object type Group {
        BigInteger[1..1] id = groups.group_id;
        String[1..1]     name = groups.name;
        // ...
    }

    object type User {
        BigInteger[1..1] id = users.user_id;
        String[1..1]     name = users.name;
        String[1..1]     email = users.email;
        // ...
    }

    association {
        Group[0..n] groups = join users.user_id to group_member_map.member_id,
                             join group_member_map.group_id to groups.group_id;
        User[0..n]  members = join groups.group_id to group_member_map.group_id,
                              join group_member_map.member_id to users.user_id;

        Date[1..1]  membershipDate = group_member_map.membership_date;
    }
```

Note

In this case, all events are auto-generated and will only work if the link attribute (in this case, membershipDate) is defined within the mapping table. If the DDL generator is used, the above association definition will create the correct mapping table.

It is also possible to have a link attribute that returns a full object type (e.g. another User) rather than a simple object type (e.g. a Date). Everything in this example is the same if you replace `Date` with `User`. The only extra assumption made is that the full object type (e.g. User) only has a single Object Key and that key is the item stored in the mapping table (e.g. user_id must be stored in the mapping table in the case of User).

9.8.2. Link Attributes API

Once the PDL has been written to describe link attributes, the java API may access the values using the following methods (remove the "\" and make it all one line):

- `DataObject DataAssociation.add(DataObject object)` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataAssociation.html#add(com.arsdigita.persistence.DataObject)
- `Object DataAssociationCursor.getLinkProperty(java.lang.String)` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataAssociationCursor.html#getLinkProperty(java.lang.String)
- `classname` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/\ DataAssociationCursor.html#getLink()

9.8.2.1. Creating and Initializing Links

The `DataAssociation.add(DataObject object)` method is used when creating a link between two objects. If the association was defined with the association keyword, this method returns the link DataObject. The link DataObject can then be used to set any link attributes that have been defined. This is illustrated in the following example:

```
// Put the current session in a variable for convenience.
Session ssn = SessionManager.getSession();

// Get a user.
DataObject user = ssn.retrieve(new OID("User", ...));
// Get a group.
DataObject group = ssn.retrieve(new OID("Group", ...));

// Get the "members" association so that we can add a member to the
// group.
DataAssociation members = (DataAssociation) group.get("members");

// Add the user to the association.
DataObject link = members.add(user);
java.util.Date now = new java.util.Date();
link.set("membershipDate", now);

// Persist the changes.
group.save();
```

Note

Only `group.save()` is called to persist the changes. Changes to the links are automatically persisted when the parent object is saved.

9.8.2.2. Reading Link Attributes

The `DataAssociationCursor.getLinkProperty(java.lang.String)` method can be used to read the values of a link attribute while iterating over a cursor.

```
// Put the current session in a variable for convenience.
Session ssn = SessionManager.getSession();

// Get a group.
DataObject group = ssn.retrieve(new OID("Group", ...));

// Get the members association.
DataAssociation members = (DataAssociation) group.get("members");

// Iterate over the members of the group and print out the
// membership date.
DataAssociationCursor cursor = members.getDataAssociationCursor();
while (cursor.next()) {
    // Fetch the users email for the current row.
    String email = cursor.get("email");
    // Fetch the link property for the current row.
    java.util.Date membershipDate =
        (java.util.Date) cursor.getLinkProperty("membershipDate");
    System.out.println(email + ": " + membershipDate);
}
```

9.8.2.3. Updating Link Attributes

The `DataAssociationCursor.getLink()` method can be used to fetch the DataObject that represents a link. This DataObject can then be used to modify the link and persist the results.

```
// Put the current session in a variable for convenience.
Session ssn = SessionManager.getSession();

// Get a group.
DataObject group = ssn.retrieve(new OID("Group", ...));

// Get the members association.
DataAssociation members = (DataAssociation) group.get("members");

// Iterate over the members of the group and modify each link.
DataAssociationCursor cursor = members.getDataAssociationCursor();
while (cursor.next()) {
    // Fetch the link object for the current row.
    DataObject link = cursor.getLink();

    // Get the membership date from the link data object.
    java.util.Date oldDate = (java.util.Date) link.get("membershipDate");

    // Incrememnt the membership date by 1 millisecond.
    java.util.Date newDate = new java.util.Date(oldDate.getTime() + 1);

    // Set the new date
    link.set("membershipDate", newDate);
}

// Persist all the changed links.
group.save();
```

Note

Only `group.save()` is called to persist the changes. Changes to the links are automatically persisted when the parent object is saved.

9.8.2.4. Filtering and Ordering based on Link Attributes

To access the values of a link attribute when filtering or setting the order of a data association, simply put the special `link` prefix in front of the link attribute name.

```
// Put the current session in a variable for convenience.
Session ssn = SessionManager.getSession();

// Get a group.
DataObject group = ssn.retrieve(new OID("Group", ...));

// Get the members association.
DataAssociation members = (DataAssociation) group.get("members");
```

```
// Iterate over new members of the group.
DataAssociationCursor cursor = members.getDataAssociationCursor();

// Get yesterday’s date.
java.util.Date yesterday =
    new java.util.Date(System.currentTimeMilis() - 1000 * 60 * 60 * 24);

// Restrict to recent members
Filter f = cursor.addFilter("link.membershipDate > :date");
f.set("date", yesterday);

// Order by membership date, descending
cursor.setOrder("link.membershipDate desc");

while (cursor.next()) {
    ...
}
```

9.9. Dynamic Object Types

Thus far, the documentation has assumed that developers know the types of objects that they will be dealing with at compile time. More concretely, it has assumed that all SQL table definitions and PDL files have already been defined by the developer. In some situations, however, this assumption is invalid.

Sometimes developers want to provide site administrators with an interface to create their own types of objects and store custom information within the database. Since it is not possible to anticipate all information anyone will ever want to store in the system and it is not feasible to require developers to manually create all types, the system contains the notion of a `DynamicObjectType` that provides developers with the ability to create data schemas (and persist data to that schema) at run time. This is one of the main motivations of the Metadata-Driven SQL.

The implementation of dynamically storing information has been broken into two parts. The first part, dynamically creating new object types, uses the `DynamicObjectType` class. The second part, dynamically associating objects to each other, is handled by `DynamicAssociation`.

9.9.1. Using Dynamic Object Types

In order to provide developers with the ability to declare objects at run time instead of at compile time, the persistence system has the notion of a DynamicObjectType. A Dynamic Object Type allows users to dynamically create and modify Object Types. It can be used to create a subtype of an existing object type, add and remove Attributes and RoleReferences, and perform many other tasks related to the new object type.

The way that this is done is relatively simple. The developer uses methods within the class to set the Model and object type name and create attributes. When the code saves the current object type, it does the following:

1. Creates the DDL that needs to be executed to store information about the specific object type.
2. Generates the PDL that needs to be stored so that the object type will be instantiated when the server restarts.
3. Generates the events for storing information in the schema in memory so that the ObjectType returned can be used within `DataObjects`.

4. Executes the DDL to create the schema. Unfortunately, this causes the system to commit the current transaction, so it is important that you do not do anything else in the same transaction as saving the DynamicObjectType.
5. Assuming the DDL executed successfully, saves the PDL that was generated to a table within the database. The information in this table is then read in on every server startup, so that the DynamicObjectTypes essentially become permanent.
6. Regenerates events for any children of this object type that may have been affected by any changes that were made.

9.9.2. Dynamic Associations

The DynamicObjectType supports creating new object types. However, it does not allow the dynamic creation of associations that involve static object types. Specifically, if a developer wanted to dynamically create an association between Users and the Publications that they read, the only way to do that would be to add that information to the PDL file, recompile and restart the server. To address this problem, the system provides a `DynamicAssociation` class.

This class saves information about the association following approximately the same steps as the DynamicObjectType. The main difference between the classes, besides the types of events they generate, is that the DynamicAssociation can be used to map two dynamically defined Object Types, two statically defined Object Types, or one dynamically defined Object Type and one statically defined Object Type. This is in contrast to the DynamicObjectType that only deals with dynamically defined Object Types.

9.10. Frequently Asked Questions

This document contains a list of the most frequently asked questions concerning the WAF persistence layer. For terminology and definitions, please see the *Persistence Glossary*.

1. General Questions

1.1. Why does the Red Hat WAF have a persistence layer? Why doesn't it just create the SQL when and where it is needed?

A persistence layer has been developed as part of the WAF for the purpose of encapsulating database access routines. This is desirable for at least two reasons: abstraction and database independence. By encompassing the SQL statements in Java objects, the database access routines are developed in a scalable environment. Also, the applications are isolated from the schema changes (which may occur in future versions) since the database access routines are encapsulated by a standard API. The second reason, database independece, is desirable because it means that the Java code can easily be ported from one database to another by changing a small set of centrally located files (as opposed to modifying every file that interacts with the database).

1.2. Why does the Red Hat WAF have a persistence layer instead of just using standard EJB? Why is Red Hat reinventing the wheel?

WAF is primarily a framework for building web applications, and Red Hat's reasoning for its current architectural direction is best summarized by Sun's own J2EE Blueprints at http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.

In addition, there are numerous examples that one could concoct where an EJB server (at least initially) could be deemed to be an overkill given the problem being tackled. This is the sledgehammer-to-crack-a-nut problem. In essence, the J2EE specification does not mandate a 2, 3, or multitier application model, nor realistically could it do so. The point is that it is important to use appropriate tools for a given problem space.

Given that the most common usage for WAF deployment is, to use Sun's term, the "Web-centric application scenario", Red Hat opted for the "3-tier Web-centric" architecture, in which "[t]he Web container is essentially hosting both presentation and business logic, and it is assumed that JDBC (and connectors in the future) will be used to access the EIS resources."

Coarse-grained interoperation of WAF with other systems would certainly be a sound application of EJB or JMS. However, Red Hat does not currently have clear requirements in this area. More activity is anticipated in this area during future WAF implementations, initiated both by Red Hat and by others building on top of WAF.

(As an aside, commercial J2EE containers such as WebLogic and Oracle's 9i typically make it a goal to provide a "simple mechanism for balancing load across several machines without too much thinking," even when you are only using the Servlet API, as opposed to EJBs.)

Regarding the other reasons for using EJBs outside of distributed objects, the following points should clarify the extent to which Red Hat is "reinventing the wheel":

1. Persistence: There are two standard types of persistence: Bean-Managed Persistence (BMP) and Container-Managed Persistnce (CMP). BMP does not provide much advantage, and CMP, unfortunately, does not provide automatic SQL generation. In short, the WAF persistence layer is designed to address a different set of requirements than CMP.
2. Connection pooling: Red Hat is not reinventing connection pooling; rather, its code is writtten against the J2EE-standard API for connection pooling, the JDBC 2.0 Optional Package at http://java.sun.com/products/jdbc/articles/package2.html.
3. Transaction monitoring: EJB's declarative-style transaction management is one of its more attractive features, but not compelling enough for Red Hat to make it the deciding factor in using EJBs. Red Hat does not "reinvent" transaction management, either; it simply leaves the applications responsible for managing transactions.

Red Hat has found, through input from prospective customers, partners, and analysts, that the market perspective on the necessity and appeal of the "full J2EE architecture" is not clear-cut. Being able to run WAF atop a J2EE web container alone has benefits of its own: simplicity of development and deployment, low run-time overhead, and broad availability of stable open-source/affordable implementations. In other words, 3-tier Web-centric is quite defensibly the appropriate tool for Red Hat's primary problem space.

1.3. What are the primary benefits/goals of the persistence API?

The persistence API provides the following benefits:

- It provides an abstraction for physical data models for greater database independence.
- It isolates the java classes from data model changes (e.g., if a column is renamed, no Java code must be changed).
- It provides support for user-defined object types.
- It decreases the amount of time it takes to implement a new object type.
- It can automatically generate SQL statements that can be run on different databases. Thus, it is possible to write one PDL file that can be used on N databases. Without persistence, a developer may be required to write N different DDL scripts and then N different sets of SQL statements to use the tables.

2. Developer Specific Questions

2.1. I am new to the persistence API. Where do I start?

Start with this book's table of contents which lists all available persistence documents. In addition, the Section 10.2 *Domain Objects Tutorial* introduces the API used by the WAF and provides you with code samples for implementing your own Domain Objects.

Additionally, you may want to consult the document on Section 9.2 *Beginning With Data Objects*, as well as material on Section 9.5 *Filtering, Ordering, and Binding Parameters*.

2.2. If I am creating an object type what kind of metadata do I need to define?

The best place to find a full explaination and examples is in the Section 9.2.3 *Defining an Object Type in PDL*. However, as a short summary, you must specify the following:

- For each attribute:
  - A mapping of the attribute to the database column in which it resides.
  - The SQL type of the attribute. For instance, if the data type of the attribute is a string, the SQL type might be VARCHAR(20) or VARCHAR(4000) or even CLOB.
- For each association, it is important to specify the *Join Path* that the persistence layer will know how to join the SQL tables to each other.
- For each data query, a mapping from the returned attribute name to the data type of the attribute is required. This allows the code whether an INTEGER SQL type should be an INTEGER data type or actually a BigDecimal data type.

All of the metadata is required so that the persistence layer can successfully generate a data model as well as all of the events to manipuate the object.

2.3. Where can I find example PDL (Persistence Definition Language) files that define object types?

At the top level of any project you will find a `pdl` directory. In there, you will find many PDL files that should provide plenty of examples.

2.4. Where do I put my PDL files?

PDL files should be kept in a directory at the same level as the SQL and src files. If the PDL does not contain any database specific queries (e.g., something containing a `connect by` statement) then it should end in (`.pdl`). Otherwise, you should create a copy of the query for each database and name each copy appropriately: `.ora.pdl` for Oracle or `.pg.pdl` for PostgreSQL.

2.5. I want to create a new object type that extends ACSObject. What do I do?

When you extend an object type, the new object type will inherit all the properties (attributes and role references) of the super type. A child object type can override the parent's attributes (but not role references), as long as the new attribute does not violate any constraints placed on the supertype's attribute. For example, suppose that the parties object type defines an email attribute that is required. That is, parties.email has multiplicity of 1..1. You can create a persons object type that extends parties. You cannot make the email address optional in the persons object type because it is already required by the supertype.

Suppose you are defining an object type named "ChewingGum" that extends ACSObject. To specify that an object type extends another, do the following:

- Before defining your object type but after declaring your model, make sure to import the ACSObject type.

  ```
  import com.arsdigita.kernel.*;
  ```

- When defining your object type, specify that the supertype of the "Chewing Gum" object type is "ACSObject" and import the namespace for ACSObject. In the PDL file, you currently write:

  ```
  object type ChewingGum extends ACSObject {
      ---
  }
  ```

- When you define your table in the database, make the primary key reference the `acs_objects` table. This is only required, however, when you manually specify your SQL. In most cases, defining the PDL will be enough to get your started (you must, however, still load the auto-generated table). The following shows how to manually define the table.

  ```
  create table acs_objects (
         object_id          integer primary key
  );
   create table chewing_gum (
         gum_id             integer primary key
                            references acs_objects(object_id)
  );
  ```

- When you create your Java class, you can extend the ACSObject class in the `com.arsdigita.kernel` package.

  See Section 9.2.6 *Object Type Inheritance* for a longer example.

2.6. What is a `DataQuery` and how do I use it?

A `DataQuery` encapsulates SQL queries. You can retrieve a `DataQuery` by name from the PDL file, set filters on its attributes, execute the event, iterate over the result set, and access the attribute values.

Underneath, the actual SQL query and data model used are hidden from application code.

For more information, see the documentation on Section 9.4 *Named SQL Events*.

2.7. How do I specify a custom SQL query?

You can create a `DataQuery` by specifying a SQL query and as many attribute/column mappings as you like. You can then define bind variables or set filters on that query to gain the final query you wish.

2.8. What is an Object/Reference Key and why do I need one?

The `object key` (*Object Key*) declared in the PDL files is used by the persistence system to uniquely identify a particular object in the database. For instance, the `object key` for `acs_object` is "id", which is the name of the attribute that maps to the `object_id` column in the `acs_objects` table.

Object keys can contain singule values or they can be compound (they can be specified using 1 to N attributes). For more information, see Section 9.2.3.3 *Object Key*.

2.9. How do I retrieve a `DataQuery`, `DataCollection`, or `DataAssociation` in the order I want (e.g., there is a `sort_order` link attribute that I want to sort by)?

To do this, use the setOrder() method. For further information, see Section 9.5.2 *Ordering*.

2.10. What is the OID class used for?

The OID class encapsulates the details of the primary key of an object. You use instances of OID for retrieving an object from the database and you set the OID to a known value. You can also get an OID object from a DataObject.

2.11. I would like to retrieve all instances of a particular object type (e.g., com.arsdigita.kernel.User). How do I do this?

There is a method on com.arsdigita.persistence.Session (http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/persistence/Session.html) that retrieves a DataCollection for all instances of a data object type. The DataCollection can be filtered to narrow the results. It is possible to request individual properties of the collection without instantiating an object. You can also instantiate the data object, and if you use the DomainObjectFactory, you can get a DomainObject from the data object. It is recommended, however, that you do not instantiate the object unless absolutely necessary, because creating the object consumes extra memory resources as compared to iterating through the DomainCollection.

As a specific example, to get all users in the system, you can execute the following line:

```
DataCollection authors =
SessionManager.getSession().retrieve("com.arsdigita.kernel.User");
```

For more information, see Section 9.2.5 *Creating and Retrieving Objects in Java*.

Chapter 10.
Kernel Tutorial

This chapter discusses how to use permissioning and domain objects, answers frequently-asked questions about the WAF security mechanisms and API, and touches on extending the authentication system. Please refer to Chapter 4 *WAF Component: Kernel* for more information about the concepts used in these tutorials.

10.1. Permissions Tutorial

10.1.1. Granting Access

Granting access to a party is accomplished by creating a `PermissionDescriptor` and passing it to `PermissionService.grantPermission`. The following example grants `read` privilege on `MyACSObject 50` to `Group 5`:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.PermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;

OID acsObject = new OID("example.MyACSObject",
new BigDecimal(50));

OID party = new OID("com.arsdigita.kernel.Group", new BigDecimal(5));


PermissionDescriptor perm =
new PermissionDescriptor(PrivilegeDescriptor.READ,
acsObject, party);

PermissionService.grantPermission(perm);
```

The next example grants `admin` privilege on all objects to `User 100`:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.UniversalPermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;

OID party = new OID("com.arsdigita.kernel.User", new BigDecimal(100));


PermissionDescriptor perm =
new UniversalPermissionDescriptor(PrivilegeDescriptor.ADMIN,
party);

PermissionService.grantPermission(perm);
```

10.1.2. Revoking Access

Revoking a privilege on an object from a party is accomplished by creating a PermissionDescriptor and passing it to PermissionService.revokePermission. The following example revokes read privilege on MyACSObject 50 from Group 5:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.PermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;

OID acsObject = new OID("example.MyACSObject",
new BigDecimal(50));

OID party = new OID("com.arsdigita.kernel.Group", new BigDecimal(5));


PermissionDescriptor perm =
new PermissionDescriptor(PrivilegeDescriptor.READ,
acsObject, party);

PermissionService.revokePermission(perm);
```

The next example revokes admin privilege on all objects from User 100:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.UniversalPermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;

OID party = new OID("com.arsdigita.kernel.User", new BigDecimal(100));

PermissionDescriptor perm =
new UniversalPermissionDescriptor(PrivilegeDescriptor.ADMIN,
party);

PermissionService.revokePermission(perm);
```

10.1.3. Basic Access Check

The basic access check indicate whether a user has a privilege on an object. User X has privilege Y on object Z if either of the following is true:

- Privilege Y or admin has been granted universally to user X or some group to which X belongs.
- Privilege Y or admin has been granted on object Z or some object from which Z inherits permissions (via Z's context) to user X or some group to which X belongs.

To perform this check, you create a PermissionDescriptor and pass it to PermissionService.checkPermission. The following example checks read privilege on MyACSObject 50 for User 100:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.PermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;

OID acsObject = new OID("example.MyACSObject",
new BigDecimal(50));
```

```
OID party = new OID("com.arsdigita.kernel.User", new BigDecimal(100));

PermissionDescriptor perm =
new PermissionDescriptor(PrivilegeDescriptor.READ,
acsObject, party);

if (PermissionService.checkPermission(perm)) {
// user 100 has read access on object 50
} else {
// user 100 does NOT have read access on object 50.
// You might handle this case by displaying an
// "access forbidden" message.
}
```

As the previous examples have shown, operations performed on `PermissionDescriptor` can also be performed on `UniversalPermissionDescriptor` when dealing with universal access control (that is, access to all objects). The same applies for `PermissionService.checkPermission()`. The following example checks whether user `100` has `admin` privilege universally:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.UniversalPermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;

OID party = new OID("com.arsdigita.kernel.User", new BigDecimal(100));


PermissionDescriptor perm =
new PermissionDescriptor(PrivilegeDescriptor.ADMIN,
party);

if (PermissionService.checkPermission(perm)) {
// user 100 has admin access universally (that is, on all objects).
} else {
// user 100 does not universal admin access universally.
// You might handle this case by displaying an
// "access forbidden" message.
}
```

10.1.4. Allowed Targets Check

The allowed targets check is performed on a `DomainCollection` or `DataCollection` in order to filter the result set of domain/data objects to only those on which a given user has a given privilege. The rules for determining whether a given user has a given privilege on an object are described in Section 10.1.3 *Basic Access Check*.

You can be perform this filtering by using either `PermissionService.filterObjects(DomainCollection, PrivilegeDescriptor, Party)` or `PermissionService.filterObjects(DataCollection, PrivilegeDescriptor, Party)`. The following example retrieves all `MyACSObjects` on which `User 100` has `read` privilege:

```
import com.arsdigita.kernel.permissions.PermissionService;
import com.arsdigita.kernel.permissions.PermissionDescriptor;
import com.arsdigita.kernel.permissions.PrivilegeDescriptor;
import com.arsdigita.persistence.OID;
import com.arsdigita.persistence.DataCollection;
import com.arsdigita.persistence.SessionManager;
```

```
DataCollection objects = SessionManager.getSession()
.retrieve("example.MyACSObject");

OID party = new OID("com.arsdigita.kernel.User", new BigDecimal(100));


PermissionService.filterObjects(object, PrivilegeDescriptor.READ, party);

// iterate over filtered results
while (objects.next()) {
...
}
```

10.1.5. Setting the Access Control Context of an ACSObject

An `ACSObject` can inherit permissions from another `ACSObject`, which is called its `context`. To set the `context` of an object, use `PermissionService.setContext(object, context)`. The `context` may be set to `null` to signify that the object should not inherit permissions from any other object. Note that `universal permissions` still apply to any object, even if its context is `null`.

10.1.6. Defining New Privileges

There are 4 pre-defined privileges in the system, similar to file system privileges:

- `read` - The privilege to read or view an object.
- `write` - The privilege to write or edit an object.
- `create` - The privilege to create new objects within an object. For example, a user who has `create` privilege on a given **directory** is allowed to create new **files** within the directory.
- `admin` - Implies all other privileges.

Application developers may cautiously define new privileges to represent specific `actions` in the context of an application. For example, a content management system may support `publishing` a content item that is in final draft, but this operation should be restricted to certain parties for each content item. To accomplish this, the content management application creates a `publish` privilege from its initializer, as illustrated by the following code fragment.

