Sun™ ONE Application Framework Developer's Guide

Sun™ ONE Studio 5 update 1

Sun Microsystems, Inc.
www.sun.com

Part No. 817-4359-10
October 2003, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at `http://www.sun.com/patents` and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Forte, Java, NetBeans, iPlanet, docs.sun.com, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans le produit qui est décrit dans ce document.

En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à `http://www.sun.com/patents` et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats - Unis et dans les autres pays.

Ce produit est un document protege par un copyright et distribue avec des licenses qui est en restreignent l'utilisation, la copie, la distribution et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, parquelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Forte, Java, NetBeans, iPlanet, docs.sun.com, et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits protant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

UNIX est une marque enregistree aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.

Contents

Preface

This Sun™ ONE Application Framework *Overview* introduces the Sun ONE Application Framework and discusses what it is, how it works, and what sets it apart from other Web application frameworks.

How This Book Is Organized

Chapter 1, "Overview and Architecture" on page 13, provides an overview of the Sun ONE Application Framework architecture, and attention to how the various parts are combined to write a Sun ONE Application Framework application.

Chapter 2, "Develop an Application" on page 23, explains in detail the creation and usage of application components that can then be assembled into a complete functional application.

Chapter 3, "Programming Guide" on page 51, describes common programming scenarios and explains how to use certain fundamental objects in the Sun ONE Application Framework.

Chapter 4, "Deploy an Application" on page 85, covers the preparation of a Sun ONE Application Framework application for deployment in most J2EE containers, as well as deployment-time configuration of a Sun ONE Application Framework application.

Chapter A, "Troubleshooting" on page 101, outlines known Troubleshooting issues including symptom, probable cause, probable solution, and comments for each known issue.

Using UNIX Commands

This document might not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. See the following for this information:

- Software documentation that you received with your system
- Solaris™ operating environment documentation, which is at

  http://docs.sun.com

Related Documentation

Application	Title	Part Number
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Overview, Sun ONE Studio 5 update 1*	817-4360-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Tutorial, Sun™ ONE Studio 5 update 1*	817-4358-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework IDE Guide, Sun ONE Studio 5 update 1*	817-4104-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Component Author's Guide, Sun ONE Studio 5 update 1*	817-4362-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Component Reference Guide, Sun ONE Studio 5 update 1*	817-4661-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Tag Library Reference, Sun ONE Studio 5 update 1*	817-4361-10

Accessing Sun Documentation

You can view, print, or purchase a broad selection of Sun documentation, including localized versions, at:

http://www.sun.com/documentation

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

http://www.sun.com/service/contacting

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

http://www.sun.com/hwdocs/feedback

Include the title and part number of your document with your feedback:

Sun ONE Application Framework Developer's Guide, part number 817-4359-10

CHAPTER 1

Overview and Architecture

This Sun™ ONE Application Framework *Overview and Architecture* chapter provides an overview of the Sun ONE Application Framework architecture, and attention to how the various parts are combined to write a Sun ONE Application Framework application.

The name of the technology underlying the Sun ONE Application Framework is JATO. There are occasional references to JATO throughout this document, especially in class and other programmatic names. Consider the names Sun ONE Application Framework and JATO to be equal to one another.

Overview

What is the Sun ONE Application Framework?

The Sun ONE Application Framework is a mature, powerful, standards-based J2EE Web application framework geared toward enterprise Web application development. The Sun ONE Application Framework unites familiar concepts such as display fields, application events, component hierarchies, and a page-centric development approach, with a state-of-the-art design based on the Model-View-Controller and Service-to-Workers patterns.

Who Should Be Interested in the Sun ONE Application Framework?

The Sun ONE Application Framework is primarily intended to address the needs of J2EE developers building medium, large, and massive-scale Web applications. Although the Sun ONE Application Framework can be, and has been used for small Web applications, its primary advantages are not as readily apparent at that scale. The Sun ONE Application Framework especially shines when applications will be maintained for a long period, undergo many changes, and grow in their scope. In short, the Sun ONE Application Framework excels at helping develop enterprise applications.