```
if (PrivilegeDescriptor.get("publish") == null) {
PrivilegeDescriptor.createPrivilege("publish"); }
```

10.2. Domain Objects Tutorial

10.2.1. Building the Domain Object

This section will take you step by step through the process of building a domain object.

10.2.1.1. Starting the Class

A domain object starts off as a standard Java class that extends one of the four domain object base classes. Because of the auditing and permissioning requirements, `AuditedACSObject` is the base class.

The code in Example 10-1 illustrates the initial definition of the class and the import statements.

```
package com.arsdigita.notes;

import com.arsdigita.db.Sequences;
import com.arsdigita.domain.DataObjectNotFoundException;
import com.arsdigita.auditing.AuditedACSObject;
import com.arsdigita.kernel.Stylesheet;
import com.arsdigita.kernel.User;                              (1)
import com.arsdigita.persistence.DataAssociation;
import com.arsdigita.persistence.DataObject;
import com.arsdigita.persistence.PersistenceException;
import com.arsdigita.persistence.OID;
import com.arsdigita.persistence.metadata.ObjectType;

import java.math.BigDecimal;
import java.sql.SQLException;
import java.util.Date;
// Stylesheets
import java.util.Locale;
import java.util.ArrayList;

import org.apache.log4j.Category;

/**
 * Note class.  Extends AuditedACSObject to implement      (2)
 * persistent Note objects.
 *
 * <p>
 * Note: according to current DomainObject docs, APIs are not yet
 * stable. Changes may invalidate the Note class.
 *
 * @author Scott Seago
 **/
public class Note extends AuditedACSObject {                   (3)


    private static Category log =                              (2)
        Category.getInstance(Note.class.getName());

    /**
     * BASE_DATA_OBJECT_TYPE represents the full objectType name for the
     * Note class
     **/
    private static final String BASE_DATA_OBJECT_TYPE =        (4)
          "com.arsdigita.notes.Note";
...

    /**
     * Returns the appropriate object type for a Note so that proper
     * type validation will occur when retrieving Notes by  OID
     *
     * @return The fully qualified name of of the base data object
     * type for the Note object type.
     */
    protected String getBaseDataObjectType() {
        return BASE_DATA_OBJECT_TYPE;                      (4)
    }

}
```

(1) The `AuditedACSObject` is imported from the auditing service package.

(2) **Log4j** is used as a uniform debugging tool across WAF. The convention for initializing the **log4j** system is to initialize a category with the name of the current class, as shown here.

(3) By extending this baseclass, this Java class becomes a domain object. It inherits a data object and methods for manipulating it. Calls to these methods are automatically observed and modifications are logged in the audit trail.

(4) The data object type for the domain object is declared as a member variable, `BASE_DATA_OBJECT_TYPE`. It is final and static, so the compiler will replace it with a constant. An inherited function, `getBaseDataObjectType`, is overridden to return this variable. This method is used by the constructors on the superclasses to verify that the data object matches this type or is a subtype. For more information on this topic, see Section 10.2.1.2 *Constructors*.

Example 10-1. Starting to Build the Domain Object

10.2.1.2. Constructors

Before any of the methods of a class can be used, an instance must be constructed in memory. There are several different varieties of constructors for domain objects, all of which are illustrated in the `Notes` domain object.

There are two different types of constructors for domain objects. One is used for constructing domain objects by creating a new data object. The no-arg constructors — the `String typename` and the `ObjectType type` — are this type of constructor. These constructors use the persistence system to create a new data object that matches the type passed as an argument.

The other type of constructor relies on retrieval of an existing data object to construct the domain object. The first of these takes an Object ID or `OID` as input, and then retrieves a data object corresponding to that `OID`. An `OID` is comprised of two pieces: a `BigDecimal` numeric ID and an object type identifier. The second constructor takes a data object as input and constructs a `DomainObject` to wrap it. This constructor does not need to construct the underlying data object, so it is preferable to use this constructor if the data object has already been loaded into memory.

These constructors exist in `DomainObject`, `ObservableDomainObject`, `ACSObject`, and `AuditedACSObject`. Subclasses should replicate these constructors, unless there is a good reason to restrict them. The comments following Example 10-2 explain the overall utility of each constructor, and why each subclass should use them. Additional constructors can be added, and this is encouraged if it makes the class more convenient to use.

Note

> Be aware that none of the constructors will change the persistent state of the `DomainObject` or its data object. To store a domain object's properties in the database, the `save` method must be called. To delete a `DomainObject`, the `delete` method must be called.

```
public class Note extends AuditedACSObject {
    /**
     * BASE_DATA_OBJECT_TYPE represents the full objectType name for the
     * Note class
     **/
    private static final String BASE_DATA_OBJECT_TYPE =
       "com.arsdigita.notes.Note";

    /**
     * Default constructor. The contained DataObject is
     * initialized with a new DataObject with an
     * ObjectType of "Note".
     *
     * @see AuditedACSObject#AuditedACSObject(String)
     * @see com.arsdigita.persistence.DataObject
     * @see com.arsdigita.persistence.metadata.ObjectType
     **/
    public Note() {                                              (1)
        this(BASE_DATA_OBJECT_TYPE);
    }

    /**
     * Constructor. The contained DataObject is
     * initialized with a new DataObject with an
     * ObjectType specified by the string
     * typeName
     *
     * @param typeName The name of the ObjectType of the
     * contained DataObject.
     *
     * @see AuditedACSObject#AuditedACSObject(ObjectType)
     * @see com.arsdigita.persistence.DataObject
     * @see com.arsdigita.persistence.metadata.ObjectType
     **/
    public Note(String typeName) {                               (2)
        super(typeName);
    }

    /**
     * Constructor. The contained DataObject is
     * initialized with a new DataObject with an
     * ObjectType specified by type
     *
     * @param type The ObjectType of the contained
     * DataObject.
     *
     * @see AuditedACSObject#AuditedACSObject(ObjectType)
     * @see com.arsdigita.persistence.DataObject
     * @see com.arsdigita.persistence.metadata.ObjectType
     **/
    public Note(ObjectType type) {
        super(type);                                             (3)
    }


    /**
     * Constructor. Retrieves a Note instance, retrieving an existing
     * note from the database with OID oid. Throws an exception if an
     * object with OID oid does not exist or the object
     * is not of type Note
     *
     * @param oid The OID for the retrieved
     * DataObject.
     *
     * @see AuditedACSObject#AuditedACSObject(OID)
     * @see com.arsdigita.persistence.DataObject
     * @see com.arsdigita.persistence.OID
     *
     * @exception DataObjectNotFoundException Thrown if we cannot
     * retrieve a data object for the specified OID
     *
     **/                                                     (4)
    public Note(OID oid) throws DataObjectNotFoundException {
        super(oid);
    }
...
}
```

(1) The empty constructor is used to construct a new `Note` with its default data object type. This is the common case for constructing a new `Note`, where a specific subtype of domain object is not needed and an existing `Note` is not being retrieved.

(2) The `typeName` specified in this constructor must be a `String` that names an object type. This does the same thing as the constructor that takes an object type, but is more convenient, because you only need to specify a `String` with the type name.

(3) This constructor takes an `ObjectType` that matches the base object type, in this case, `com.arsdigita.notes.Note` or a subtype. The constructor requiring an `ObjectType` can be used to specify a specific subtype of the `Note`s data object. This is useful if the data object may be used with different domain objects that have different methods.

In most cases, the empty constructor, which defaults to the base object type, is sufficient, and any reason to use the other constructors should be specified. Even if the `Note` domain object will not be used with multiple object types, it is necessary to have these constructors for a subclass to provide this functionality. Without providing the constructors on every class, a subclass cannot rely on the constructor chain to reach the constructors on `ACSObject` and `DomainObject`.

(4) This constructor is used to construct a `Note`s domain object with a `Note`s domain object that has already been created. The `OID` is used to retrieve an instance of the correct domain object. The retrieved data object can then be accessed using the `set`/`get` methods and all other methods defined.

Example 10-2. Domain Object Constructors

The role of each constructor can be clarified by examining how the constructors are implemented in `DomainObject`, the root base class for all domain objects.

```
public abstract class DomainObject {
    private final DataObject m_dataObject;

    /**
     * Constructor. The contained DataObject is
     * initialized with a new DataObject with an
     * ObjectType specified by the string
     * typeName.
     *
     * @param typeName The name of the ObjectType of the
     * new instance.
     *
     * @see com.arsdigita.persistence.Session#create(String)
     * @see com.arsdigita.persistence.DataObject
     * @see com.arsdigita.persistence.metadata.ObjectType
     **/
    public DomainObject(String typeName) {
        Session s = SessionManager.getSession();           (1)
        if (s == null) {
            throw new RuntimeException("Could not retrieve a session
                from " + "the session manager while instantiating " +
                "a class with ObjectType = " + typeName);
        }

        m_dataObject = s.create(typeName);
        initialize();
    }

    /**
     * Constructor. The contained DataObject is
     * initialized with a new DataObject with an
     * ObjectType specified by type.
     *
     * @param type The ObjectType of the new instance.
     *
     * @see com.arsdigita.persistence.Session#create(ObjectType)
     * @see com.arsdigita.persistence.DataObject
     * @see com.arsdigita.persistence.metadata.ObjectType
     **/
    public DomainObject(ObjectType type) {
        Session s = SessionManager.getSession();           (1)
        if (s == null) {
            throw new RuntimeException("Could not retrieve a session
                from " + "the session manager while instantiating " +
                "a class with ObjectType = " + type.getName());
        }

        m_dataObject = s.create(type);
        initialize();
    }

    /**
     * Constructor. The contained DataObject is retrieved
     * from the persistent storage mechanism with an OID specified by
     * oid.
     *
     * @param oid The OID for the retrieved
     * DataObject.
     *
     * @exception DataObjectNotFoundException Thrown if we cannot
     * retrieve a data object for the specified OID
     *
     * @see com.arsdigita.persistence.Session#retrieve(OID)
     * @see com.arsdigita.persistence.DataObject
     * @see com.arsdigita.persistence.OID
     **/
    public DomainObject(OID oid) throws DataObjectNotFoundException {
        Session s = SessionManager.getSession();
        if (s == null) {
            throw new RuntimeException("Could not retrieve a session (2)
                from " + "the session manager while instantiating " +
                "a class with OID = " + oid.toString());
        }

        m_dataObject = s.retrieve(oid);
        if (m_dataObject == null) {
            throw new DataObjectNotFoundException
                ("Could not retrieve a DataObject with " +
                 "OID = " + oid.toString());
        }
        initialize();
    }

    /**
     * Constructor. Creates a new DomainObject instance to encapsulate
     * a given data object.
     *
     * @param dataObject The data object to encapsulate in the new domain
     * object.
     * @see com.arsdigita.persistence.Session#retrieve(String)
     **/
    public DomainObject(DataObject dataObject) {
        m_dataObject = dataObject;
        initialize();
    }                                                       (2)

    /**
     * Returns the base data object type for this domain object class.
     * Intended to be overridden by subclasses whenever the subclass will
     * only work if their primary data object is of a certain base type.
     *
     * @return The fully qualified name ("modelName.typeName") of the base
     * data object type for this domain object class,
     * or null if there is no restriction on the data object type for
     * the primary data object encapsulated by this class.
     **/
    protected String getBaseDataObjectType() {
        return null;
    }

    /**
     * Called from all of the DomainObject constructors
     * to initialize or validate the new domain object or its
     * encapsulated data object.  This was introduced in order to
     * support efficient validation of the encapsulated data object's
     * type.  If the validation is typically performed in class
     * constructors, then redundant validation is performed in
     * superclass constructors.  This validation now occurs here.
     **/
    protected void initialize() {
        if (m_dataObject == null) {
            throw new RuntimeException
                ("Cannot create a DomainObject with         (3)
                   a null data object");
        }

        String baseTypeName = getBaseDataObjectType();
        if (baseTypeName == null) {
            return;
        }
        // ensure data object is instance of baseTypeName
        // or a subtype thereof.
        ObjectType.verifySubtype(baseTypeName,
                                 m_dataObject.getObjectType());
    }
...
}
```

(1) These constructors are used to create an empty data object. The `Session` class is used look up the metadata that defines the object type and returns the data object. Because the data object is a private member variable, it is denoted `m_dataObject`.

(2) These two constructors are used to create a `DomainObject` when a data object already exists. The `OID`-based constructor searches for a data object with the given `OID`. If the object cannot be found, a DataObjectNotFound exception is thrown. Otherwise, the `DomainObject` is initialized with the given data object. The data-object-based constructor is used if the data object is available, so that an additional one does not need to be retrieved from the database.

(3) The `initialize()` is called after the domain object instance is created. One use of this function is to validate the data object type to see if it is the same type or a subtype of the base data object type. If not, an exception is thrown. In order for this to work, your subclass must implement `getBaseDataObjectType()`, as in the `Note`s example. Subclasses of DomainObject can add their own initialization logic, but should always call `super.initialize()` in the first line of the function so that initializations from the superclasses are processed.

Example 10-3. Implementing Constructors

10.2.1.3. Adding Create Methods

The constructors described in Section 10.2.1.2 *Constructors* are intended to be used for creating empty data objects or retrieving existing data objects. However, in some situations, it is useful to have a method that takes in a set of application-specific parameters and creates a domain object for you:

```
/**
 * Creates a new note and sets the title, body, and theme.
 *
 * @param title The title describing the note.
 * @param body  The body of the note.
 * @param theme The theme of the note.
 */
public static Note create(String title, String body, NoteTheme theme) {
    Note note = new Note();
    note.setTitle(title);
    note.setBody(body);
    note.setTheme(theme);
    return note;
}
```

This method is static, so it can be called without an instance of a `Note`. It constructs a new note and then sets the title, body, and theme properties to the values passed into the method. It then returns the constructed `Note` object. The `Note`'s data is not saved in the database by the constructor. Such data is not persisted until the `save` method is executed.

10.2.1.4. DomainObject Methods

Several methods are inherited from DomainObject that are useful for building other methods for subclasses. In general, methods are added to subclasses to provide application logic specific to a business domain. A common case of this are accessors and mutators for data object properties. Such methods are usually called by process objects or UI components.

10.2.1.4.1. Accessors and Mutators

Because DomainObjects wrap data objects, accessors and mutators for the domain object properties are common methods to provide on a domain object. The DomainObject class provides a set and get that are automatically delegated to the data object member of DomainObject. The set method is used to set properties of the domain object to a value. The get method is used to retrieve the values of those properties.

Using the Note domain object, the following example demonstrates how to implement an accessor and mutator for the contained data object's property.

```
/**
 * Retrieve the title of the note.
 *
 * @return The title.
 */
 public String getTitle() {
     return (String) get("title");
 }

/**
 * Set the title of the note.
 *
 * @param title A title for the note.
 */
 public void setTitle(String title) {
     set("title", title);
 }
```

The accessor, getTitle works by calling get with the name of the property to retrieve. The getmethod returns a java.lang.Object, so the return type must be casted to a String. The setTitle method works by calling the set with the name of the property to be set and the value.

The member data object in DomainObject is private to prevent access to the data object outside of the methods, ensuring an interface/implementation barrier. Subclasses of DomainObject cannot access the data object directly but must use methods on DomainObject instead. Because this barrier guarantees that all access to the properties of the data object are through set and get methods, subclasses can add behavior to these methods. For example, the ObservableDomainObject adds behavior to the set method that enables observers of the DomainObject to be notified when a property is changed.

10.2.1.4.2. Initialize

The DomainObject class provides an initialize method that is executed after the constructor is run. The Domain Subclasses can override this method and add further initialization logic. However, the initialize method on the superclass should be run to ensure that all inherited member variables are initialized and so that the data object type checking is executed.

A good use of the initialize method is to set initial values for data object properties. The ACSObject class uses the initialize method to determine the value for the ObjectType property.

```
/**
```

```
 * Called from base class constructors (DomainObject constructors).
 */
protected void initialize() {
    super.initialize();
    if (isNew()) {
        String typeName =
            getObjectType().getModel().getName() +
            "." + getObjectType().getName();

        set("objectType", typeName);
    }
}
```

10.2.2. Conventions

1. Every domain object with a public constructor should have a `public final static String BASE_DATA_OBJECT_TYPE` variable with the value equal to the primary data object type used by the domain object. The common use for the `BASE_DATA_OBJECT_TYPE` variable is to construct an `OID` for the data object given its numeric id. For convenience, you can provide a no-arg constructor that will dispatch to the object type constructor with the `BASE_DATA_OBJECT_TYPE` variable.
2. The protected method `getBaseDataObjectType` is required to provide proper data object type checking. In the common case, the method returns the `BASE_DATA_OBJECT_TYPE` variable. The `DomainObject initialize` method ensures that the data object used to create a `DomainObject` is equal to that type or a subtype of the object type returned by this method. For example, if you try to instantiate an `ACSObject` with a data object that is not a subtype of the `ACSObject` data object type, you will get an exception. By default, this method returns null, which disables the type checking

10.2.3. Examples

10.2.3.1. Creating Messages

By default, a new `Message` has a MIME type of `text/plain`, so creating and storing simple text messages is straightforward:

```
Party from; Message msg = new Message(from, "the subject", "the body");
msg.save();
```

To create a message in a different format, like `text/html`, the API provides a method to set the MIME type when the `Message` is created:

```
Message msg = new Message(from, subject);
msg.setBody("<p>this is the body</p>", MessageType.TEXT_HTML);
msg.save();
```

Finally, a `Message` can also be instantiated by retrieving it from the database by supplying its unique ID:

```
Message msg = new Message(new BigDecimal(1234));
```

Note that the Message class only support bodies with a primary MIME type of text. The MessageType interface defines legal MIME types for use in specifying a message body.

10.2.3.2. Referring to Other Objects

Messages can store an optional reference to an ACSObject. This is useful for attaching messages to other items in the system. For example, a discussion forum might implement a Topic class to represent a collection of related messages. The process of adding a message to a given topic can be implemented by setting a reference to that topic:

```
// One topic in a discussion forum
class Topic extends ACSObject {

// Add a new posting for this topic
addNewPosting (User postedBy, String subject, String post) {
Message msg = new Message(postedBy, subject, post);
msg.setRefersTo(this);
msg.save();
}
}
```

10.2.3.3. Attachments

Additional pieces of content can be attached to a message. These content items can be any media type with a supported DataHandler. Convenience methods are provided for attaching plain text documents directly to a message:

```
Message msg = new Message(from, subject);
msg.setText("See the attached document.");
msg.attach("A simple document", "simple.txt");
msg.save();
```

To attach more complex types of content, you use a MessagePart and an appropriate data source or DataHandler. There are convenience methods to attach content retrieved from a File or a URL. The following example shows how to attach an uploaded image to a message:

```
File file = new File("image.gif");

MessagePart part = new MessagePart();
part.setContent(image, "image.gif", "An image");

Message msg = new Message(from, subject, "See attached image");
msg.attach(part);
msg.save();
```

The code for downloading the image from a remote server is almost identical:

```
URL url = new URL("http://mysite.net/image.gif");

MessagePart part = new MessagePart();
part.setContent(url, "image.gif", "An image from mysite");

Message msg = new Message(from, subject, "See attached image");
msg.attach(part);
msg.save();
```

Note that the above two examples do not explicitly set a MIME type for the content. This is determined automatically when the content is handed off to a `DataHandler` for processing. In both cases shown above the content will be of type `image/gif`.

10.2.3.4. Simple Threading

A `Message` object can store a reference to another `Message` object to implement basic threading. A reply is created using the `reply()` method of the parent Message:

```
Message parent = new Message(new BigDecimal(1234)); Message reply =
parent.reply();

reply.setFrom(from);
reply.setText("I do not agree, because...");
reply.save();
```

When a reply is created this way, the subject is initialized as appropriate for a reply to the parent, and a reference to the parent message is stored internally. The client code must set other message properties, including the body of the reply.

10.2.3.5. Advanced Threading

Many applications need to store a more complex, structured set of responses than the flat organization provided by `Message`. The `ThreadedMessage` class provides support for organizing messages into a tree structure as outlined above. The API for `ThreadedMessage` is identical, but the `reply()` also takes care of setting a special sort key that determines the proper location of a message in the tree.

The following example shows a simple discussion thread and the technique used to retrieve all messages from the database for display in the correct order. The structure of the underlying tree is as follows:

```
msg0
msg2
    msg4
    msg5
msg3
    msg6
msg1
```

Although in a real application the structure of messages and replies would be generated over time, the following example shows the sequence of calls that produces the same organization, for illustration purposes.

```
// A common object that all messages refer to.
// In practice this might a forum or a content
// item that is being discussed by a group of
// collaborators.

ACSObject anchor;

// Create the message

ThreadedMessage msg[];

msg[0] = new ThreadedMessage();
msg[0].setRefersTo(anchor);

msg[1] = new ThreadedMessage();
```

```
msg[1].setRefersTo(anchor);

msg[2] = msg[0].replyTo(from,body);
msg[3] = msg[0].replyTo(from,body);
msg[4] = msg[2].replyTo(from,body);
msg[5] = msg[2].replyTo(from,body);
msg[6] = msg[3].replyTo(from,body);
```

Note that the two root-level messages (0 and 1) explicitly set the reference to our anchor object, but the replies do not. This information is automatically transferred to the replies when they are created, as are the subject and other properties, in the same way as `Message.reply()`.

In addition, `ThreadedMessage.replyTo()` also sets a special sort key that determines the correct position of the message in the tree. Details of the sort keys are not important, but they are generated so that all common messages can be retrieved in the correct order using a single order by clause. A special query provided by the messaging package is used to reconstruct the tree structure shown above:

```
Session session = SessionManager.getSession();
DataQuery query = session.retrieveQuery
 ("com.arsdigita.messaging.getMessageTree");
query.addEqualsFilter("object", anchor);

while (query.next() {
ThreadedMessage msg = new ThreadedMessage
 ((BigDecimal) query.get("id"));
System.out.println("message " + msg.toString());
}
```

The `addEqualsFilter` is required to retrieve only those messages that refer to the same `ACSObject`.

10.3. Security Service FAQ

This section addresses common questions developers have when implementing a security service through the kernel.

1. How do I determine whether the user is logged in?

`KernelHelper.getKernelRequestContext(request).getUserContext().isLoggedIn()` returns true if the user is logged in, false otherwise.

2. How do I get the ID for the current user?

`KernelHelper.getKernelRequestContext(request).getUserContext().getUserID()` returns the current user's ID. This method throws an exception if the user is not logged in, so check with `isLoggedIn()` first. If the calling code needs a `User` object for the current user, it should call `getUser()` instead.

3. How do I get the `User` object for the current user?

`KernelHelper.getKernelRequestContext(request).getUserContext().getUser()` returns a `User` object for the current user. This method throws an exception if the user is not logged in, so check with `isLoggedIn()` first. This method throws DataObjectNotFoundException if the user doesn't exist; calling code should catch this exception and handle it appropriately. The `User` object provided by this method is cached for the current thread, so multiple calls to `getUser()` incur at most one database hit.

`KernelHelper.getCurrentUser(request)` is a convenience method that returns the current `User` object if available, returns null if the user is not logged in, or throws a RuntimeException if the user is logged in but does not exist.

4. How can a user be logged in but not exist in the database?

WAF logs in users automatically using cookies (or other mechanisms), so it is possible for a client to present a valid cookie for a user that doesn't exist. This might happen if the user is deleted, the database is cleared, or an attacker spoofs a non-existent user's cookie. WAF uses cryptographic mechanisms to make the last case unlikely.

5. How do I require that a user is logged in for a given page?

`com.arsdigita.ui.login.UserAuthenticationListener` is a `RequestListener` that requires that the user is logged in and that the user exists. This listener should be added to any Bebop `Page` that requires login using `Page.addRequestListener()`.

If the user is not logged in or doesn't exist, the client is redirected to the login page with return_url set to the original request URL. This ensures that the client will return to the original page after logging in.

If the user is logged in, this listener provides the methods `isLoggedIn()` and `getUser()`. Note that these methods differ from those in `UserContext` since they also ensure that the user exists.

6. How do I require that a user is securely authenticated (using SSL or similar) for a given page?

This is not yet provided in a single API, but this can be accomplished using a `UserAuthenticationListener` and then calling `com.arsdigita.kernel.Initializer.getSecurityHelper.isSecure(request)` to check whether the current request is secure.

Note

In the current implementation of WAF, a user must be logged in securely to access a page over SSL. This restriction will be loosened in the future, and an API for checking secure authentication will be provided.

7. How do I get the `HttpSession` object for the current request?

WAF does not currently wrap session retrieval, so `request.getSession()` will return the current `HttpSession` object.

Note

Session management is independent of user authentication, so a session will always exist, whether or not the user is logged in.

8. How do I log in a user using a username and password?

`KernelHelper.getKernelRequestContext(request).getUserContext().login(username, password, forever)`, where `forever` should be true if the user should be granted a permanent login cookie, false if the cookie should only last for the current session. For an example, see `com.arsdigita.ui.login.UserRegistrationForm`.

Tip

> There are several subtle security issues in managing passwords safely, so developers are encouraged to use the provided UI code rather than creating their own. This ensures that developers automatically benefit from fixes in future releases.

9. How do I log in as another user (Administrators only)?

Three methods allow an administrator to assume the role of another user — the "\" is used to break the lines for printing purposes only::

1. `KernelHelper.getKernelRequestContext(request).getUserContext().\`
   `login(username)`
2. `KernelHelper.getKernelRequestContext(request).getUserContext().\`
   `login(userID)`
3. `KernelHelper.getKernelRequestContext(request).getUserContext().\`
   `login(User)`

The first two versions of this method are convenience wrappers for the third. These methods require that the current user be an administrator.

After calling any of these methods, the `UserContext` is updated with the target user's information; however, cached user information (such as that provided by `UserAuthenticationListener`) will still contain the administrator's information.

Immediately after calling any of these methods, the client should be redirected to a new page (such as the user workspace). See `com.arsdigita.ui.admin.UserLoginPage` for an example.

10. How do I log out a user?

`KernelHelper.getKernelRequestContext(request).getUserContext().logout()` logs out the current user. `com.arsdigita.ui.login.UserLogoutListener` is an `ActionListener` that logs out the user by calling this method.

Developers can link to this action listener using an `ActionLink`. Alternately, developers can link to a `com.arsdigita.ui.login.UserLogoutPage`; this page logs out the user and immediately redirects the client to the registration page.

11. How do I get a user's email address?

The user's `User` object given to the method `user.getPrimaryEmail().getEmailAddress()` returns the user's primary email address as a string. It is possible that `getPrimaryEmail()` or `getEmailAddress()` might return null.

`com.arsdigita.ui.login.EmailInitListener` is a `FormInitListener` that initializes a form parameter with the current user's email address, if possible. If the user is not logged in or has no email address, it leaves the parameter unchanged. See `com.arsdigita.ui.login.UserRegistrationForm` for an example.

12. How do I set a user's password?

Given the user's `UserAuthentication` object, `auth.setPassword(password)` sets the user's password.

Caution

> Use the method `auth.setPassword(password)` carefully, as it can allow an attacker to override the user's password.

`com.arsdigita.ui.login.PasswordValidationListener` is a `ParameterValidationListener` that checks whether a form parameter value is a strong password. This listener should always be used before setting the user's password. See `com.arsdigita.ui.login.ChangePasswordForm` for an example.

13. What is / How do I get a user's `UserAuthentication` object?

There are four methods for retrieving a user's `UserAuthentication` object:

1. `UserAuthentication.retrieveForLoginName(String loginName)`
2. `UserAuthentication.retrieveForUser(OID userOID)`
3. `UserAuthentication.retrieveForUser(BigDecimal userID)`
4. `UserAuthentication.retrieveForUser(User user)`

The first is appropriate if the username is entered in a form; the last two are appropriate if the user is already logged in. The `UserAuthentication` object allows code to check and set a user's password and password retrieval question and answer.

Caution

> Access to the `UserAuthentication` object is dangerous, since it can allow an attacker to compromise a user's account. If at all possible, use the provided UI components and login modules rather than access this object directly.

14. What is / How do I use the `SecurityLogger` class?

`SecurityLogger` wraps the **Log4j** category called `SECURITY`. It provides static methods for appending entries to that log, and automatically inserts the current date and time and, if applicable, the current client's IP address.

`SecurityLogger` is used to log security-relevant events for future auditing. Examples of such events are bad passwords, malformed cookies, and expired login page accesses. To use the class, simply call the appropriate class method, such as `SecurityLogger.warn()`.

10.4. Extending the Authentication System

The main strength of PAM is that the system may be extended with new authentication technologies. This is done by implementing new `LoginModules`. For example, an LDAP server can be used to authenticate a user by username and password. To integrate this behavior into WAF, a new login module (for example, `LDAPLoginModule`) must be defined.

`LDAPLoginModule` would replace `LocalLoginModule` in the login configuration. It needs its own configuration information to determine how to connect to the LDAP server. This information can be hard-coded into the module, read from a file, provided by an `Initializer`, or passed as an option in the login configuration.

If the LDAP username does not match the WAF username, another login module must be defined to map the LDAP username to a WAF user ID. This module, (for example, `LDAPUserLoginModule`), would replace `UserIDLoginModule` and must implement `MappingLoginModule.getUserID(username)` appropriately.

Changing the authentication recipe involves editing the sequence of login modules in the login configuration. For example, you can create a variety of authentication sequences in the login configuration by combining various login modules:

- Replace certain modules.
- Change the order of modules.
- Change the control flag on the modules.

Caution

> Not all of the combinations will make sense in terms of authentication practices, and may even be a security risk. As part of your exploration with authentication, you can read up on the specifics at http://java.sun.com/security/jaas/apidoc/javax/security/auth/login/Configuration.html.

Chapter 11. Services Tutorials

These tutorials build upon the discussion in Chapter 5 *WAF Component: Services*.

11.1. Categorization Tutorial

This section presents an overview of using the **categorization** layer within WAF. The focus in this section is on describing the main classes and interfaces that a programmer uses to work with the **categorization** API.

11.1.1. Classes

To use the **categorization** layer, you should become familiar with the following classes.

- `Category.java`
- `CategoryFilter.java`
- `CategorizedObject.java`

`com.arsdigita.categorization.Category`

http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/Category.html

A `Category` encapsulates a persistent category object. This object contains the name and description for the category, as well as information about whether the category is enabled. It also provides an API for placing objects within this category, creating subcategories under the instance category, and retrieving parent and child categories and objects.

This class has several notable methods that you should understand before beginning work with **categorization**. The URLs have been line broken with a "\" for printing purposes:

- `addChild(ACSObject acsObject)` is used to create subcategories under the instance `Category` and to categorize objects within the instance `Category`.

  http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/Category.html\
  #addChild(com.arsdigita.kernel.ACSObject)

- `addRelatedCategory(Category category)` is used to specify a related category, that is, a category that is not necessarily a parent or a child, but is related in nature.

  http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/Category.html\
  #addRelatedCategory(com.arsdigita.services.categorization.Category)

- `removeChild(ACSObject object)` is used to delete a mapping with a child.

  http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/Category.html\
  #removeChild(com.arsdigita.kernel.ACSObject)

- `getChildCategories()` returns all subcategories for the given category.

  http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/Category.html\
  #getChildCategories()

- `getChildObjects()` returns all objects categorized within the instance `Category`.

  http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/Category.html\
  #getChildObjects()"

- `setFilter(CategoryFilter filter)` can be used to filter the children or parents returned by calls to a given category. By default, the `Filter` is set to return only enabled categories.

  http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/Category.html\
  #setFilter(com.arsdigita.categorization.CategoryFilter)

`com.arsdigita.categorization.CategorizedObject`

http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/CategorizedObject.html

This class is the *leaf* node of a category tree, containing the actual data object and methods to find the parent. That is, when a subcategory is asked for its objects, it returns a collection of `CategorizedObjects`. Methods within `CategorizedObject` can then be used to retrieve information about the parents of the given Object as well as the actual ACSObject.

`com.arsdigita.categorization.CategoryFilter`

http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/CategoryFilter.html

This class allows you to filter the categories that will be returned as parents or children of categories. Currently, it allows you to choose enabled categories only, disabled categories only, or all categories. By subclassing this class, applications can add further filtering criteria.

Other Classes

These classes work together to provide access to a `Bebop Tree` that can be used to display the category tree:

- `com.arsdigita.categorization.CategoryTreeModel` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/CategoryTreeModel.html
- `com.arsdigita.categorization.CategoryTreeNode` — http://rhea.redhat.com/doc/waf/6.0/api/com/arsdigita/categorization/CategoryTreeNode.html

These classes are not of much importance to you unless you want to change the way the tree appears. Displaying a category hierarchy is a multiple-step process. First, create a `TreeModelBuilder` that will instantiate a `CategoryTreeModel` (based either on a query or some other external data source). This `TreeModelBuilder` is passed to the `Tree`, which uses your builder to create its `TreeModel`. This process is necessary since a `DataObject` cannot be used across connections.

11.2. Categorization Scenarios

11.2.1. Creating a new Category

```
// Creates the Movies category at the top level with a description
Category movies = new Category("Movies", "long television shows");
movies.save();
// Create the Romantic Comedies category under the Movies category.
Category romance = new Category("Romantic Comedies",
                                "Comedies with love stories");
romance.setDefaultParentCategory(movies);
romance.save();
// Create the Titanic category under the Romantic Comedies category.
Category titanic = new Category("Titanic", "A category for large movies");
// set this as the default category
```

```
titanic.setDefaultParentCategory(romance);
titanic.save();
// Create the Drama category under the Movies category.
Category drama = new Category("Drama", "long, not funny stories");
drama.save();
// Make Drama a parent category (but not the default parent) of Titanic.
drama.addChild(titanic);
drama.save();
// Finally, make movies a parent of Drama
drama.setDefaultParentCategory(movies);
drama.save();
```

Example 11-1. Creating a new category

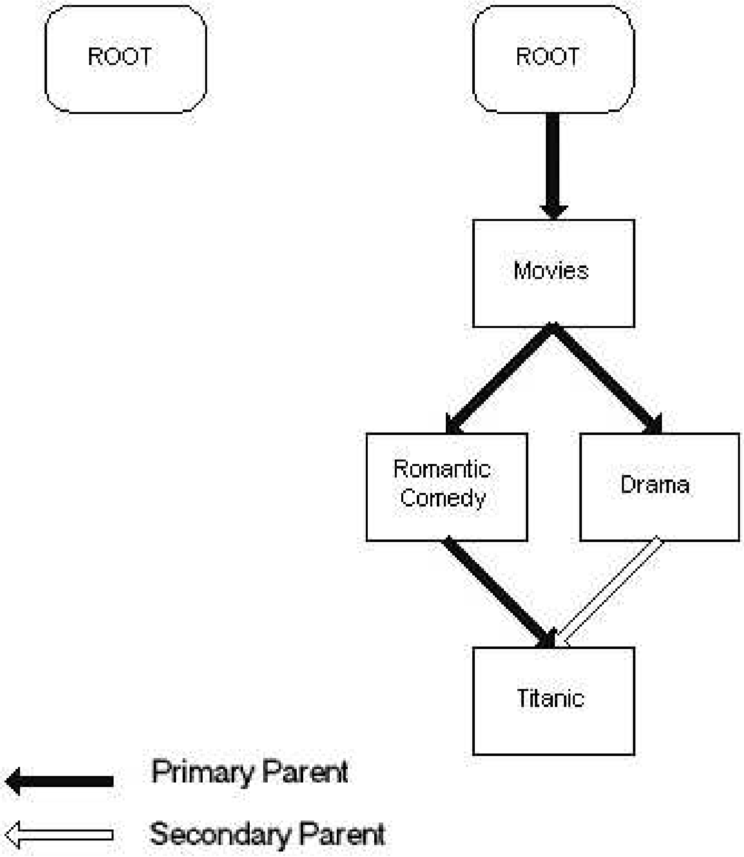


Figure 11-1. Category hierarchy before and after code execution

In Example 11-1, notice that whenever a new item is instantiated, it must be explicitly saved. This allows developers to easily back out of changes in the middle of the transaction.

Additionally, adding the "Titanic" category to two categories looks the same as adding it to a single category. The first category to which an item (category or other `ACSObject`) is added is explicitly added as the default category. If a category/item only has one parent, that parent is implicitly the default.

11.2.2. Deleting a Category

```
// Fetch the Movies category from the database.
Category category = new Category(new OID(123));
// Delete the Movies category.
category.deleteCategoryAndRemap();
```

Example 11-2. Category deletion

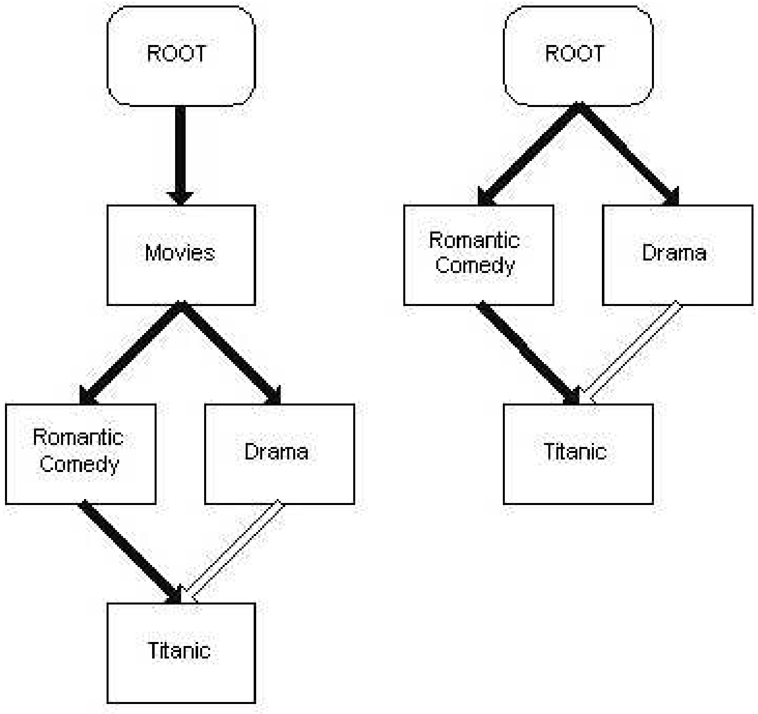


Figure 11-2. Category hierarchy before and after deleting a category

You can delete a category in the following ways:

- Deleting Leaf Category — If the category is a leaf, call `Category.delete()`.
- Remapping Child Categories — If you want to remove the current category and make all categories that were pointing to it point to its default parent (as in the example), call `Category.deleteCategoryAndRemap()`.
- Orphaning Children — If you want to delete the category without remapping any of the child categories, call `Category.deleteCategoryAndOrphan()`.
- Deleting Category Subtree — If you want to delete the entire subtree, including the passed-in category and all categories below it whose default ancestry can be traced to the root, call `Category.deleteCategorySubtree()`.

11.2.3. Adding more Parent Categories

```
Category a = new Category(new OID(123));
Category b = new Category(new OID(124));
Category c = new Category(new OID(125));
if ( a != null && b != null && c != null ) {
// Makes Category B the parent of Category A
b.addChild(a);
b.save();
// Makes Category C the parent of Category A
c.addChild(a);
c.save();
// Sets Category B as the default parent of Category A
a.setDefaultParentCategory(b);
a.save();
}
```

Example 11-3. Adding more parent categories

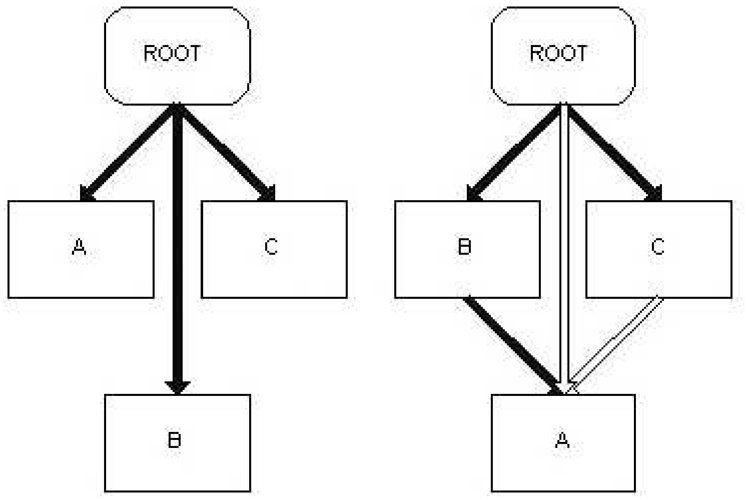


Figure 11-3. Category hierarchy before and after adding parent categories.

11.2.4. Removing Parent Categories

```
// Fetch category A from the database.
Category a = new Category(new OID(123));
// Fetch the parent category
Category b = a.getDefaultParentCategory();
b.removeChild(a);
b.save();
a.setDefaultParentCategory(b.getDefaultParentCategory());
```

Example 11-4. Removing Parent Categories

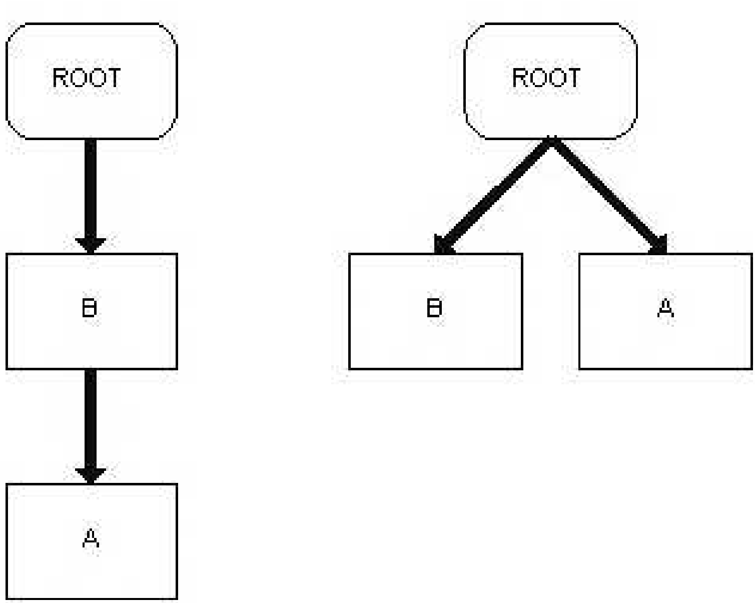


Figure 11-4. Category Hierarchy before and after removing parent categories

11.2.5. Retrieving all Subcategories of a given Category

```
// assume that we have the Category OID (call it categoryOID)
Category category = new Category(categoryOID);
// the Collection will be a Collection of Category objects
Collection subcategories = category.getChildCategories();
```

Example 11-5. Retrieving Subcategories of a given Category

11.2.6. Retrieving all child objects of a given Category

```
// assume that we have the Category OID
// (call it categoryOID)
Category category = new Category(categoryOID);
// the Collection will be a Collection of
// CategorizedObjects
Collection childObjects = category.getChildObject();

// print out the toString for each object
// and its parents
Iterator childIterator = childObjects.iterator();
while (childIterator.hasNext()) {
CategorizedObject object =
 (CategorizedObject) childIterator.next();
System.out.println("object = " + object.getObject());
Iterator parents = object.getParentCategories().iterator();
while(parents.hasNext()) {
System.out.println("parent = " +
 ((Category) parents.next()).toString());
}
}
```

Example 11-6. Retrieving child objects of a given Category

11.3. Notification Tutorial

This section deals with several types of **notification** services you can create in WAF — simple notification, notification with attachment, notification digest, and email alerts.

11.3.1. Creating a Simple Notification

A basic notification requires a *sender*, a *receiver*, a *subject*, and a *body*. All notifications require a `Message` object to send, but in the case of a simple text message, an API allows you to provide the relevant information and constructs a temporary `Message` object for you. By default, these notifications will be deleted after they are sent.

The following code fragment depicts the simplest case of sending a notification to a user:

```
User to   = getRecipient();
User from = getSender();

Notification n = new Notification(to, from, "Subject", "Body");
n.save();
```

This notification will be sent to the user on the next run of `SimpleQueueManager.to` and `from` must be persistent `Parties` in the database.

You can also create a notification and send it to a `Group`. When the notification is processed, it will be sent to each individual `User` in the `Group`.

```
Group group = getTargetGroup();
User  from  = getSender();

Notification n = new Notification(group, from, "Subject", "Body");
n.setExpandGroup(Boolean.TRUE);
n.save();
```

Note that this option is turned on by default, so the call to `setExpandGroup` is not strictly necessary.

11.3.2. Creating a Simple Notification with an Attachment

For more complicated notifications, it is better to construct a separate `Message` object and then wrap a **notification** around it for email delivery to a user. The following example shows how to construct a multipart message with attachments, using both an HTML document fetched from a URL and a simple text document constructed on the fly.

```
// Create a message for the body of the notification
User from = getSender();
Message msg = new Message(from, "subject", "See attachments");
```

```
// Attach some content fetched from a URL
URL url = new URL("http://www.yahoo.com/");
MessagePart part = new MessagePart("Yahoo", "Yahoo index page");
part.setDataHandler(new DataHandler(url));
msg.attach(part);

// Attach a simple text document
msg.attach("Some additional text", "file.txt", "description);

// Save the message
msg.save();

// Send the message to the recipient
User to = getRecipient();
Notification n = new Notification(to, msg);
n.save();
```

11.3.3. Creating an Hourly Digest

Applications that send frequent notifications to users are often designed to group those messages into a digest. A common example is a workflow application that sends users a summary of new and completed tasks every hour.

The **notification** service supports using an explicit digest. After creating a digest, you can use it to send future notifications by passing the digest to the **notification** constructor. All messages sent to a user through the same digest will be combined into a single outbound message when the digest is next processed.

```
// Party responsible for sending the digest
User from = getDigestSender();

String subject = "my-digest";
String header  = "Recent messages";
String footer  = "To unsubscribe, visit
 http://mysite.net/unsubscribe";

Digest digest = new Digest(from, subject,
 header, footer);
digest.setFrequency(Digest.HOURLY);
digest.save();

// Create a couple of notifications and send
// them using the hourly digest

Notification n0 = new Notification(digest, user,
 from, "Subject0", "Body0");
n0.save();

Notification n1 = new Notification(digest, user,
 from, "Subject1", "Body1");
n1.save();
```

These two separate notifications will be composed into a single email message and sent to the user during the next run of the digest queue manager.

Applications typically create their digests when they are installed, and then use them as part of processing alerts for users.

11.3.4. Email Alerts

The examples above assume that applications will simply need to notify users of some event via email. However, one of the powerful features of the **notification** service is that it is built on top of the WAF **messaging** service. One of the goals of this design is to make it efficient for messaging applications to send `Message` objects directly to users via email, without the need to store any additional information in the database.

A simple example of this is a bulletin-board (*bboard*) application that stores each post as a `Message`, and then sends each post via email to Users who have subscribed to bboard alerts. In addition to the basic text of the message, such an application might also need to wrap additional text around it to include introductory remarks and possibly action links that users can manipulate.

For example, a simple message such as `this is my post` would typically be sent out with additional information:

```
Message-ID: <574077721.1185532184.CCMMail@localhost>
Date: Fri, 5 Oct 2001 00:43:21 -0700 (PDT)
From: bboard-robot@somedomain.net
To: user@somedomain.net
Subject: a simple post
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Posted by: Bboard User
Category: Announcement

this is my post

----------
This is a posting from the company news bboard.
To reply you can go to:
http://mysite.com/bboard/news/
To unsubscribe from this or other bboard posts go to:
http://mysite.com/bboard/subscriptions/
```

The **notification** API includes methods to define the additional text that is wrapped around the contained `Message` object so that these emails can be generated without creating and storing a duplicate `Message` object in the database. In the example above, the header and signature of the notification would be specified as follows:

```
Notification notice = new Notification(to,msg);
notice.setHeader
    ("Posted by: " + poster.getName() + "\n" +
     "Category: " + forum.getCategory());
notice.setSignature
    ("---------\n" +
     "This is a posting from the company news bboard.\n" +
     "To reply you can go to:\n" +
     "http://mysite.com/bboard/news/\n" +
     "To unsubscribe from this or other bboard posts go to:\n" +
     "http://mysite.com/bboard/subscriptions/");
notice.save();
```

Applications that use extensive email alerts will probably want to extend `Notification` so that the header and signature are set in a special constructor for each notification to be sent out.

For efficiency, applications should avoid sending email alerts to individual users. It is much more efficient to store the list of recipients in a `Group` and send a single notification to the group, letting the **notification** service expand the group into the list of individual recipients. If you follow this approach, an email alert going out to 1000 users will occupy the same amount of space in the database as a single email.

11.4. Workflow Tutorial

This tutorial covers various tasks that can be accomplished with the **workflow** service — a simple workflow, adding a task, rolling back a task, and completing a task.

11.4.1. Simple Workflow

These steps describe the procedure in Procedure 11.1, *Creating a Simple Workflow*. The procedure covers creating a *hello world* workflow and describing the task state change.

1. Design a workflow by defining the desired tasks and the dependencies between them.
2. Create a new named task class for the different task types. Be sure to define the `BASE_DATA_OBJECT_TYPE`, the `getBaseDataObjectType`, constructors, and clone methods. Also include the methods to invoke.
3. Write the PDL file to support the new task type.
4. Create an instantiator for `DomainObjectFactory`.
5. Create a workflow template based on the design.
6. Instantiate a workflow from a workflow template.
7. Call the workflow start method to start the Engine.

Creating a Simple Workflow

1. Design your workflow.
2. Create a new task type by subclassing `task`:

```
public class HelloWorldTask extends Task {

    //The base data object is used for the domain object
    public static final String BASE_DATA_OBJECT_TYPE =
        "com.arsdigita.workflow.simple.HelloWorldTask";

    protected String getBaseDataObjectType() {
        return BASE_DATA_OBJECT_TYPE;
    }

    //constructors are not inherited from subclasses
    //so you'll need to add them
    public HelloWorldTask(String label, String description) {
        this(BASE_DATA_OBJECT_TYPE);
        initAttributes(label,description);
    }

    public HelloWorldTask(DataObject taskDataObject) {
        super(taskDataObject);
    }

    public HelloWorldTask() {
      this(BASE_DATA_OBJECT_TYPE);
      setState(DISABLED);
    }
    public HelloWorldTask(OID oid) throws
        DataObjectNotFoundException {
        super(oid);
    }

    public HelloWorldTask(BigDecimal id) throws
        DataObjectNotFoundException {
        this(new OID(BASE_DATA_OBJECT_TYPE, id));
    }

    protected HelloWorldTask(ObjectType type) {
        super(type);
    }

    protected HelloWorldTask(String typeName) {
        super(typeName);
    }


    // Add functionality in these methods
    public void enableEvt() {
        System.out.println(getLabel()+": I am enabled");
    }

    public void disableEvt() {
        System.out.println(getLabel()+": I have been disabled");
    }

    public void finishEvt() {
        System.out.println(getLabel()+": I am finished");
    }

    public void rollbackEvt() {
        System.out.println(getLabel()+":  I was rollbacked");
    }

    //Write clone functionality. Note the use of copyAttributes
    //
    public Object clone() {
        HelloWorldTask taskClone = new HelloWorldTask();
        copyAttributes(taskClone);
        return taskClone;
    }

    //Add this method because future versions will
    protected void copyAttributes(HelloWorldTask taskClone) {
        taskClone.setLabel(getLabel());
        taskClone.setDescription(getDescription());
        taskClone.setActive(isActive());
    }
}
```

Ensure that the constructors, events, and clone methods are coded for the new task type.

3. Write the PDL for the new subclass of task. This is a simple case; you would probably have additional tables for select and DML statements.

```
object type HelloWorldTask extends Task {

    retrieve {
        super;
    }

    insert  {
        super;
    }

    update {
        super;
    }

    delete {
       super;
    }

}
```

4. In your package initializer section, create and add a new instantiator.

```
DomainObjectInstantiator instHelloWorldTask =
     new ACSObjectInstantiator() {
     public DomainObject doNewInstance(DataObject
     dataObject) {
         return new HelloWorldTask(dataObject);
     }
};

DomainObjectFactory.registerInstantiator(
     "com.arsdigita.workflow.simple.",
     instHelloWorldTask);
```

5. Create the workflow template.

```
public class WFTutorialHello {

    public void WFTutorialHello() {
        HelloWorldTask helloTask1 =
         new HelloWorldTask("hello 1", "description");
        HelloWorldTask helloTask2 =
         new HelloWorldTask("hello 2", "description");

        //Note(1): Save the task before adding the dependency
        helloTask1.save();
        helloTask2.save();
        helloTask2.addDependency(helloTask1);
        helloTask2.save();

        User user = new User(__USER_ID__)
        Workflow wf = new Workflow("hello example",
         "tutorial example of creating a task");

        wf.addTask(helloTask1);
        wf.addTask(helloTask2);

        //Note(2): The workflow mad
        helloTask1.save();
        helloTask2.save();
        wf.start();

        helloTask1.finish();
        helloTask2.finish();
    }
}
```

There are a couple of subtle issues to note with regard to the above example. (These issues are commented with `NOTE(n)` in the code.) First, it is important to save the task before adding dependencies. Otherwise, the internal persistence will fail. Additionally, ensure that a save is called on each task after adding to workflow, since the workflow may call a couple of methods on the added task.

11.4.2. Adding a Task to a Workflow

To add a task, you call the `addTask` method in workflow. The task is inactive when created, and can be set to be active by calling the `setActive(true)`. A task is not considered part of the workflow until it becomes active. Tasks are created in an inactive state to allow you to preview them before altering the workflow state.

Once a task is active, it can affect the workflow and other dependent tasks. Manually setting the task state is required only when adding a new task to an in-progress workflow. When the workflow is started, it will automatically set all tasks as active.

11.4.3. Rolling Back a Task

Rolling back a process to a certain task is done by enabling a task early on in the workflow. For example, to move the process to `task A`:

```
//Get reference to task A
taskA.enable();
```

This will set the workflow back to the state that preceded finishing `task A`.

11.4.4. Completing a Task

When a task is enabled, call the `finish` method to complete it:

```
//Get reference to task A and call finish method
taskA.finish();
```

11.5. Versioning Tutorial

For a high-level overview and the basic definition of versioning, please refer to Section 5.11 *Versioning Service*.

Conceptually, versioning can be divided into two parts:

1. One part concerns PDL syntax that allows the developer to declaratively request certain object types to be versioned.
2. The other part consists of the Java API provided by the `com.arsdigita.versioning` package. This API provides classes and methods for interacting with and making use of the versioning service.

This tutorial explains the PDL syntax first, with examples covered in Section 11.5.5 *PDL Syntax*. If you wish to skip ahead, you can go directly to Section 11.5.4 *Versioning service API*.

The implementation and semantics of the versioning service have changed in a number of major ways after the 5.2 release. If you are familiar with the older implementation and would like a brief summary of the major differences between the old and current implementations, please refer to Section 11.5.6 *Differences between WAF 5.2 and 6.0*.

11.5.1. Data-Object-Level Service

The versioning service operates on the data object level. Putting the `versioned` keyword in front of the `object type` definition makes all instances of this type versioned. For example:

```
versioned object type Quux {
    BigInteger[1..1] id = quuces.id INTEGER;
    String[1..1] name   = quuces.name VARCHAR;
    object key(id);
}
```

Example 11-7. The `versioned` keyword

The consequences of marking an `object type` as `versioned` are explained in Section 11.5.2 *Fully versioned types*.

11.5.2. Fully versioned types

All instances of the `object type` `Quux` defined in Example 11-7 are versioned.

Marking one `object type` as versioned may impose constraints on other `object types`, forcing them to be versioned, too. Two most straightforward examples of this are *subtyping* and *composite-component relationship*.

```
object type GreatQuux extends Quux {
    String[0..1] email  = great_quuces.email VARCHAR;

    component Foobar[0..n] foobars = join great_quuces.id
                                       to foobars.great_quux_id;

    Country[1..1] country = join great_quuces.country_id
                              to countries.id;

    reference key (great_quuces.id);
}
```

Example 11-8. Components and required compound attributes

In Example 11-8, the `GreatQuux` `object type` is versioned, because it extends a versioned type. If the `object type` *X* is marked as versioned, then the versioning service infers that the subtypes and components of *X* are also versioned.

This happens because a data object fully owns its components. As part of properly versioning a data object, we must version its components. In Example 11-8, this semantic constraint dictates that the `object type` `Foobar` is also versioned.

Generally speaking, versioning a data object means recording its creation and deletion events, as well as all changes to its attributes. For example, if a Great Quux's `email` was *quux@example.com* on 19 Jan. and changed to *great_quux@example.com* on 12 Apr., the versioning service will have a record of this.

In the event that this data object is rolled back to its 19 Jan. state, then the email address has to be changed back to *quux@example.com*. This is an example of versioning a *scalar attribute*.

We use the term scalar attribute to refer to a property that has a *simple object type*, such as `BigInteger`, `Boolean`, or `String`.

If the properties type of an `object type` are compound, they are referred to as *compound attributes*. In Example 11-8, `Country` is a compound `object type` (the exact PDL definition is not shown for the sake of brevity). Thus, the property `country` of the `object type` `GreatQuux` is a compound attribute.

As the example of the `email` property demonstrates, the case of versioning scalar attributes is a straightforward one, while compound attributes require a little extra care. Section 11.5.3 *Recoverable types* explains another situation in which compound attributes require special treatment.

11.5.3. Recoverable types

The case of compound attributes like `country` is a little more complex. Let's look at Example 11-8 again. Suppose the Great Quux had Wonderland listed as his country of residence on Jan 19. On Apr 12, his country of residence changed to Eriador; the Wonderland data object was deleted altogether. What should happen if we try to rollback the Great Quux to his Jan 19 state? Since the data object is versioned, its `country` attribute should be rolled back to Wonderland. But Wonderland no longer exits in the database. One solution is to set the `country` to null. This would not work in this case, because this attribute is required - it has the multiplicity of `1..1`. Therefore, we should be able to recover the deleted data object.

This leads to the notion of a *recoverable* data object. A data object is recoverable if the versioning service is able to undelete it after it has been deleted. Being recoverable is a weaker requirement than being fully versioned. The versioning service may not be able to rollback a recoverable object back to an arbitrary point in its history, but it should record enough information to undelete it, in case it is ever deleted.

The versioning service infers that if a versioned `object type` has a required compound attribute, then the `object type` of the attribute must be marked as recoverable. The versioning service does not normally record changes to recoverable objects, but it keeps track of them in case they are deleted. If a recoverable object is deleted, the versioning service records the last known values of its attributes in order to be able to undelete it.

What does this mean in terms of Example 11-8? The versioning service treats the `Country` object type as `recoverable`, because it is a required compound attribute of the versioned type `GreatQuux`.

Why not make the `Country` `object type` fully versioned? Because we don't want to version more data than is actually needed. If a developer wants to make the `Country` `object type` fully versioned, they should explicitly mark it as such in the PDL definition.

What if the `Country object type` ***is*** marked as versioned? Should we rollback the `country` attribute when we rollback the Great Quux object? We cannot. Since it is not a component of `GreatQuux`, `Country` data objects may be included as an attribute by other `object types`. If we rollback a country as part of rolling back the Great Quux object, we may affect a lot of other objects that should not have been necessarily affected by this rollback.

11.5.4. Versioning service API

For backwards compatibility, some of the old versioning APIs are preserved. For example, you can tag the state of a data object in the current transaction by calling the `applyTag` method on the `VersionedACSObject` class. However, this class has been deprecated.

The preferred point of entry into the API provided by the versioning service is the `Versions` class. One important capability it has is providing methods for invoking the following functionalities:

1. Tagging the state of a data object in the current transaction. Note that you can tag the state of a data object, even if no changes were made to it in the current transaction. You can also tag more than one data object within the same transaction.
2. You can retrieve a collection of tagged transactions for a data object. Note that since the data object may have been deleted, the method for retrieving the collection takes in an *`OID`* parameter rather than a `DataObject`.
3. You can roll a data object back to any previous point in its history.
4. You can compute the difference between any two states of a data object.
5. You can suspend versioning till the end of the current transaction.

Please refer to the Javadoc API at http://rhea.redhat.com/doc/waf/6.0/ for details.

11.5.5. PDL Syntax

To specify which `object type`s should be versioned, developers can use the keywords `versioned` and `unversioned`. Consider the following example.

```
versioned object type VT1 {
    BigInteger[1..1] id = t_vt1.id INTEGER;
    object key(id);
}

object type VT2 extends VT1 {
    component UT3[0..n] ut3s = join t_vt2.id to t_ut3.vt2_id;
    reference key(t_vt2.id);
}

object type C1 {
    BigInteger[1..1] id = t_c1.id INTEGER;
    A1[1..1] a1s = join t_c1.a1_id to t_a1.id;
    component UT4[0..1] ut4 = join t_c1.ut4_id to t_ut4.id;
    object key(id);
}

association {
    VT2[0..1] vt2 = join t_c1.vt2_id to t_vt2.id;
    component C1[0..n] c1s = join t_vt2.id to t_c1.vt2_id;
}

object type A1 {
    BigInteger [1..1] id = t_a1.id INTEGER;
    UT4[1..1] ut4 = join t_a1.ut4_id to t_ut4.id;
    object key(id);
}

object type UT1 {
    BigInteger[1..1] id = t_ut1.id INTEGER;
    UT3[1..1] ut3attr = join t_ut1.ut3_id to t_ut3.id;
    object key(id);
}

object type UT2 {
    BigInteger[1..1] id = t_ut2.id INTEGER;
    object key(id);
}

versioned object type VUT2 extends UT2 {
  UT1[1..1] ut1 = join t_vut2.ut1id to t_ut1.id;
  unversioned String[1..1] unverAttr = t_vut2.unver_attr VARCHAR;
  unversioned component UT5[0..n] ut5 = join t_vut2.ut5id to t_ut5.id;
  versioned UT6[0..n] ut6s = join t_vut2.ut6id to t_ut6.id;
  reference key(t_vut2.id);
}

object type UT3 {
    BigInteger[1..1] id = t_ut3.id INTEGER;
    composite VT2[1..1] vt2 = join t_ut3.vt2_id to t_vt2.id;
    object key(id);
}

object type UT4 {
    BigInteger[1..1] id = t_ut4.id INTEGER;
    object key(id);
}


versioned object type VTC3 {
    BigInteger[1..1] id = t_vtc3.id INTEGER;
    object key(id);
}

object type C2 {
    BigInteger[1..1] id = t_c2.id INTEGER;
    composite VTC3[1..1] vtc3 = join t_c2.composite_id to t_vtc3.id;
    object key(id);
}

object type UT5 {
    BigInteger[1..1] id = t_ut5.id INTEGER;
    object key(id);
}

object type UT6 {
    BigInteger[1..1] id = t_ut6.id INTEGER;
    object key(id);
}
```

Example 11-9. Sample PDL definitions

For the sake of brevity, Example 11-9 only has key attributes for `object types` and does not show any scalar attributes.

The graph in Figure 11-5 provides a visual representation of the above PDL definitions. Wheat-colored nodes represent `object types` that are marked as `versioned` in Example 11-9. There are two types of edges. If an edge is labeled `extends`, it means that the type at the edge head extends, in the PDL sense, the type at the edge tail. For example, `VT2` extends `VT1`. This relationship is important, because subtypes of a versioned type are also versioned.

The other kind of edge is one that shows attributes of an object type. For example, consider the edge labeled `rqd:ut3attr` that connects `UT1` and `UT3`. It means that the `object type` `UT1` has a required attribute `ut3attr` of type `UT3`.

There are two kinds of compound attributes that are important from the versioning point of view: required (`rqd`) and components (`cnt`).

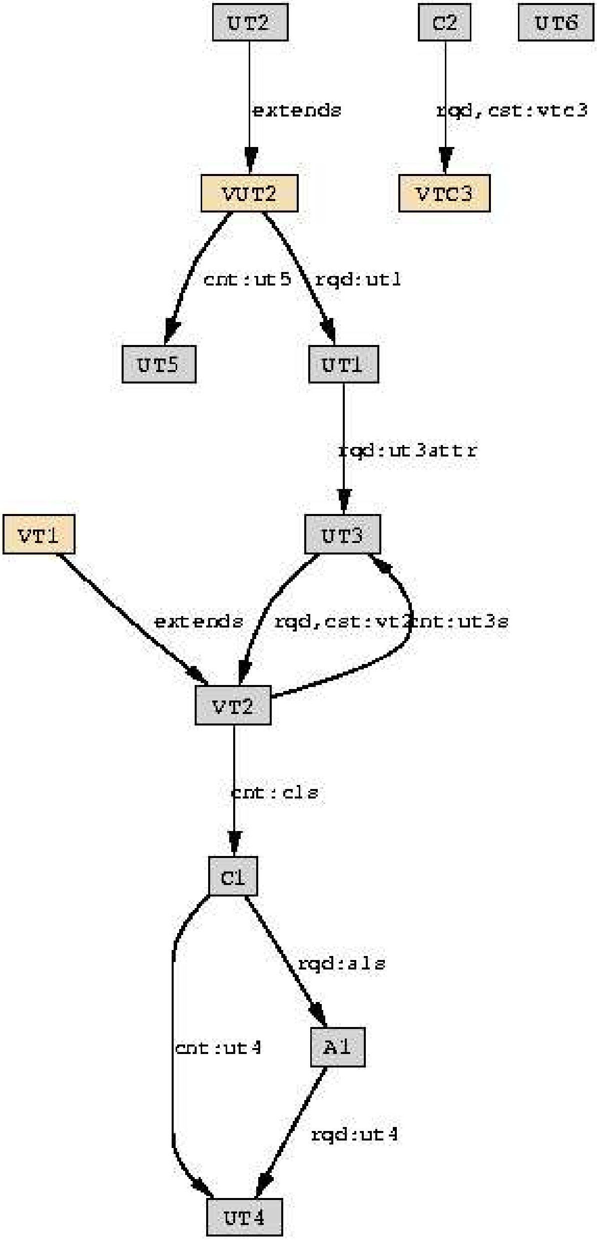


Figure 11-5. PDL definition graph

Based on Figure 11-5, the versioning service decides which `object types` should be versioned, recoverable, or simply ignored. The result of this decision can be visualized as follows.

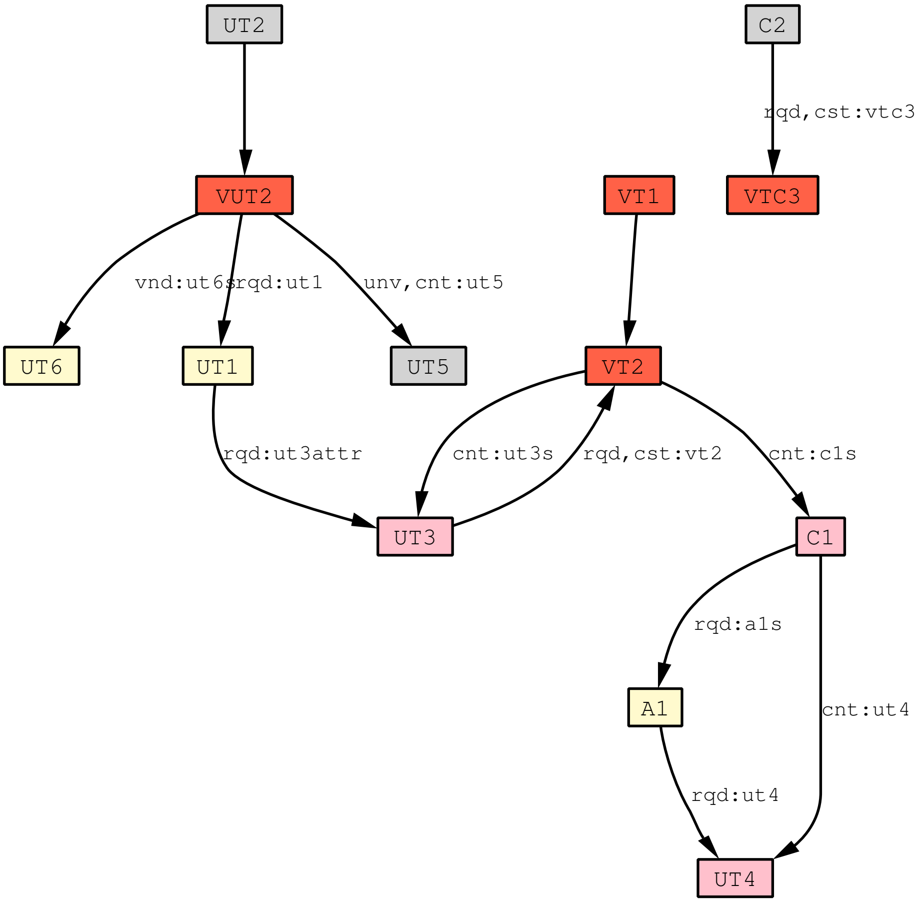


Figure 11-6. Versioning dependence graph

Red-colored nodes indicate those `object types` that are fully versioned by virtue of being marked versioned (`vnd`) or having a supertype that is marked `vnd`.

Pink-colored nodes represent those types that are *co-versioned* by virtue of being a component of a versioned or co-versioned `object type`.

Yellow-colored nodes denote those types that are treated as *recoverable* by virtue of being a required attribute of a versioned or recoverable type.

Gray nodes represent types that are ignored by the versioning service.

There are two kinds of edges in Figure 11-6:

1. Unlabeled edges represent the subtype relationship between two nodes. For example, `VUT2` extends `UT2`.
2. Labeled edges are those where the edge tail is an `object type` that has an attribute whose type is the edge head. For example, `VT2` has the component attribute `c1s` of type `C1`.

Four kinds of attributes are important from the versioning perspective: marked versioned (`vnd`), marked unversioned (`unv`), required (`rqd`) and components (`cnt`).

11.5.6. Differences between WAF 5.2 and 6.0

If you used the versioning service in previous releases of WAF prior to and including 5.2, you will notice a number of differences between the old and current implementation.

The old implementation operated on the domain object level. The current implementation operates on the data object level. Key differences that stem from this are:

1. Previously, you had to subclass `VersionedACSObject` in order to make your domain objects versioned. This is no longer necessary.
2. Consequently, you are no longer required to use the methods `getMaster()`, `setMaster()`, etc. The notion of master is gone. Although marked deprecated, these methods remain in order to provide some degree of backwards compatibility on the API level.
3. In the old versioning systems, calling the `delete()` method never actually deleted the underlying data object. It merely marked it as deleted via the is_deleted flag in the vc_objects table. This was referred to as a *soft delete*. It was possible to *hard-delete* an object via the `permanentlyDelete()` method of the `VersionedACSObject` class. This is no longer supported. In the current versioning system, all deletes are hard deletes.

   As a consequence, it is no longer necessary to write queries such as this:

   ```
   select
     f.id, frob_name
   from
     frobs f,
     vc_objects v
   where
     f.id = v.object_id
     and v.is_deleted = '0';
   ```

   Instead, one can simply write:

   ```
   select
     f.id, frob_name
   from
     frobs f
   ```

4. The old versioning provided the ability to turn off versioning on a per instance basis. The method `trackChanges()` could be overloaded to tell the versioning service whether or not this domain object wished to be versioned. The new versioning service no longer provides this capability. What it offers instead is the ability to turn off versioning entirely until the end of the current transaction by calling `Versions.suspendVersioning()`.

   It is possible to request that an attribute of an object be not versioned. For instance:

   ```
   versioned object type Frob {
     BigInteger[1..1] id = frobs.id INTEGER;

     unversioned String[0..1] type   = frobs.type VARCHAR;
   ```

```
  String[1..1] name    = frobs.frob_name VARCHARA;
}
```

In the above example, the versioning system will ignore changes to the `type` attribute of the `Frob` `object type`.

5. The current versioning service provides the ability to compute a diff between any two points in history of a versioned data object. The old service did not provide a similar capability.

Chapter 12.
Presentation (Bebop) Tutorial

This chapter provides useful tutorials for learning and exercising the presentation system using Bebop, as introduced in Chapter 6 *WAF Component: Presentation*. The focus is primarily on integrating other presentation methods such as CSS, CSL and JSP into WAF, while demonstrating how to use Bebop. This chapter does not provide any general tutorials on these other tools. For a starting point to finding tutorials about these methods, see Chapter 14 *References*.

12.1. Calling XSLT from a WAF Application

The gateway to XSLT from WAF is the `PresentationManager` interface, whose `servePage` method selects a stylesheet to use, applies the selected stylesheet to an XML document, and sends the output from the transformation to the client. This information was originally discussed in Section 6.2.1 *Integrating XSLT with WAF*.

The most commonly used implementation of `PresentationManager` is the provided `BasePresentationManager`, which searches for a stylesheet for a request by first looking for a stylesheet associated with the active site node, then the parents of the active site node, and finally the default stylesheet for the current package. It also takes the locale and output type for the request into account when selecting stylesheets.

Any selected stylesheet may use the `xsl:import` or `xsl:include` tags to import rules from another stylesheet. There are two common uses for this: a package's default stylesheet will import stylesheets from other packages that it depends on; and a customized stylesheet (for example, for a site node) can import another stylesheet and the rules defined in the customized stylesheet will override conflicting rules in the imported one.

Calling XSLT from WAF applications is transparent if you use Bebop — you pass a Bebop `Page` object directly as an argument to `BasePresentationManager.servePage()`. That form of `servePage` will save the step of explicitly generating an XML document as output from the `Page` object:

```
Page p = ... Bebop page ...;
BasePresentationManager.getInstance().servePage(p, request,
response);
```

Any WAF application code that wants to render an XML Document object into HTML output directly, without using Bebop, would call the `servePage` method of `PresentationManager` using approximately this pattern:

```
import com.arsdigita.templating.PresentationManager; import
com.arsdigita.xml.Document;

Document doc = ... generated document ...

// grab a presentation manager instance
PresentationManager pm = SomePMClass().getInstance();
pm.servePage(doc, request, response);
```

12.2. Handling Pre-Formatted Text

You may occasionally have HTML-preformatted text that you need to integrate into a Bebop component. This might come from a globalization translator, user input HTML, or as part of feed or stream from an outside source. This section discusses methods for handling this pre-formatted text intelligently.

WAF provides an explicit technique for this situation by using the Bebop `Label` component and by disabling output escaping in the XSLT processor for the context of the label. The code:

```
Label l = new Label("This is a <b>new</b> label.", false);
l.generateXML(pageState, element);
```

will generate the XML fragment:

```
<bebop:label escape="yes">This is a &lt;b>new&lt;/b&gt;
   label.</bebop:label>
```

The result is that the HTML-preformatted text is a single, text-node child to the `bebop:label` element. On output, the `escape="yes"` means that the angle brackets will *not* be output-escaped. They will appear in the HTML stream as angle brackets and be interpreted as such by the user's web browser, causing the word `new` to appear in bold.

Note

The `escape="yes"` attribute to `bebop:label` is confusing because it is shorthand for what will become the XSLT attribute `disable-output-escaping="yes"`.

By default, the XSLT processor escapes output according to the output mode specified in the stylesheet with the `xsl:output` element. This default makes angle brackets appear as angle brackets *in the user's browser*, which means they are transmitted as &lt;'s in the HTML stream.

If you want angle brackets to show up in your output (e.g less-than signs for mathematics), you want to enable output escaping, which is the default setting.

```
Label l = new Label("3.141 < pi < 3.142");
```

Caution

Inserting HTML formatting into the Bebop DOM, where it is difficult to style globally, is not generally recommended.

An alternative solutions is to parse the string argument to the `Label` constructor as a DOM fragment, instead of simply disabling output escaping, so that the formatting elements are a full-fledged part of the DOM and eligible for global styling. A drawback to this is the expense of parsing XML.

An additional drawback is that most people do not write well-formed XHTML, while browsers are much more tolerant than XML parsers. In other words, it is possible to write the following:

```
Label l = new Label("<ul> <li> list item 1 <li> list item 2
   </ul>", false);
```

This will be displayed correctly as an unordered list in the browser, even though the label contents are not well-formed [X]HTML.

12.3. Site-Wide Master Pages

Usually the pages within the scope of a single application share a common layout. For example, all of the pages may have the same four-panel layout: the same header, footer, and sidebar content, while the main content of the page varies. Another way of looking at this is that the main content of each page is enclosed in another *master* page.

There are two aspects to making shared-layout pages like this:

1. A declaration of page components in Java. With Bebop, the top-level `Page` object must contain the components that generate the dynamic content displayed in each of the header, footer, sidebar, and main panels as DOM fragments.
2. An XSLT stylesheet to render the page properly. The XSLT stylesheet is responsible for selecting the content for each of the panels on the page, and placing them appropriately in the output, with the right look-and-feel (color scheme, dimensions, etc.) Note that this means whatever DOM is generated by the page components above must allow for the XSLT stylesheet to distinguish between the content that belongs in the header, footer, etc.

The preferred way to implement shared-layout pages like this in Bebop is to subclass `Page`. Users of this class add components to the page as usual, but the page's constructor and DOM-generation methods are overloaded to pre-fill the page with the appropriate boilerplate components.

The following example is a `Page` that always contains a header, footer, and sidebar:

```
package com.arsdigita.bebop.demo;

import com.arsdigita.bebop.*;
import com.arsdigita.xml.*;
import com.arsdigita.dispatcher.RequestContext;
import com.arsdigita.dispatcher.DispatcherHelper;

/**
 * This is a common page for a fictitious site, SockPuppet.com.
 * It includes a common header, a footer, and a main "content"
 * area.  We override the .generateXML method to pre-fill the page
 * with the boilerplate content.
 */
public class SockPuppetPage extends Page {

    private Component m_top;
    private Component m_bottom;
    private Component m_side;

    public SockPuppetPage() {
        this("");
    }

    public SockPuppetPage(String s) {
        super("SockPuppet.com: " + s);
        m_top = new SiteHeader();
        m_bottom = new SiteFooter();
        m_side = new SiteSide();
    }

    public void generateXML(PageState ps, Document doc) {
        Element page = generateXML(doc);
        Element layout = new Element("socksite:layout", SOCKSITE_XML_NS);
        page.addContent(layout);

        addContents(layout, ps);
    }
    protected void addContents(Element layout, PageState ps) {
        Element topPanel =
           new Element("socksite:top", SOCKSITE_XML_NS);
        layout.addContent(topPanel);
        m_top.generateXML(ps, topPanel);

        Element sidePanel =
           new Element("socksite:side", SOCKSITE_XML_NS);
        layout.addContent(sidePanel);
        m_side.generateXML(ps, sidePanel);

        Element bottomPanel =
           new Element("socksite:bottom", SOCKSITE_XML_NS);
        layout.addContent(bottomPanel);
        m_bottom.generateXML(ps, bottomPanel);

        Element mainPanel =
           new Element("socksite:main", SOCKSITE_XML_NS);
        layout.addContent(mainPanel);
        m_panel.generateXML(ps, mainPanel);
    }

    /**
     * Header component.  Demonstrates dynamic content.
     */
    private class SiteHeader extends Label {
        public SiteHeader() {
            super(new PrintListener() {
                public void prepare(PrintEvent pevt) {
                    Label target = (Label)pevt.getTarget();
                    PageState ps = pevt.getPageState();
                    RequestContext rc =
                      DispatcherHelper.getRequestContext
                        (ps.getRequest());
                    target.setLabel("SockPuppet.com:
                      dynamic page header."
                       + "  You requested: " + rc.getOriginalURL());
                }
            }
        }
    }

    /**
     * Footer component.  All static.
     */
    private class SiteFooter extends Label {
        public SiteFooter() {
            super("SockPuppet.com: static footer.");
        }
    }

    /**
     * Sidebar component.  All static.
     */
    private class SiteSide extends Label {
        public SiteSide() {
            super("SockPuppet.com: static sidebar.");
        }
    }
}
```

You would use this `Page` like any other, for example, `Page p = new SockPuppetPage(); p.add(...); ...`. However, when you call `buildDocument`, the generated DOM will include all of the content from the header, footer, etc.:

```
<bebop:page>

... some page boilerplate ...

<socksite:top>
  <bebop:label>
    SockPuppet.com: dynamic page header.  You requested: ...
  </bebop:label>
</socksite:top>

<socksite:side>
  <bebop:label>
    SockPuppet.com: static side panel
  </bebop:label>
</socksite:side>

<socksite:bottom>
  <bebop:label>
    SockPuppet.com: static footer
  </bebop:label>
</socksite:bottom>

<socksite:main>
  ... main contents here ...
</socksite:main>

</bebop:page>
```

Note

The order of the components in the DOM may have nothing at all to do with the ordering of the output.

The page components just ensure that the right content is generated with the right logical structure. It is up to the XSLT stylesheet to ensure that a `socksite:top` element really appears at the top of the page.

You would need an XSLT template rule such as the following to convert the above DOM into meaningful XHTML output:

```
<!-- match the layout element and put the
header, footer, etc., in their appropriate places. -->

<xsl:template match="socksite:layout"
               xmlns:socksite="http://www.sockpuppet.com/xmlns">

 <xsl:apply-templates select="socksite:top"/>

 <table>
   <tr>
     <td width="25%"><xsl:apply-templates
        select="socksite:side"/></td>
     <td><xsl:apply-templates
        select="socksite:main"/></td>
   </tr>
 </table>

 <xsl:apply-templates select="bebop:bottom"/>

</xsl:template>
```

This template must be associated with the appropriate application (for example, the main **SockSite** application) or site node.

12.4. Varying a Shared Layout

Once you have a shared-layout Page subclass, such as `SockPuppetPage` above, you will likely have single pages that vary the boilerplate content slightly. For example, you might have a single page which uses the same header, footer, and sidebar as all other pages on your site, but which adds an extra boilerplate component above the header, such as a banner ad.

You can accomplish this with further sub-classing. You can define a subclass of `SockPuppetPage`, `BannerAdSockPuppetPage`, which has all of the same pre-filled components but adds a second header, a `BannerAd` component:

```
package com.arsdigita.bebop.demo;

import com.arsdigita.bebop.*;
import com.arsdigita.xml.*;
import com.arsdigita.dispatcher.RequestContext;
import com.arsdigita.dispatcher.DispatcherHelper;
// fictitious banner ad management system
import com.arsdigita.personalization.BannerAdManager;

public class BannerAdSockPuppetPage extends SockPuppetPage {

    private Component m_banner_ad;

    public BannerAdSockPuppetPage() {
        this("");
    }

    public BannerAdSockPuppetPage(String s) {
        // calling super() adds default header, footer, and sidebar
        super("SockPuppet.com: " + s);
        m_banner_ad = new BannerAd();
    }

    public void generateXML(PageState ps, Document doc) {
        Element page = generateXML(doc);
        Element layout = new Element("socksite:layout", SOCKSITE_XML_NS);
        page.addContent(layout);

        // generate XML for header, footer, sidebar, and main panel
        super.addContents(layout, ps);

        // just add the new banner ad here
        Element bannerAd = new Element("socksite:bannerAd",
           SOCKSITE_XML_NS);
        layout.addContent(bannerAd);
        m_banner_ad.generateXML(ps, bannerAd);
    }

    /**
     * Header component.  Demonstrates dynamic content.
```

```
      */
    private class BannerAd extends SimpleComponent {
        public void generateXML(PageState ps, Element parent) {
            Element elt = new Element("socksite:bannerAd",
                SOCKSITE_XML_NS);
            elt.setText(BannerAdManager.getInstance().getBanner());
            parent.addContent(elt);
        }
    }
}
```

You must also amend the XSLT stylesheet:

```
<!-- match the layout element and put the
header, footer, etc., in their appropriate places. -->

<xsl:template match="socksite:layout"
              xmlns:socksite="http://www.sockpuppet.com/xmlns">

 <xsl:apply-templates select="socksite:top"/>

 <xsl:apply-templates select="socksite:bannerAd"/>

 <table>
   <tr>
     <td width="25%"><xsl:apply-templates
        select="socksite:side"/></td>
     <td><xsl:apply-templates select="socksite:main"/></td>
   </tr>
 </table>

 <xsl:apply-templates select="bebop:bottom"/>

</xsl:template>
```

Note that the above stylesheet will work *whether or not* the input DOM actually contains a banner ad; the `<xsl:apply-templates select="socksite:bannerAd"/>` code will simply be ignored if no such element exists in the DOM.

12.5. Special-Case Stylesheets

You may occasionally want to override the default stylesheet rules for one particular page. Because of the way XSLT works, you can choose to override only some `template match=...` rules, allowing others to be handled by the default stylesheet.

You should handle this in the dispatcher for your application, or by overriding the `getStylesheet-Transformer()` or `servePage` methods in a `PresentationManager` class specific to your application. For example:

```
public class SockPuppetPresentationManager extends
    BasePresentationManager {

    public void servePage(Page p,
                          HttpServletRequest req,
                          HttpServletResponse resp)
        throws IOException, ServletException {

        RequestContext rctx = DispatcherHelper.getRequestContext(req);
        String url = rctx.getRemainingURLPart();
        // serve page as usual, except for URLs foo/*; those get a
        // special-case stylesheet
        if (!url.startsWith("foo/")) {
            super.servePage(p, req, resp);
        } else {
            // use custom stylesheet for foo/*
            // in reality, probably cache Transformer for performance
            Stylesheet fooSS = ... get overridden rules
               from somewhere ...;
            Transformer xformer = fooSS.newTransformer();
            Document doc = p.buildDocument(req, resp);
            Writer out = resp.getWriter();
            xformer.transform(new DOMSource(doc), new StreamResult(out));
        }
    }
}
```

12.6. UI Tutorial

The permissions UI provides reusable components for building customized permission administration interfaces.

The default implementation of it resides under /permissions/. It supports a standard set of permissions: READ, WRITE, CREATE, DELETE, ADMIN. A user or group with admin privilege on an object can share the privileges with other parties using this interface.

There are two requirements to show an object's permission tables.

- The viewing user must be authenticated by having logged so that the user ID can be read from the user cookie.
- The ACSObject ID must be in the page state as a global parameter (current implementation to support URL redirects and URL referencing in WAF Tcl style).

The first step is to create the embedding Bebop Page for the permissions UI. A simple example is ObjectPermissionsPage.java. This only has a UserAuthenticationListener in the constructor and builds a PermissionsPane using the default constructor. Note, that the default constructor will give you the permissions table with READ, WRITE, CREATE, DELETE, ADMIN privileges (as defined in PermissionsConstants.java). You can use this to build permissions tables with a privilege array customized for your application:

```
PermissionsPane(PrivilegeDescriptor[] privs)
```

The main class that provides components for the permissions UI is PermissionsPane. It has dual functionality:

1. It implements the generic permissions UI for ACSObjects,
2. It provides getter methods to retrieve only parts of the interface.

If you use the PermissionsPane class directly, you get a UI layout equivalent to what you see at http://yourhost.com/permissions/, using with your privileges instead. If you want to use only certain components of PermissionsPane, you must subclass it and overwrite the following public methods:

```
public void register(Page p),
public void showAdmin(PageState s),
public void showGrant(PageState s),
public void showNoResults(PageState s).
```

The register determines which and how many components you are going to use. The `show`*`Foo`* methods manage visibility of components. You can also write your own visibility manager in the subclass in a similar way, and just use that. You could also overwrite the other public methods, for example, to provide alternative or static components for faster testing, or to save a member variable in the subclass. Also, consider the `reset()` function.

The components that can be extracted from `PermissionsPane` are:

- `public Party getRequestingUser(PageState s)`
- `public ACSObject getObject(PageState s)`
- `public Label getTitle()`
- `public String[] getPrivileges()`
- `private PermissionsTables getPermissionsTables()`
- `public SimpleContainer getDirectPermissionsPanel()`
- `public SimpleContainer getInheritedPermissionsPanel()`
- `public Form getUserSearchForm()`
- `public SimpleContainer getPermissionGrantPanel()`
- `public SimpleContainer getPermissionsHeader()`
- `public SimpleContainer getNoSearchResultPanel()`
- `public SimpleComponent getContextPanel()`

These functions all use lazy instantiators, meaning they are not constructed until they are needed. Therefore instantiating `PermissionsPane` alone is not expensive, the components are computed only once and on the fly.

Note, if you want the same HTML style as in the default implementation, you MUST import the `permissions.xsl` stylesheet in the application's stylesheet, for example:

```
<xsl:import href="../../content-section/xsl/permissions.xsl"/>
```

Otherwise the Bebop components are rendered in their default style.

12.7. Working With Formbuilder

The **formbuilder** service makes it possible for non-technical application administrators to build forms (typically HTML forms). It has an API that lets developers mount those forms in an application. The developer is responsible for the processing of the form whereas the administrator takes care of the form elements and visual appearance.

12.7.1. Implementing and Registering a Process Listener

Your process listener must be a named Java class in order for it to be used by a persistent form. If your form listener needs certain parameters with specific names and data types, ensure that the listener implements the `com.arsdigita.formbuilder.AttributeMetaDataProvider` interface.

You register the listener by going to the **formbuilder** admin interface mounted under /formbuilder. On the index page, choose to add a new persistent form and supply the name of the listener that was implemented.

You may now notify any application administrator that the form is ready for editing. If the listener implements the `com.arsdigita.formbuilder.AttributeMetaDataProvider` interface, the **formbuilder** admin UI will make sure that the persistent form complies with this contract and submits the parameters that the listener needs.

12.7.2. Mounting a Persistent Form in an Application

The **formbuilder** admin UI represents persistent forms via the `com.arsdigita.formbuilder.SimpleQuestionnaire` class. You may retrieve an instance of this class via its integer id or via the admin name that you specified in the admin UI (the admin name is unique).

Once you have created the instance of `SimpleQuestionnaire` that you need, you have two options:

- The dynamic option involves adding a `com.arsdigita.bebop.MetaForm` to the page and will ensure that admin changes to the form will take immediate effect.
- The static approach involves adding a normal Bebop `com.arsdigita.bebop.Form` to your application page so that admin changes to the form will take effect only after server startup.

The dynamic approach is more convenient for the administrator, in that he can immediately see how the form will look in the application. However, if the page has a heavy load of users, the static approach is preferable, since it is much more efficient. This is because it does not dynamically build the component hierarchy of the form on every request.

Your decision to use static or dynamic will be based upon your usage patterns. One good method to follow is to use the dynamic approach when in heavy development of an application and it's forms, then switch to static when the pages go to the live server and are subject to public loads. It is usually preferred to keep the approach static on a non-production, site-development server. When you switch to the static approach, it is recommended to restart the server regularly (e.g. with a daily **cron** job) so as to capture changes made by application administrators.

Example 12-1 is an example of static mounting of a persistent form. In this case, the form is built only once upon server startup. The web server must be restarted for any changes to take effect.

```
Page applicationPage = new Page();
SimpleQuestionnaire questionnaire =
 new SimpleQuestionnaire(new BigDecimal("773"));
applicationPage.add(questionnaire.createComponent());
applicationPage.lock();
```

Example 12-1. Static Mounting of a Persistent Form

Example 12-2 is an example of dynamic mounting of a persistent form. Changes to the form by the administrator will take immediate effect on the application page.

```
Page applicationPage = new Page();
applicationPage.add(new MetaForm() {

    public MetaForm() {
        super("application_form_name");
    }

    public Form buildForm(PageState pageState) {

        SimpleQuestionnaire questionnaire =
         new SimpleQuesionnaire(new BigDecimal("773"));

        return questionnaire.createComponent();
    }
}
);

applicationPage.lock();
```

Example 12-2. Dynamic Mounting of a Persistent Form

Chapter 13. Web Applications Tutorials

13.1. Support for Globalization

This section discusses the *internationalization* (*I18n*) features of Java and WAF, and how to use this to localize web applications.

See Chapter 14 *References* for more references discussing the concepts of globalization.

WAF supports globalization in a variety of ways:

1. It allows the user to select a preferred `Locale`, either by registering a preference with WAF, or by sending the appropriate `Accept-Language HTTP` header with each request.
2. It allows the user to send and receive data in the user's preferred character set encoding.
3. It provides mechanisms for storing multiple localized versions of resources for an application. This is true for both static and dynamic resources.
4. It offers APIs for creating and accessing these localized resources.
5. It provides a globalized *WUI* (*Web User Interface*) toolkit — for more information, see Section 6.4 *Bebop - Reusable Web UI Components*.
6. It allows stylesheets (used for styling of content) to be associated with more than one `Locale`, thereby allowing you to create different layouts based on `Locale`. These will be applied automatically by the system as determined by the user's preferred `Locale`.

13.2. Locale Negotiation

When a user requests a URL, the server determines the appropriate `Locale` for the request. It does this by negotiating between the client's preferences and the server's capabilities.

All valid URLs on the server are associated with an instance of some application.

1. If that application instance has an associated `Locale`, this will be used as the `Locale` for the request. If the application instance does not have an associated `Locale`, the server checks whether the user has a preferred `Locale` as kept by the WAF preferences service.
2. If there is a preferred `Locale`, and if that `Locale` is supported by the application, that is the chosen `Locale` for the request.
3. If there is not a preferred `Locale` the server continues on to check the `Accept-Language HTTP` headers. Checking each of the values in this HTTP header in order of descending *q-value*, the server selects the first one that is supported by the application being served.
4. If no `Locale` can be determined, the server finally selects the default `Locale` for the server as the `Locale` for the request. This algorithm is well documented in the `LocaleNegotiator` class.

This negotiation allows applications to support one or many languages without interacting with other applications on the system. Thus, your application automatically tries to present itself to the user in his or her preferred language, regardless of the other applications running on that server.

Note

> If an application built on WAF relies on a service provided by another package, that package must also be globalized and must support any languages the WAF application is intended to support.
>
> For example, if your application relies on the **places** service to relate and/or present geographical data, you must ensure that this service supports all the `Locale`s that you wish your application to support.

You can retrieve the `Locale` for the current request by calling the `getLocale()` method of the current `RequestContext`. For example:

```
java.util.Locale locale =
   DispatcherHelper.getRequestContext(request).getLocale();
System.out.println("The locale for the current request is: "
   + locale.toString());
```

Another option, *character set encoding negotiation* (*charset negotiation*) is more complicated than `Locale` negotiation.

13.3. ResourceBundles and MessageCatalogs

`ResourceBundle`s [1] are Java classes for grouping localizable resources. The class also provides methods for finding the appropriate `ResourceBundle` based on a `Locale` and for retrieving resources from it by performing a lookup based on a key.

A `ResourceBundle` has a *base name* and optionally an associated `Locale`. A base name is a namespace used to group resources. A `ResourceBundle` without an associated `Locale` is the default `ResourceBundle` for that base name. If a `ResourceBundle` for a particular `Locale` *hierarchy* is not found, the `ResourceBundle` with no associated `Locale` is used. These are used to store resources that are common across all `Locale` instances.

The `Locale` hierarchy is simply the mechanism that is used to fall back from very specific `Locale`s to more generic ones. For example, from `en_US_WIN95`, the hierarchy falls back to `en_US` and then to `en` and finally to the empty `Locale`.

Typically, one base name is used per application. That is, all resources for an application or service are grouped in the same `ResourceBundle`. In cases where this is not convenient, different base names can be used.

An example of the normal use of `ResourceBundle`s is the **notes** application. The base name it uses is `com.arsdigita.notes.NotesResources`. It groups all its localizable resources into that `ResourceBundle`. On the other hand, the **places** service uses multiple base names, since it maintains a large amount of data, which may not be needed by all developers. For example, a developer might just want to translate the names of countries into a particular language, and not require any other place data. Therefore, the **places** service divides its resources into different base names, including `com.arsdigita.places.CountriesResources`, `com.arsdigita.places.USStatesResources` and so on.

It is recommended that you use `PropertyResourceBundle` [2] to store *static resources*. Static resources remain the same for the lifetime of the running server. This includes strings on WUI elements (such as form labels, page titles, and error messages) and names of objects that do not change frequently (such as country and city names).

[1] http://java.sun.com/j2se/1.3/docs/api/java/util/ResourceBundle.html

[2] http://java.sun.com/j2se/1.3/docs/api/java/util/PropertyResourceBundle.html

For example, `PropertyResourceBundle`s might appear as follows for a **HelloWorld** application, in English, Spanish, and French:

```
HelloWorldResources_en.properties

    hello_world=Hello world!

HelloWorldResources_es.properties

    hello_world=Hola todo el mundo!

HelloWorldResources_fr.properties

    hello_world=Salut tout le monde!
```

For *dynamic resources*, the `MessageCatalog` class is provided. Dynamic resources are those that change while the server is running. Most of these have to do with user-contributed content, where a user can be an administrator of the site or a regular user. For example, if the `Categorization` service was internationalized and application users were to be allowed to create categories, the application could allow the user to create a category and name it in more than one language, or mark the category name for translation into the different languages that the application supports. Since this kind of data is hard to maintain in `PropertyResourceBundle`s, it is stored it in database-backed `MessageCatalog`s.

A `MessageCatalog` is simply a way to store `ResourceBundle`s in the database. To use them, you create a `MixedResourceBundle` . This inherits from Java's `ResourceBundle` and combines a `PropertyResourceBundle` and a `MessageCatalog`, which have the same base name, into one. It does this by reading the `PropertyResourceBundle` from the file system and the `MessageCatalog` from the database.

No code goes into a `MixedResourceBundle`; all the code is in the superclass itself. A `MixedResourceBundle` for the **notes** application might appear as follows:

```
package com.arsdigita.notes;

import com.arsdigita.globalization.MixedResourceBundle;

public class NotesResources_en_US extends MixedResourceBundle {}
```

Note

> This `MixedResourceBundle` is for the base name `com.arsdigita.notes.NotesResources` and the `Locale en_US`. This stems from the `ResourceBundle` name and indicates the `Locale` associated with the `ResourceBundle`. In this case, this `ResourceBundle` should contain resources in American English.

13.4. Accessing Resources

The `GlobalizedMessage` class is specifically designed to facilitate management and access to localized resources. This class represents a single globalized resource. It contains the base name of the `ResourceBundle`, which contains the resource and the `key` to use to lookup the resource, and an optional array of arguments to interpolate into the localized resource using Java's `MessageFormat`

[3] class. It also contains a `localize(java.util.Locale)` method, which is used to localize the resource to a particular `Locale`.

You should use this class to represent all localizable data for your application. In Section 13.5 *Globalization and Bebop*, you will learn how to pass `GlobalizedMessage`s to Bebop components so that they will perform the lookup when it is time to display the localized resource to the user.

The following example demonstrates how to access the `hello_world` resource defined in `PropertyResourceBundle`. Assuming that those files are in the `com.arsdigita.helloworld` package, you would create a `GlobalizedMessage` to represent that resource and localize it as follows:

```
GlobalizedMessage message = new GlobalizedMessage(
    "hello_world",
    "com.arsdigita.helloworld.HelloWorldResources"
);

// print in English:
System.out.println((String) message.localize(new Locale("en", "", "")));

// print in Spanish:
System.out.println((String) message.localize(new Locale("es", "", "")));

// print in French:
System.out.println((String) message.localize(new Locale("fr", "", "")));
```

The output will appear as:

```
Hello world!
Hola todo el mundo!
Salut tout le monde!
```

You can also make a parameter out of resources that will be interpolated using the `MessageFormat` class. The next example will use the following `PropertyResourceBundle`s:

```
WineResources_en.properties

    this_wine_is=This is a {0} wine.
wine_color=red

WineResources_es.properties

    this_wine_is=Este es un vino {0}.
wine_color=rojo

WineResources_fr.properties

    this_wine_is=C'est un vin {0}.
wine_color=rouge
```

It is possible to create a `GlobalizedMessage` to perform the proper interpolation of the resource. For example, this works if the `wine_color` resource is retrieved from a `MessageCatalog` and is not stored statically in the `PropertyResourceBundle`.

```
GlobalizedMessage message = new GlobalizedMessage(
    "this_wine_is",
    "com.arsdigita.wine.WineResources",
    {"wine_color"}

);
```

[3] http://java.sun.com/j2se/1.3/docs/api/java/text/MessageFormat.html

```
// print in English:
System.out.println((String) message.localize(new Locale("en", "", "")));

// print in Spanish:
System.out.println((String) message.localize(new Locale("es", "", "")));

// print in French:
System.out.println((String) message.localize(new Locale("fr", "", "")));
```

The output will appear as follows:

```
This is a red wine.
Este es un vino rojo.
C'est un vin rouge.
```

Note

When a `key` is not found in a `ResourceBundle`, or when the `ResourceBundle` is not found at all, the `localize(java.util.Locale)` method returns the `key`. This is done for two reasons: first, it is more user-friendly to display the `key` than it is to display an exception or error message. Second, this method makes it easier for a translator to see all the resources that need to be translated on a particular page.

Caution

There are `GlobalizedMessage` constructors that do not have the base name of the `ResourceBundle` as a parameter. When these constructors are used, the base name of the `ResourceBundle` for the current application is used. This information is retrieved from the current `ApplicationContext`.

Using these constructors is discouraged because the method is non-deterministic with regard to changes in the application stack. For example, if an application written by one developer — e.g. **notes** — runs on top of a **portal** application written by another developer, and it is using these certain `GlobalizedMessage` constructors, an error can occur.

The first application is no longer the *current running application*, and therefore the base name of the `ResourceBundle` for the `GlobalizedMessage`s will be set incorrectly. It will be set to the base name of the `ResourceBundle` for the **Portal** application.

Only use these constructors if you are sure that your application will always be at the top of the application stack.

13.5. Globalization and Bebop

Bebop is enabled to use `GlobalizedMessage`s wherever the localizable content is displayed to the user.

To display localized content to the user using Bebop, pass a `GlobalizedMessage` to the Bebop widget. For example, to write a globalized **HelloWorld** page using Bebop:

```
package com.arsdigita.helloworld;

import com.arsdigita.bebop.Label;
import com.arsdigita.bebop.Page;
import com.arsdigita.dispatcher.Dispatcher;
import com.arsdigita.globalization.GlobalizedMessage;

public class HelloWorldDispatcher extends Dispatcher {
    // do some stuff, set up URL to method map, etc.

    private Page buildHelloWorldPage() {
        Label message = new Label(new GlobalizedMessage(
            "hello_world",
            "com.arsdigita.helloworld.HelloWorldResources"
        ));

        Page page = new Page(message);
        page.add(message);
 page.lock();
 return page;
    }
}
```

When the `Page` is served, Bebop will automatically call the `localize(java.util.Locale)` method of each of the `GlobalizedMessage`s on the `Page` with the `Locale` of the current request as an argument. This will allow the pages to be displayed to each user in his or her preferred language.

13.6. Localizing Stylesheets

WAF uses XSL stylesheets to lay out XML content that is usually produced by Bebop, but which could be produced by other sources. For more information, see Section 6.4 *Bebop - Reusable Web UI Components*. These stylesheets can be associated with `Locales`. This allows a developer to lay out a `Page` differently based on `Locale`.

Stylesheets can also contain static text. This text can be translated in another stylesheet that is associated with the appropriate `Locale`.

The system selects the stylesheet to use for each request by matching it to the `Locale` of the current request. It performs matching in the same way that `ResourceBundles` do, by using the `Locale` hierarchy.

13.7. Sending Mail Messages

Sending Plain Text Messages

Plain text messages can be sent using a static convenience method:

```
Mail.send(to, from, subject, body);
```

You can use the standard format for email addresses, for example, `person@mydomain.net`, `Person person@mydomain.name`, and `Person LastName person@mydomain.net`. Multiple recipients can be specified in a single string separated by commas.

To access the complete set of header attributes, create a `Mail` object, invoke the various set methods, and call send:

```
Mail msg = new Mail();
msg.setTo("User1 user1@example.com, User2 user2@example.com");
msg.setFrom("Me me@example.com");
msg.setReplyTo("User3 user3@example.com");
msg.setCc("User4 user4@example.com");
msg.setBcc("User5 user5@example.com");
msg.setSubject("this is the subject");
msg.setBody("this is the body");
msg.send();
```

Sending Rich Text Messages

Rich text messages (i.e. HTML) require using the `setBody(html,alternate)` method to specify both the HTML and plain text versions of a message to send out:

```
Mail msg = new Mail(to, from, subject);
msg.setBody("<p>This is the HTML body</p>",
            "This is the plain text body");
msg.send();
```

Attachments

Message attachments are handled using one of the various `attach(...)` methods. Support is provided for File, URL and ByteArray datasources. The URL datasource is the simplest to use (and can also fetch from a local file using the appropriate protocol specification). All attachments require a name, a description, and an optional disposition. The MIME type is determined automatically.

For example, the following code fragment shows how to attach an image whose content is fetched from a URL:

```
URL  url = new URL("http://example.com/image.gif");
Mail msg = new Mail(to, from, subject, body);
msg.attach(url, "image.gif", "A sample image");
           msg.send();
```

The protocol for attaching content from a local file is similar:

```
File path = new File("image.gif");
Mail msg  = new Mail(to, from, subject, body);
msg.attach(path, "image.gif", "A sample image");
msg.send();
```

The Mail service provides a utility class called `ByteArrayDataSource` that can be used to attach virtually any type of content contained in memory by supplying its data as a `byte[]` array.

Chapter 14. References

The following references are pointers to additional information that is relevant to WAF but beyond the scope of this guide.

Martin Fowler's book *Patterns of Enterprise Application Architecture* covers many of the design patterns used in WAF. For more information, see http://www.martinfowler.com/books.html#eaa.

For more information regarding *object-relational persistence* see the following third-party links:

- http://www.rational.com/products/whitepapers/296.jsp
- http://www.ambysoft.com/persistenceLayer.pdf

The OpenACS (http://www.openacs.org) project is the community-driven development of the previous generation Web Application Framework architecture written in the TCL programming language. Many of the design concepts that appear in Red Hat WAF had precursors in the ACS TCL and OpenACS world. In particular:

- http://openacs.org/doc/openacs-4-6-3/object-system-design.html is the design document that covers the basic object model that was the ancestor of the WAF object model.
- http://openacs.org/doc/openacs-4-6-3/groups-design.html discusses the basic groups model that was the ancestor of the current WAF group model.

The Java programming language provides extensive support for globalization:

- The `java.text` Java package (http://java.sun.com/j2se/1.3/docs/api/java/text/package-summary.html). This class contains most of Sun's globalization classes.
- The `java.util.Locale` class is Sun's representation of a `Locale`, and can be found at http://java.sun.com/j2se/1.3/docs/api/java/util/Locale.html.
- The `java.util.ResourceBundle` class and its subclasses help maintain and retrieve externalized localized resources. It can be found at http://java.sun.com/j2se/1.3/docs/api/java/util/ResourceBundle.html.

The official W3C Recommendations are the definitive references on XSLT, XPath expressions, and related standards.

- XSLT — (http://www.w3.org/TR/xslt)
- XPath — (http://www.w3.org/TR/xpath).
- XHTML 1.0 (http://www.w3.org/TR/xhtml1).
- Canonical XML 1.0 (http://www.w3.org/TR/xml-c14n).

Section C.2 *Java Coding Standards — References and Related Reading* has links to Java coding standards and best practices.

IV. Appendixes

Table of Contents

Appendix A.
Bebop Tag Library Reference

A.1. Bebop/JSP

Tag	Description
`<show:all/>`	Simplifies showing all the components of a page. One such directive is the equivalent of including a `<show:component>` directive for each component you have defined for your page
`<show:page [pageClass=...] [pmClass=...] [master=...]> ... </show:page>`	Sets up a Bebop page for use in the JSP. Page is specified by class name `(pageClass).` The `pageClass` option is omitted if the page is set up in the same JSP (by including a page-definition JSP) or within a master page. The resulting XML produced by the JSP is then displayed by the presentation manager class specified by `pmClass` (default `BasePresentationManager`).
`<show:component name=... />`	Shows a component from the Bebop page in the JSP. There is no need for more specific tags for each component here. The component is displayed with default global styling.
`<show:form name=...> ... </show:form>`	Shows a form from the Bebop page in the JSP. If the form tag contains a body, it is *not* displayed with the default global styling though the <form> tag and page state (input type=hidden...) will be preserved. If the tag body is empty, though, the entire form will be displayed with default styling.
`<show:list name=...> ... <show:listItem/> ... </show:list>`	Shows the contents of a model-backed List from Bebop page in the JSP. The body of the tag will be displayed for each item of the list, so this tag is effectively a loop. The list itself is *not* displayed with the default global styling though the individual list items contained within will be where indicated by <show:listItem/>. Like with show:form, if the tag body is empty, the entire list will be displayed with default styling.
`<show:slave/>`	Includes the contents of the slave page into this point in the master page.

Table A-1. Tags available for displaying Bebop components

A.2. Available Page Definition Tags

Caution

These tags are highly experimental.

Tag	Description
`<define:page name="..." title="..." [pageClass="..."]`	The `define:page` tag is the top-level tag for Bebop-JSP integration, and it is responsible for several processing actions: — Declaring a Bebop `Page` object. — Generating an XML document from Bebop. — Putting this XML document into the request attributes for the show:page tags. `name` is the name of the `pageContext` attribute that refers to the Bebop page. The optional `pageClass` attribute is the name of the `Page` class that will be used as the base class for the created page.
`<define:image [name=...] src=... [height=...] [width=...] [alt=...] [border=...]/>`	Creates an `Image` object at this point in the page.
`<define:link [name=...] url=.../>`	Creates an `Link` object at this point in the page.
`<define:form name=... [method=...] [action=...] [encType=...]>`	The `define:form` tag indicates that a Bebop form should be created at this particular location in the page.
`<define:text name=... [size=...] [maxLength=...] [type=...]/>`	Creates a `TextField` at this particular location in the page.
`<define:textArea name=... [rows=...] [cols=...] [wrap="nowrap|soft|hard"]/>`	Creates a `TextArea` at this particular location in the page.
`<define:submit name=... [label=...] [bundle=...]/>`	Creates a form submit **button** at this particular location in the page, with an optional specified **label**. If a resource bundle is specified the **label** is used as a key.
`<define:radioGroup name=...>`	Creates a radio group. Must be enclosed within a form.
`<define:checkboxGroup name=... [vertical="true"]>`	Creates a checkbox group. Must be enclosed within a form. Checkboxes are arranged horizontally by default, but can be arranged vertically with the `vertical` attribute.
`<define:select name=...>`	Creates a selection widget. Must be enclosed within a form.
`<define:multipleSelect name=...>`	Creates a multi-selection widget. Must be enclosed within a form.

Tag	Description
`<define:option name=... [value=...] [selected=...] [bundle=...] />`	defines an option within a select widget or checkbox/radio group.
`<define:component classname=...>`	Creates a component of the given class and adds it at the current location in the page. Requires that the component class have a no-argument constructor.

Table A-2. Tags available for defining Bebop components

Note that if a JSP tag does not exist for adding a particular component to a page, it is trivial to create. As a last resort, you can use the generic `define:component` tag if the class has a no-argument constructor. It is also trivial to escape into Java from a JSP and create a component object explicitly in a Java scriptlet:

```
<define:page name="p" title="page title">
  <%
    MyComponent c = new MyComponent();
    p.add(c);
  %>
</define:page>
```

Appendix B.
PL/SQL Standards

Like any other part of WAF, PL/SQL code must be maintainable and professional. This means that it must be consistent and therefore must abide by certain standards. The standards will ensure that our product will be useful long after the current people building and maintaining it are around. This chapter addresses some standards and guidelines that will help us achieve this goal.

This chapter is written from the experience and perspective of the Red Hat Applications development team. Because our product is open source, these standards are useful for all developers to understand and incorporate.

B.1. General

1. All PL/SQL code must be well documented. We must write code that is maintainable by others. This is especially true in our case because we are building an open source toolkit, so anyone can download and browse the source code. Our motto is, "Document like you are trying to impress your favorite programming professor."
2. It is important to be consistent throughout an application as much as is possible given the nature of team development. This means carrying style and other conventions such as naming within an application, not just within one file.

B.2. Coding Standards

1. Encapsulation of related functionality is key to making our software maintainable and upgradeable. Try to bundle your code into packages whenever possible. This will make upgrading, bug fixing, customizing, and many other things, a possibility.
2. When creating functions or procedures, use the following template. It demonstrates most of the guidelines set forth in this document that correspond to functions and procedures:

   ```
   create or replace procedure|function <proc_or_func_name> (
   <param_1>    in|out|inout <datatype>,
   <param_2>    in|out|inout <datatype>,
   ...
   <param_n>    in|out|inout <datatype>
   )
   [return <datatype>]
   is
   <local_var_1>    <datatype>
   <local_var_2>    <datatype>
   ...
   <local_var_n>    <datatype>
   begin
   ...
   end <proc_or_func_name>;
   /
   show errors
   ```

3. Always use `create or replace procedure|function <proc_or_func_name>`. It makes reloading packages much easier and painless to someone who is upgrading or fixing a bug.

4. Always qualify `end` statements. The `end` statement for a package should be `end` *<package_name>*`;`, not just `end;`. The same is true for procedures, functions, package bodies, and triggers.

5. Always use the `show errors` SQL*Plus command after each PL/SQL block. It will help you debug when there are compilation errors in your PL/SQL code.

6. Name parameters as simply as possible. That is, use the column name if the parameter corresponds to a table column. The syntax `v_*` and `*_in` is being deprecated in favor of the named parameter notation. Therefore, of these two examples, the first one is preferred:

```
acs_user.create(first_names => 'Jane',
last_name => 'Doe', etc.)
acs_user.create(first_names_in => 'Jane',
last_name_in => 'Doe', etc.)
```

   To achieve this we must fully qualify arguments passed into procedures or functions when using them inside a SQL statement. This will get rid of any ambiguities in your code by telling the parser when you want the value of the column and when you want the value from the local variable.

   For example:

```
create or replace package body mypackage
     .
     .
     procedure myproc(party_id in parties.party_id%TYPE) is begin
         .
         .
         delete
         from parties
         where party_id = myproc.party_id;
         .
         .
     end myproc;
     .
     .
end mypackage;
/
show errors
```

7. Explicitly designate each parameter as *in*, *out*, or *inout*.

8. Each parameter should be on its own line, with a tab after the parameter name, then *in*/*out*/*inout*, then a space, and finally the datatype.

9. Use `%TYPE` and `%ROWTYPE` whenever possible.

10. Use `t` and `f` for booleans, not the PL/SQL `boolean` datatype because it cannot be used in SQL queries.

11. All `new` functions (for example, `acs_object.new, party.new,` etc.) should optionally accept an ID:

```
create or replace package acs_object
as
function new (
    object_id       in acs_objects.object_id%TYPE default null,
    object_type     in acs_objects.object_type%TYPE default 'acs_object',
    creation_date   in acs_objects.creation_date%TYPE default sysdate,
    creation_user   in acs_objects.creation_user%TYPE default null,
    creation_ip     in acs_objects.creation_ip%TYPE default null,
    context_id      in acs_objects.context_id%TYPE default null
) return acs_objects.object_id%TYPE;
```

The function above takes the optional argument `object_id`. Do this to allow people to use the same API call when they are doing double-click protection. That is, they have already gotten an `object_id` and now they want to create the object with that `object_id`.

B.3. Coding Style

Some general style guidelines to follow for the purpose of consistency across applications.

1. Standard indentation is four spaces. Our PL/SQL code is not only viewable in the SQL files but also through our SQL and PL/SQL browsers. This means that we should try to make it as consistent as possible to all source code readers.
2. Lowercase everything, with the exception of %TYPE and %ROWTYPE.

B.4. Constraint Naming Standards

A constraint naming standard is important for one primary reason: The `SYS_*` name **Oracle** assigns to unnamed constraints is not very understandable. By correctly naming all constraints, we can quickly associate a particular constraint with our data model. This gives us two real advantages:

- We can quickly identify and fix any errors
- We can reliably modify or drop constraints

Why do we need a naming convention? **Oracle** limits names, in general, to 30 characters, which is hardly enough for a human-readable constraint name.

B.4.1. Abbreviations

We propose the following naming convention for all constraints, with the following abbreviations taken from the **Oracle** documentation. Note that we shortened all of the constraint abbreviations to two characters to save room.

Constraint type	Abbreviation
`references` (foreign key)	fk
`unique`	un
`primary key`	pk
`check`	ck
`not null`	nn

B.4.2. Format of Constraint Name

`<table name>_<column_name>_<constraint abbreviation>`

In reality, this won't be possible because of the character limitation on names inside **Oracle**. When the name is too long, we will follow these steps in order:

1. Abbreviate the table name with the table's initials (for example, users -> u and users_contact -> uc).

2. Truncate the column name until it fits.

```
create table example_topics (
    topic_id    integer
        constraint example_topics_topic_id_pk primary key
);

create table constraint_naming_example (
    example_id           integer constraint cne_example_id_pk primary key,
    one_line_description varchar(100)
        constraint cne_one_line_desc_nn not null,
    body                 clob,
    up_to_date_p         char(1) default('t')
        constraint cne_up_to_date_p_check check(up_to_date_p in ('t','f')),
    topic_id
        constraint cne_topic_id_nn not null
        constraint cne_topic_id_fk references example_topics,
    -- Define table level constraint
    constraint cne_example_id_one_line_unq
        unique(example_id, one_line_description)
);
```

Example B-1. Example of Constraint Name

Note

- If you have to abbreviate the table name for one of the constraints, abbreviate it for all the constraints.
- If you are defining a multicolumn constraint, try to truncate the two column names evenly.

B.4.3. Naming `primary keys`

Naming `primary keys` might not have any obvious advantages. However, in Example B-2, the `primary key` helps make the SQL query clearer.

```
SQL> set autotrace traceonly explain;

SQL> select * from constraint_naming_example, example_topics
where constraint_naming_example.topic_id = example_topics.topic_id;

Execution Plan
----------------------------------------------------------
0   SELECT STATEMENT Optimizer=CHOOSE
1    0   NESTED LOOPS
2    1     TABLE ACCESS (FULL) OF 'CONSTRAINT_NAMING_EXAMPLE'
3    1     INDEX (UNIQUE SCAN) OF 'EXAMPLE_TOPICS_TOPIC_ID_PK' (UNIQUE)
```

Example B-2. Primary Key Naming

Being able to see `EXAMPLE_TOPICS_TOPIC_ID_PK` in the trace helps us to know exactly which table **Oracle** is querying.

If we had not named the constraints, the execution plan would look like:

```
Execution Plan
----------------------------------------------------------
0        SELECT STATEMENT Optimizer=CHOOSE
1    0     NESTED LOOPS
2    1       TABLE ACCESS (FULL) OF 'CONSTRAINT_NAMING_EXAMPLE'
3    1       INDEX (UNIQUE SCAN) OF 'SYS_C00140971' (UNIQUE)
```

The `SYS_C00140971` by itself provides no information as to which index is being used in this query, and more importantly, the name of this constraint will vary from database to database.

Mark Lindsey (<`lindsey@acm.org`>) provided another good reason to name `primary keys` and `unique` constraints. **Oracle** creates an index for every `primary key` and `unique` constraint with the *same name as the constraint*. It is an unfortunate DBA who has to wrestle with storage management of tens of mysteriously-named indexes.

B.4.4. Naming `not null` Constraints is Optional.

Red Hat Applications developers are undecided on whether or not to name `not null` constraints. If you want to name them, please do so and follow the above naming standard. Currently, naming `not null` constraints is *not* a requirement of WAF.

Note

Naming the `not null` constraints does not help immediately in error debugging (for example the error will say something like Cannot insert null value into column). We do recommend naming `not null` constraints to be consistent in our naming of all constraints.

B.4.5. Upgrade Scripts

Data model upgrade scripts are a crucial part of database-based applications. Standards help to ensure that upgrade scripts behave correctly and consistently. This helps save time in development, maintenance, and support.

Tip

An upgrade script should be written anytime that a change is made to a component's sql creation script that results in a different schema and/or data set. The new upgrade script should be written so that, when applied to the previous creation script, it results in the same schema and data set.

Every component should have exactly zero or one upgrade script per release per supported database. However, an upgrade script may source other files.

Upgrade scripts go in an `upgrade/` directory below the directory containing the corresponding creation script. The name of the file should be <*component-name*>-<*old-version-name*>-<*new-version-name*>`.sql`.

Example:

```
cms/sql/oracle-se/upgrade/cms-4.6.4-4.6.5.sql
```

Extending this process farther, there should be a single upgrade script for all of WAF per version. This upgrade script lives in `kernel/sql/oracle-se/upgrade` and is called `core-platform-<`*`old-version-name`*`> -<`*`new-version-name`*`>.sql`.

Similar to `core-platform-create.sql`, this script will not have its own SQL commands, but will simply source other files. Updating this script is the responsibility of each component developer.

Appendix C.
Java Standards

> *A good developer knows that there is more to development than programming. A great developer knows that there is more to development than development.*
>
> *(Ambler, 1999)*

The purpose of this document is to establish a set of standards for any code written in Java for use in WAF.

Why have conventions?

- Most of the lifetime cost of a piece of code is maintenance.
- Over its lifetime, code is usually maintained by people other than the original author.
- Conventions improve the readability of code by providing a consistent level of quality.
- Open source code is shipped as one of the pieces of the product. The code must be as clean and well packaged as other pieces of the product.
- Code is not written for compilers alone. People read your code, and style matters. For an approach to English, a good starting point is *The Elements of Style by Strunk & White* [1].

C.1. WAF Standards

In general, when programming for WAF, follow the Sun Java, JSP, and Javadoc conventions:

- Code Conventions for the Java Programming Language — http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
- Code Conventions for the JavaServer Pages Technology Version 1.x Language — http://developer.java.sun.com/developer/technicalArticles/javaserverpages/code_convention/
- Writing Javadoc comments — http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

This chapter discusses WAF coding conventions where they differ from the Sun conventions or if they cover something that is not in the standard convention.

1. Import statements should be alphabetized, separated into sections of relevant functionality, and enumerated without using `*`.
2. All lines of source code should be indented uniformly using spaces and *not* tabs. Four spaces is the norm, although in some cases more space is required. Two spaces is forbidden under all circumstances.
3. Class fields, unless constants, are `private`. A private field called foo is named `m_foo`. A static private field called foo is named `s_foo`.
4. Exceptions are used to signal erroneous behavior or usage inside of a class. All stubbed methods should throw a runtime Error or UnsupportedOperationException with appropriate comments. Do not write generic `catch (Exception e)` blocks; catch the specific exceptions instead.

[1] http://www.bartleby.com/141/.

5. Acronyms, such as JDBC or URL, should always be capitalized when used. For instance, write `JDBCLoader` instead of `JdbcLoader`. This makes it clear that the letters are part of an acronym and is easier to read because it matches how it is referred to in English prose. Abbreviations that are not acronyms should have their first letter capitalized, with the exception of ID.
6. When debugging, avoid using `System.out.println` or `System.err.println`. Instead, use Log4j as described in Section 8.5 *Using logging for debugging*.

C.2. Java Coding Standards — References and Related Reading

- Ambler, S.W. (1999) *Writing Robust Java Code* — http://www.ambysoft.com/javaCodingStandards.html.
- Davis, M. (2000) *Incremental Development with Ant and JUnit* — http://www-4.ibm.com/software/developer/library/j-ant/index.html.
- Hunt, A. & Thomas, D. (1999) *The Pragmatic Programmer* — http://www.pragmaticprogrammer.com/ppbook/index.shtml.
- Sun (1999) *Code Conventions for the Java™ Programming Language* — http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html.
- JUnit Home Page — http://www.junit.org/
- Ant home page — http://jakarta.apache.org/ant/.

Appendix D. PDL Syntax

This chapter introduces the syntax for the Persistence Definition Language.

D.1. PDL Grammar

The following is an Extended Backus-Naur Form (EBNF) grammar that defines the current PDL language. EBNF is a standard notation for formally defining a language that extends BNF (they are functionally equivalent -- EBNF simply introduces some elements for better readability).

In this grammar, items within angle braces <*ITEM*> are terminals (tokens), while those not within angle brackets are non-terminals. If you are unfamiliar with EBNF, http://www.garshol.priv.no/download/text/bnf.html is a good starting point.

```
/** Top level constructs. */

 file :=
     model <SEMI>
     ( pdl_import <SEMI> )*
     ( object_type | association | data_operation )*
     <EOF>

 model := <MODEL> idpath

 pdl_import := <IMPORT> idpath ( <DOT> <STAR> )?

 object_type :=
    ( <VERSIONED> )?
    ( <QUERY> | <OBJECTTYPE> ) id
    ( ( <EXTENDS> type |
        <CLASS> javaClass <ADAPTER> javaClass |
        <RETURNS> <INT> <DOT> <DOT> ( id | <INT> ) ) )?
    <LBRACE> ( statement )* <RBRACE>

 association :=
     <ASSOCIATION> <LBRACE> property <SEMI> property <SEMI>
     ( property <SEMI> )*
     ( option_block )?
     ( event )* <RBRACE>

 data_operation :=
     <DATA_OPERATION> id <LBRACE> ( option_block )? sql_block <RBRACE>


/** First level constructs. */

 statement := ( simple_statement <SEMI> | compound_statement )
 simple_statement :=
     ( property_stmt | object_key | reference_key |
       unique_key | aggressive_load | join_stmt )

 compound_statement :=      ( event | option_block )


/** Simple statements */

 property_stmt := property
 property :=
     ( <IMMEDIATE> )?
     ( ( <VERSIONED> | <UNVERSIONED> ) )?
     ( <UNIQUE> )?
     ( ( <COMPONENT> | <COMPOSITE> ) )?
     type ( multiplicity )? id
     ( <EQ> ( column | join_path ) )?

 multiplicity := <LBRACKET> integer <DOT> <DOT> ( id | integer ) <RBRACKET>

 object_key := <OBJECTKEY> <LPAREN> id ( <COMMA> id )* <RPAREN>

 reference_key := <REFERENCEKEY> <LPAREN> column <RPAREN>

 unique_key := <UNIQUE> <LPAREN> id ( <COMMA> id )* <RPAREN>

 aggressive_load := <AGGRESSIVE> <LPAREN> path ( <COMMA> path )* <RPAREN>

 join_stmt := join_path


/** Compound statements */

 event :=
     ( ( ( <INSERT> | <UPDATE> | <DELETE> ) |
       ( ( <ADD> | <REMOVE> | <CLEAR> ) ( id )? ) |
       ( <RETRIEVE> ( <ALL> | <ATTRIBUTES> | id )? ) )
       <LBRACE> ( sql_block | <SUPER> <SEMI> )*
       <RBRACE> | sql_block )

 sql_block :=
     ( <CALL> | <DO> ) <SQL>
     ( <MAP> <LBRACE> ( mapStatement <SEMI> )+ <RBRACE> )?

 mapStatement := ( binding | mapping )

 binding := path <COLON> db_type

 mapping := path <EQ> path

 option_block := <OPTIONS> <LBRACE> ( option <SEMI> )+ <RBRACE>

 option := id <EQ> optionValue

 optionValue := <TRUE> | <FALSE> | <STRINGLIT>

/** Shared definitions */

 join_path := join ( <COMMA> join )*

 join := <JOIN> column <TO> column

 column := id <DOT> id ( db_type )?

 db_type := id ( <LPAREN> integer ( <COMMA> integer )? <RPAREN> )?

 type := idpath

 path := idpath

 javaClass := idpath
 idpath := id ( <DOT> id )*

 id := <ID>
 integer := <INT>
```

D.2. PDL Reserved Words

D.2.1. PDL Keywords

The following list, in conjunction with the Section D.2.2 *PDL Attribute Types*, details all PDL reserved words. Explanations are provided where applicable. Examples of how to use almost all of these keywords can be found in the Chapter 9 *Persistence Tutorial*.

D.2.1.1. Escaping PDL Keywords

Similar to most other languages, the PDL reserved words are not valid except in the context defined for them unless they are properly escaped. For instance, naming something "composite" or "component" will cause a compilation error because the parser found a "composite" token instead of an "id" token.

The desire to use keywords as well as legal SQL identifiers that would not otherwise be allowed in PDL (such as '#') has prompted the ability to escape special characters and words. In order to use a reserved word or character in a PDL identifier, it is necessary to precede the character with a '\'. For example:

```
object type StereoReceiver {
    ...
    String[1..1] serial\# = stereo_receiver.serial\#;
    String[0..1] \unique = stereo_receiver.unique_column VARCHAR(100);
    Component[0..1] \component = join stereo_receiver.component_id
                                   to components.component_id;
    ...
}
```

D.2.1.2. Reserved Words

```
Reserved Words
---------------------------------------
ADAPTER        = adapter
ADD            = add
AGGRESSIVE     = aggressive <WS> load
ALL            = all
ASSOCIATION    = association
ATTRIBUTES     = attributes
CALL           = do <WS> call
CLASS          = class
CLEAR          = clear
COMPONENT      = component
COMPOSITE      = composite
DATA_OPERATION = data <WS> operation
DELETE         = delete
DO             = do
EXTENDS        = extends
FALSE          = false
FOREIGNKEY     = foreign <WS> key
IMMEDIATE      = immediate
```

```
IMPORT         = import
INSERT         = insert
JOIN           = join
MAP            = map
MODEL          = model
OBJECTKEY      = object <WS> key
OBJECTTYPE     = object <WS> type
OPTION         = option
OPTIONS        = options
QUERY          = query
REFERENCEKEY   = reference <WS> key
REMOVE         = remove
RETRIEVE       = retrieve
RETURNS        = returns
SUPER          = super
TO             = to
TRUE           = true
UNIQUE         = unique
UNVERSIONED    = unversioned
UPDATE         = update
VERSIONED      = versioned
WS             = ([ , \t, \n, \r, \f])+

Punctuation
-----------
SEMI     = ;
EQ       = =
DOT      = .
COMMA    = ,
STAR     = *
LBRACE   = {
RBRACE   = }
LBRACKET = [
RBRACKET = ]
LPAREN   = (
RPAREN   = )
COLON    = :

Regular expressions
-------
ID        = (<ESC>|<CH>) (<ESC>|<CH>|<DIGIT>)*
CH        = [a-z, A-Z, _, $]
ESC       = "\\" ~[]
INT       = (<DIGIT>)+
DIGIT     = [0-9]
STRINGLIT = \" ( ~[\"] | \\\" )* \"

Special Tokens
--------------
COMMENT: // (~[\n, \r])* (\n | \r | \r\n)
```

- `add` — used with associations. The PDL file uses the `add` keyword, along with an association name, to signify that a given association should be inserted into the database.
- `call` — used to signify that a `Data Operation` contains PL/SQL and should use a `CallableStatement` to execute the block of SQL defined in the `do` block.
- `clear` — used with associations. The PDL file uses the `clear` keyword, along with an association name, to signify that all associations of the given name should be deleted from the database. This can be thought of as a "remove all" keyword.

- `component` — this word is reserved for future use.
- `composite` — used to signify a definition of a composite relationship between the current Object Type and another Object Type. This is used in the same fashion as the `association` keyword.
- `data operation` — the keyword used to signify the beginning of a SQL block that is used outside of an object type. It is used for any type of data modeling language (*DML*) operations, such as insert, update, delete, and PL/SQL procedure calls.
- `default` — this word is reserved for future use.
- `delete` — used to signify the beginning of the block of code defining the delete event (for example, how to insert an object of the defined type).
- `do` — used to signify the beginning of a block of SQL that should be executed by the persistence layer. This is typically used inside defined events.
- `dml` — the token used to represent the string `data operation`. This is used to signify that DML may be executed within the code block.
- `extends` — used to signify that the declared object type is a subclass of another object type. This is similar to the Java `extends` keyword.
- `false` — the boolean value false.
- `flexfields` — this word is reserved for future use.
- `foreign key` — used to signify a foreign key when defining the table schema. It is not currently used in PDL files, but will be added in the future.
- `import` — used to signify that another `ObjectType` will be used by the containing object type. For instance, for an `ObjectType` that extends `ACSObject`, the second line of the PDL file would be `import com.arsdigita.kernel.*;`.
- `insert` — used to signify the beginning of the block of code defining the "insert" event (for example, how to insert an object of the defined type).
- `join` — a keyword, used in association with `to`, to define the way to join two database tables together. In object types, sequences of `join-to` statements are defined to assist the metadata-driven SQL system in generating the standard queries.
- `map` — used to signify the beginning of the block that maps attribute names to specific database columns.
- `model` — used to define the namespace for the given object type. This is analogous to the package definition in Java.
- `object key` — used to uniquely identify a single object of the defined type. For instance, for an `ACSObject`, the `object key` is `id`.
- `object type` — used to define the name of the given object type. This is analogous to the `Class` definition in Java.
- `options` — used to signify that the following PDL block is a group of options for the given query or type.
- `query` — used to signify the beginning of a named SQL block that is used outside of the standard `create`, `retrieve`, `update`, and `delete` blocks. It is typically followed by a name (the name is used to identify it in the Java code) and then the block of SQL to execute and the mapping of the block to attribute names.
- `reference key` — the keyword used to specify the column that holds an object type's object id locally. In metadata-driven SQL, this information is used to join against the super object type tables.
- `remove` — used with associations. The PDL file uses the `remove` keyword, along with an association name, to signify that a given association should be deleted from the database.

- `retrieve` — used to signify the beginning of the block of code defining the `retrieve` event. One example of this would be a code block specifying how to retrieve an object of the defined type. It can also be used with the keyword `all` to retrieve all objects of the defined type.
- `returns` — used to signify that a definition returns something. One example is returns within a `Data Query` definition. In that case, it is used as an optional parameter in conjunction with two values to indicate the number of rows that may be returned by the given query.
- `super` — used to designate calling the same named method within the parent category. This can only be used when the `object type` extends another object type and is typically used within SQL event declarations.
- `to` — a keyword, used in association with `join`, to define the way to join two database tables together. In object types, sequences of `join-to` statements are defined to assist the metadata-driven SQL system in generating the standard queries.
- `true` — the boolean value true.
- `unique` — signifies that only one object type may have a given value for the attribute or combination of attributes. It also tells the DDL Generator to create a unique constraint.
- `unversioned` — means that the versioning service should not version this attribute.
- `update` — the keyword used to signify the beginning of the block of code defining the `update` event (for example, how to update an object of the defined type).
- `versioned` — when used in front of an object type, this keyword tells the versioning service to version all instances of this type; when used in front of a compound attribute, this keyword tells the versioning service to version this attribute, even though it would not normally be versioned otherwise.
- `White Space` — the token used to signify spaces. That is, " ", "\t", "\n", "\r", or "\f".

D.2.2. PDL Attribute Types

The PDL Attribute Types are the allowed Java types for attributes defined in PDL. Therefore, every type seen here is the Java type returned by the persistence system.

```
BigInteger
BigDecimal
Boolean
Byte
Character
Date
Double
Float
Integer
Long
Short
String // This should be used for both Strings and Clobs.
Blob   // This actually returns a Byte[] and will be deprecated once
       // the PDL compiler supports arrays.
```

D.3. PDL and SQL Used in the Tutorial

This section provides two examples, one each of SQL and PDL usage. These can be used as quick references to the context of every SQL and PDL example throughout the WAF Developer's Guide. The PDL usage is first in Example D-1, with the SQL usage second in the Example D-2.

```
model tutorial;

object type Publication {
    BigDecimal id = publications.publication_id INTEGER;
    String name = publications.name VARCHAR(400);

    object key (id);
}


object type Magazine extends Publication {
    // we need to specify the size of the String attribute so we know
    // whether it is actually a String or if it is really a Clob
    String issueNumber = magazines.issue_number VARCHAR(30);

    // notice that because it extends Publication, there is not an
    // explicitly "object key" declaration.  Rather, there is
    // a "reference key" declaration and "id" is not defined
    // as one of the attributes
    reference key (magazines.magazine_id);
}

object type Article {
    BigDecimal id = articles.title INTEGER;
    String title = articles.title VARCHAR(30);

    object key (articles.article_id);
}

// this is an "association block" associating "articles" and "magazines"
association {
   // note that the Attribute Type is an Object Type (Article)
   // and not a standard Java Type.  Also notice the order of the
   // join path and see the note below.
   Article[0..n] articles = join magazines.magazine_id
                              to magazine_article_map.magazine_id,
                            join magazine_article_map.article_id
                              to articles.article_id;
   Magazine[0..n] magazines = join articles.article_id
                                to magazine_article_map.article_id,
                              join magazine_article_map.magazine_id
                                to magazines.magazine_id;
   // the next line is the Link Attribute
   BigDecimal pageNumber = magazine_article_map.page_number INTEGER;
}


object type Paragraph {
    BigDecimal id = paragraphs.paragraph_id INTEGER;
    String text = paragraphs.text CLOB;

    object key (paragraphs.paragraph_id);
}

association {
   Article[1..1] articles = join paragraphs.article_id
                              to articles.article_id;
   // notice the composite keyword indicates that if the article does
   // not exist then the paragraph also does not exist
   composite Paragraph[0..n] paragraphs = join articles.article_id
                                            to paragraphs.article_id;
}
```

```
object type screenName {
   BigDecimal id = screen_names.name_id INTEGER;
   String screenName = screen_names.screen_name VARCHAR(700);
   Blob screenIcon = screen_names.screen_icon BLOB;

   object key (id);
}

object type Author {
    BigInteger[1..1] id = authors.author_id INTEGER;
    String[1..1] firstName = author.first_name VARCHAR(700);
    String[1..1] lastName = author.last_name VARCHAR(700);
    Blob[0..1] portrait = authors.portrait BLOB;
    // notice the use of a join path to allow the events for the
    // Role Reference to be automatically created.
    ScreenName[0..1] screenName =
         join authors.screen_name_id to screen_names.name_id;

   object key (id);
}


query paragraphMagazines {
   BigDecimal magazineID;
   BigDecimal paragraphID;
   Integer issueNumber;
   String text;
   do {
       select m.magazine_id, p.paragraph_id, issue_number, text
       from magazines m, a, magazine_article_map ma, paragraphs p
       where ma.magazine_id = m.magazine_id
       and p.article_id = ma.article_id
   } map {
       magazineID = m.magazine_id;
       paragraphID = p.paragraph_id;
       issueNumber = m.issue_number;
       text = p.text;
   }
}



query MagazineToAuthorMapping {
    // the next two lines are declaring that objects will be returned
    Magazine magazine;
    Author author;

    options {
        WRAP_QUERIES = false;
    }

    do {
        select publications.name, issue_number, publication_id,
               authors.first_name, authors.last_name, author_id
          from magazines, publications, articles, authors,
               magazine_article_map, article_author_map
         where publications.publication_id =
               magazines.magazine_id
           and magazine_article_map.magazine_id =
               magazines.magazine_id
           and magazine_article_map.article_id =
               article_author_map.article_id
           and article_author_map.author_id = authors.author_id
    } map {
        // here we map the attributes of the objects
        // to columns returned by the query.
        magazine.name = publications.name;
        magazine.issueNumber = magazines.issue_number;
        magazine.id = publications.publication_id;
        author.authorID = authors.author_id;
        author.firstName = authors.first_name;
        author.lastName = authors.last_name;
    }
}

query MagazineWithMaxID {
    BigDecimal magazineID;
    do {
       select max(magazine_id) as magazine_id from magazines
    } map {
       magazineID = magazines.magazine_id;
    }
}

data operation createMagazine {
   do {
      insert into magazine_article_map (magazine_id, article_id)
      select :magazineID, article_id from articles where not exists
      (select 1 from magazine_article_map
      where magazine_article_map.article_id = articles.article_id)
   }
}

data operation DataOperationWithPLSQLAndArgs {
    // the "call" keyword after the "do" indicates that the following
    // is actually a piece of PL/SQL.  The system then uses a
    // java.sql.CallableStatement to execute it instead of only a
    // java.sql.PreparedStatement.
    do call {
          myPLSQLProc(:title)
    }
}

data operation DataOperationProcWithInOut {
    do call {
        DataOperationProcWithInOut(:newID, :copiedID)
    } map {
        newID : INTEGER;
        copiedID : INTEGER;
    }
}

query DataOperationWithPLSQLAndArgsAndReturnInPDL {
    do call {
       :title = DataQueryPLSQLFunction(:articleID)
    } map {
        title : VARCHAR(700);
        articleID : Integer;
    }
}

query myDataQuery {
    BigDecimal articleID;
    options {
        WRAP_QUERIES = false;
    }
    do {
       select max(article_id) from articles
    } map {
       articleID = articles.article_id;
    }
}

query UsersGroups {
      String firstName;
      String lastName;
      String groupName;
      do{
        select *
           from users, groups, membership
          where users.user_id = membership.member_id
            and membership.group_id = groups.group_id
      } map {
        firstName=users.first_name;
        lastName=users.last_name;
        groupName=groups.group_name;
      }
}


query retrieveArticlesBasedOnAuthor {
      BigDecimal authorID;
      do {
          select article_id
            from authors, author_article_map
           where authors.author_id = author_article_map.author_id
             and lower(last_name) like :lastName || '%'
      } map {
          authorID = authors.author_id;
      }
}

query retrieveSelectedArticles {
      BigDecimal articleID;
      String title;
      do {
          select article_id, title from articles
      } map {
          articleID = articles.article_id;
          title = articles.title;
      }
}

query CategoryFamily {
  Integer level;
  BigDecimal categoryID;
  String name;
  String description;
  Boolean isEnabled;
  do {
     select l, c.category_id, c.name, c.description, c.enabled_p
        from (select level l, related_category_id
                 from (select related_category_id, category_id
                          from cat_category_category_map
                          where relation_type = :relationType)
                 connect by prior related_category_id = category_id
                 start with category_id = :categoryID) m,
            cat_categories c
        where c.category_id = m.related_category_id
  } map {
     level = m.l;
     categoryID = c.category_id;
     name = c.name;
     description = c.description;
     isEnabled = c.enabled_p;
  }
}
```

Example D-1. Usage Reference for PDL

```
create table publications (
    publication_id     integer
                       constraint publications_pub_id_nn
                       not null
                       constraint publications_pub_id_pk
                       primary key,
    name               varchar(400)
                       constraint publications_pub_id_nn
                       not null
);


create table magazines (
    magazine_id       integer
                      constraint magazines_magazine_id_fk
                      references publications
                      constraint magazines_magazine_id_pk
                      primary key,
    issue_number      varchar(30)
);

create table articles (
    article_id        integer
                      constraint articles_article_id_pk
                      primary key
                      constraint articles_article_id_nn
                      not null
    title             varchar(700)
                      constraint articles_title
);

create table magazine_article_map (
    magazine_id       integer
                      constraint mag_article_map_mag_id_nn
                      not null
                      constraint mag_article_map_mag_id_fk
                      references magazines,
    article_id        integer
                      constraint mag_article_map_article_id_fk
                      references articles
                      constraint mag_article_map_article_id_nn
                      not null,
    page_number       integer
);
```

```
create table paragraphs (
    paragraph_id      integer
                      constraint paragraphs_paragraph_id_pk
                      primary key
                      constraint paragraphs_paragraph_id_nn
                      not null,
    text              clob,
    article_id        integer
                      constraint paragraphs_article_id_fk
                      references articles
                      constraint paragraphs_article_id_nn
                      not null
);

create table authors (
    author_id         integer
                      constraint authors_author_id_nn
                      not null
                      constraint authors_author_id_pk
                      primary key,
    last_name              varchar(700)
                      constraint authors_name_nn
                      not null,
    first_name              varchar(700)
                      constraint authors_name_nn
                      not null,
    portrait          blob,
    screen_name_id    integer references screen_names
);

create table screen_names (
    name_id           integer primary key,
    screen_name       varchar(700) not null,
    screen_icon       blob
);

create or replace function myPLSQLProc(v_priority in integer)
as
begin
   insert into magazines (magazine_id, title)
   select nvl(max(magazine_id), 0) + 1, :title from magazine_id;
end;
/
show errors

create or replace procedure DataOperationProcWithInOut(
       v_new_id IN Integer,
       v_copied_id OUT Integer)
as
begin
   select max(article_id) into v_copied_id from articles;
   insert into articles (article_id, title)
       select v_new_id, title from articles
       where article_id = v_copied_id;
   insert into article_author_map (article_id, author_id)
       select v_new_id, author_id from article_author_map
       where article_id = v_copied_id;
end;
/
show errors

create or replace function
```

```
   DataQueryPLSQLFunction(v_article_id in integer)
return number
is
   v_title varchar(700);
begin
   select title into v_title from articles
    where article_id = v_article_id;
   return v_title;
end;
/
show errors
```

Example D-2. Usage Reference for SQL

Persistence Glossary

The terms used by the persistence system are largely based upon the *Unified Modeling Language* (UML) terminology. For background information, please consult the UML documentation at http://www.rational.com/uml/.

Association

Associations are bi-directional semantic connections. Put another way, they represent the relationship between two separate objects (UML Classes). Objects within an association are completely independent, though they may be linked together. Data may be passed in either or both directions, and more than one association may exist between two classes. In UML diagrams, associations are represented as links (lines between the two objects).

An example of an association is the relationship between a user and a group. From the level of object modeling, the two classes involved are User and Group and there is a many-to-many association between them. In this case, the association is one of membership. That is, a user "is a member of" a group.

See Also: Composite.

Attributes

Attributes are data fields that represent some property of the containing object that is shared by all instances of the object's class. Attributes normally have names (e.g., "Address") and Types (e.g., "String" or "Boolean"). An example of this would be the "Address" of a "User."

See Also: Property.

Composite

A Composite Relationship (also known as a "Composition") is a special kind of association in which one class cannot exist without another. That is, the object *strongly owns* its parts and if the object is deleted, its parts must also be deleted. The multiplicity at the parent object end must be 1..1 or 0..1. An example of a composite relationship is a key with a keyboard. The key is useless without the keyboard; if you remove the keyboard, you are also removing the key. Composite associations are useful for modeling relationships between objects where a contained object cannot exist outside its container object.

An example specific to the WAF is the composition between Parties and WAF Objects. A Party cannot exist without the corresponding WAF Object. If the WAF Object is deleted, the Party must also be deleted. The multiplicity holds because each Party has exactly one related WAF Object.

See Also: Association.

Derived Associations

Derived Associations are exactly like Associations, except that they have one or more levels of indirection. That is, if A is associated with B, and B is associated with C, then A has a Derived Association with C.

See Also: Association.

Data Object

A Data Object is any persistent entity within a relational database that is exposed to the applications as an object. An example is a row within the "users" table. It is important to realize that Data Objects are not the same as Domain Objects, and that they actually have a many-to-many relationship. See Section 9.2 *Beginning With Data Objects* and for more information.

See Also: Domain Object.

Domain Object

Domain objects are not part of the persistence layer, but since they are the most common way to access a Data Object (which is part of the persistence layer), it is useful to discuss them here. Domain objects contain logic particular to a business domain, such as managing users or credit cards. This logic operates over a persistent data object that contains the state necessary to implement the domain-specific logic. Domain objects contain the application logic specific to the domain that requires representing persistent state. It is important to realize that Domain Objects are not the same as Data Objects. Domain Objects and Data Objects have a many-to-many relationship with each other. See the Section 10.2 *Domain Objects Tutorial* for more information

See Also: Data Object.

Filter

`Filters` are used to allow developers to restrict the results returned by a Data Query or the results represented in a Data Association or Data Collection.

Join Element

A Join Element is a single part of the larger *Join Path*. It is used to indicate how two tables can be joined to each other. In the following example, there are 4 Join Elements that make up 2 Join Paths. The first Join Element is in the form `join foo` and the second one is in the form of `to bar`.

```
association {
   Article[0..n] articles = join magazines.magazine_id
               to magazine_article_map.magazine_id,
                          join magazine_article_map.article_id
               to articles.article_id;
   Magazine[0..n] magazines = join articles.article_id
                                to magazine_article_map.article_id,
                              join magazine_article_map.magazine_id
                                to magazines.magazine_id;
}
```

See Also: Join Path.

Join Path

A Join Path is the collection of *Join Elements* seperated by commas and indicate how to join tables to each other. They are typically used within associations to indicate how the objects within the database are related.

In the following example of an association between Articles and Magazines, there are two Join Paths; the first one is in *bold* and the second is in *italics*.

```
association {
   Article[0..n] articles = join magazines.magazine_id
     to magazine_article_map.magazine_id,
     join magazine_article_map.article_id
     to articles.article_id;
   Magazine[0..n] magazines = join articles.article_id
     to magazine_article_map.article_id,
     join magazine_article_map.magazine_id
     to magazines.magazine_id;
}
```

Link Attributes

A Link is the path from one object to another. Associations are types of links. A Link Attribute is a property of this link. For instance, if a developer wanted to know when the link was created, he could store the `creationDate` as a Link Attribute.

See Section 9.3.4 *Link Attributes* and Section 9.8 *Link Attributes*.

Link Role

See: Roles

Multiplicity

Multiplicity refers to a set of values (all non-negative integers, plus infinity) that can be used to describe cardinality. PDL uses a Multiplicity to specify the relationship between objects within Associations. Specifically, it can be used to answer the question, "given an object of type A, what is the range of number of objects of type B that can be related to A?" The syntax used is "[#..#]" where # is either "0", "1", or "n". The valid multiplicities are "[0..1]" (there may or may not be a single associated object), "[0..n]" (there may be zero to n associated objects), and "[1..1]" (there is always exactly one object related to the given object). The Multiplicities with an upper bound of "n" are typically used when there is a mapping table involved. The Multiplicity of "[1..1]" is typically used when there is a non-null, single column in the table referencing another table. If no multiplicity is specified, "[0..1]" is used as the default.

Property

Within the persistence system, Property is used as the global term to describe either an *attribute* or an *association* of an Object Type.

Reference Key

The Reference Key provides the Metadata Driven SQL system information about how to join the current Object Type with its super Object Type. In a standard Object Type that does not extend anything, an Object Key is declared to uniquely identify the object within the tables. If an Object Type extends another Object Type, a Reference Key is used instead of an Object Key. This Reference Key holds information about how the table of the child type can be joined with the table of the super type. For an example, see Section 9.2.6 *Object Type Inheritance*.

If the Object Key of the parent is keyed off of multiple rows, the columns in the reference key must appear in the order in which they should be joined. For instance, the definitions below will produce the `join` syntax that follows them (remove the "\" and make it all one line).

```
model tutorial;
object type FirstObjectType {
   <Attributes defined here>
   object key (object_id_one, object_id_two);
}

object type SecondObjectType extends FirstObjectType {
   <Attributes defined here>
   reference key (secondtable.second_object_id_one, \
secondtable.second_object_id_two);
}

The above PDL would produce SQL join code similar to the following:
```

```
<select code and tables here>
where secondtable.second_object_id_one = firsttable.object_id_one
  and secondtable.second_object_id_two = firsttable.object_id_two
```

See Also: Object Key.

Roles

Roles describe how one class is associated with another. One or both classes in the association may have roles. For example, a User has a "Member" role for the group (that is, a user is a member of a group).

See Also: Role Reference.

Role Reference

Role Reference is the term used to describe an object type's reference to another object type. It is similar to having an attribute with an Object Type modifier instead of a Java type. See Section 9.3.3 *Role References* and Section 9.3.5 *Using Java to Access Associations* for more information.

See Also: Property.

Metadata Driven SQL (MDSQL)

Metadata Driven SQL is the term used to describe SQL that is automatically generated by the persistence layer. The persistence layer is able to generate statements to perform insert, update, and delete, and it selects from the database if it is given enough information about the object (e.g., information about how attributes map to database columns and how tables can be joined together). In addition, it is able to use the object type metadata to generate DDL statements that can be used to create an update the actual table definitions.

Object Key

The Object Key allows the persistence system to uniquely identify a particular Data Object for a given Object Type. Specifically, the Object Key column(s) of an Object type is similar to the "primary key" of a database table.

If a particular object has a multi-key primary key, it should have a multi-key Object Key as well. To do this, you can simply separate the attributes by commas. For instance, for a mapping table whose primary key contains the columns `object_id` (whose attribute is `id`) and `site_node_id` (whose attribute is `siteNodeID`), the object key would look like `object key (id, siteNodeID)`. This is not a strong example, however, because in the common case mapping tables should not be represented as object types. Rather, they should be used in join paths when defining associations.

For more information, see the Section 9.2.3.3 *Object Key* section of the PDL tutorial.

See Also: Reference Key.

PDL

See: Persistence Definition Language (PDL)

Persistence Definition Language (PDL)

The Persistence Definition Language has been created to provide developers with a syntax to specify data objects, their associations, and how the information is stored in the database. The files specified in PDL and SQL are the only files that contain actual SQL and thus are the only files that must be changed when porting the system to a new database.

UML Class

A UML Class is a description of a set of objects that share the same attributes, operations, and relationships. A UML Class is almost identical to a Java class.

Index

Colophon

The Red Hat Applications manuals are written in DocBook SGML v4.1 format. The HTML and PDF formats are produced using custom DSSSL stylesheets and custom jade wrapper scripts. The DocBook SGML files are primarily written in **Emacs** with the help of PSGML mode; additional authoring and editing has been done with **vi** using macros and key mappings.

Garrett LeSage created the admonition graphics (note, tip, important, caution, and warning). They may be freely redistributed with the Red Hat documentation.

The Red Hat Applications Product Documentation Team is:

Karsten Wade — Primary Writer/Maintainer of Red Hat Applications documentation for Red Hat WAF and Red Hat CMS, for example the *Red Hat Web Application Framework Installation Guide* and the *Red Hat Web Application Framework Developer's Guide*

The Red Hat Applications Product Development Team made significant contributions to the constructing, authoring, and editing of the entire line of Red Hat Applications documentation.

- Archit Shah
- Bryan Che
- Dan Berrange
- Dennis Gregorovic
- Jon Orris
- Jim Parsons
- Justin Ross
- Rafael Schloming
- Richard Li
- Scott Seago
- Vadim Nasardinov

None of the Red Hat Applications documentation would be possible without the extremely able assistance of the Red Hat Documentation Technical Lead:

Tammy Fox — Primary Writer/Maintainer of the *Red Hat Enterprise Linux System Administration Guide*; Contributing Writer to the *Red Hat Enterprise Linux Step By Step Guide*; Writer/Maintainer of custom DocBook stylesheets and scripts