Because the Sun ONE Application Framework provides core facilities for reusable components, it is well-suited to third party developers who want to provide off-the-shelf components that can be easily integrated into Web applications. These same features make the Sun ONE Application Framework suitable as a platform for building vertical Web offerings, particularly because these extension capabilities provide a well-defined way for both end users and original developers to extend and leverage existing vertical features.

What Does the Sun ONE Application Framework Do?

The Sun ONE Application Framework helps developers build enterprise Web applications using state-of-the-art J2EE design patterns. It provides a design-pattern-based skeleton upon which enterprise architects can hang other portions of their architectures. Web application developers find an easy development approach, and enterprise architects find a clearly delineated design that integrates in a well-defined way with other enterprise tiers and components.

The Sun ONE Application Framework helps developers build reusable components by providing both low- and high-level infrastructure and design patterns. Developer-defined components are first-class objects that interact with the Sun ONE Application Framework runtime as if they were native components. Components can be arbitrarily combined and reused throughout an application, across applications, and across projects and companies.

The Sun ONE Application Framework helps introduce new J2EE developers to Web application development, and empowers advanced J2EE developers by providing them a powerful toolkit with which to develop advanced features not possible with other frameworks.

What Doesn't the Sun ONE Application Framework Do?

The Sun ONE Application Framework is not an enterprise tier framework, meaning that it does not directly assist developers in creating EJBs, Web services, or other types of enterprise resources. Although the Sun ONE Application Framework is geared toward enterprise application development, it is properly a client of these enterprise tier resources, and thus provides a formal, first-class mechanism to access these resources.

Note – For a broader introduction to the Sun ONE Application Framework, see the *Sun ONE Application Framework Overview.*

The Three Tiers of the Sun ONE Application Framework Architecture

The Sun ONE Application Framework uses the time-tested Model-View-Controller, or MVC, pattern as one of its key architectural foundations. The original MVC pattern was developed to help developers write stateful client-side applications in Smalltalk. This pattern has been adapted for use in the J2EE Web tier, in a largely stateless manner. The Sun ONE Application Framework's MVC pattern is full-fledged and complete. This makes the Sun ONE Application Framework significantly different from other Web tier frameworks that advertise MVC compliance, but which actually use only parts of that pattern in significant ways.

The following points briefly explain the three parts of the MVC architecture, referred to in this document as tiers, and how they fit into the Sun ONE Application Framework:

Model Tier

In MVC terms, a Model is a presentation-neutral arbiter of data. This data can be tailored to support a given presentation, or tailored to represent application-specific data structures.

In the Sun ONE Application Framework, the bias is toward making Models that represent application data, with minimal dependence on the way the data is presented. Instead, Views are chosen that best match the format of the data.

Types of Models

There are a number of Model types in the Sun ONE Application Framework:

Model Type	Description
Model	The most general type of Model. All Model components ultimately must implement this minimal interface. This interface specifies the most fundamental Model behavior, which is the ability to get and set field values.
ContextualModel	A type of model that can hold data in named contexts. The definition of a context is Model-specific.
DatasetModel	A type of Model that contains distinct rows of data. DatasetModels can be positioned to a particular row via iterator-like methods.
MultiDatasetModel	A specialization of DatasetModel and similar to ContextualModel, this type allows access to named datasets.
ExecutingModel	A type of Model that can be executed to retrieve or update data in a backing store.
RetrievingModel	A sub-type of ExecutingModel that specifically supports a data retrieval operation. (Although this type might appear related to SQL, it is not SQL-specific.)
InsertingModel	A sub-type of ExecutingModel that specifically supports a data insertion operation. (Although this type might appear related to SQL, it is not SQL-specific.)
UpdatingModel	A sub-type of ExecutingModel that specifically supports a data updating operation. (Although this type might appear related to SQL, it is not SQL-specific.)
DeletingModel	A sub-type of ExecutingModel that specifically supports a data deletion operation. (Although this type might appear related to SQL, it is not SQL-specific.)
TreeModel	A Model that stores data in a generalized hierarchical structure rather than a Cartesian structure. Provides iteration over this data structure.
BeanAdapterModel	A Model that uses one or more JavaBeans as a backing data store.
ObjectAdapterModel	A Model that uses any Java object graph as a backing data store.
QueryModel	A JDBC -based Model that uses an RDBMS as a backing data store using SQL statements.
StoredProcModel	A JDBC-based Model that uses RDBMS stored procedures to get and set data.
WebServiceModel	A specialization of ObjectAdapterModel that uses a Web service via JAX-RPC to get and set data.
CustomModel	An abstract Model type that allows developers to implement their own data storage mechanism.
CustomTreeModel	An abstract Model type that allows developers to implement custom storage for hierarchical data.
SimpleCustomModel	A concrete Model that provides ready-to-use storage and dataset capability

View Tier

In MVC terms, a View is a presentation-specific way of displaying data from a Model. There is a relationship between a Model and View such that changes in a Model are automatically reflected in any Views attached to it, and changes in the View-presented data are automatically pushed back to associated Models.

The Sun ONE Application Framework uses the same basic definition of a View: a View presents Model data.

If you are familiar with writing a client-side application in Swing, Visual Basic, Delphi, or Powerbuilder, you will find Sun ONE Application Framework's notion of a View easy to understand. In these other environments, developers create frames, windows, and dialogs that contain child components. These child components are GUI widgets or containers for other GUI widgets, and they can be arbitrarily nested to any level.

Sun ONE Application Framework View components are almost exactly analogous. Just as you find various types of GUI widgets in a client-side application, such as display fields, panels, trees, and complex subcomponents, there are various types of Sun ONE Application Framework View components that fulfill these same roles. The Sun ONE Application Framework has specializations of Views that act as display fields, containers that can contain other Views, and a combination of these that can act as complex View components.

Types of Views

Objects that are Sun ONE Application Framework Views are actually objects that implement the `com.iplanet.jato.view.View` interface or one of its derived interfaces.

The following table shows the primary types of Views available in the Sun ONE Application Framework:

Name	Description
View	The most general type of View. All View components ultimately must implement this minimal interface. This interface specifies no specific View behavior. Therefore, you are unlikely to find useful instances of the View interface.
ContainerView	A type of View that can contain other Views. This interface adds methods specific to management of child View components.
TiledView	A special type of View that can present its child View components in a number of repeated tiles, or repeated regions. Examples of tiles might be rows or columns of a table, or tabs in a tabbed component. There is no assumption of tile layout made; simply the notion of repetition of tiles is encoded in this interface.

Name	Description
TreeView	A type of ContainerView that helps present information in a tree format.
ViewBean	A specialization of ContainerView that can serve as the top, or root, of a View component hierarchy.
DisplayField	A special type of View that has a value associated with it. The value of a display field can be presented on a page, and generally submitted back to the application by a user. There are a few special types of DisplayField, including BooleanDisplayField, ChoiceDisplayField, and CommandField.
CommandField	A special type of DisplayField that represents a user- or user-agent-activatable element in a rendered response. For example, in HTML, buttons and links would be represented in the server-side Sun ONE Application Framework application by CommandFields. When a CommandField is activated, a new request is sent to the server and a corresponding event or object is invoked on the server in response to that request.

Pages and ViewBeans

As explained above, the Sun ONE Application Framework embraces existing J2EE standards and technologies where possible and advantageous. One of the J2EE technologies useful to the Sun ONE Application Framework is Java Server Pages (JSP) technology. JSPs are natural for application developers to use, and are convenient to author and change quickly.

The Sun ONE Application Framework embraces JSPs as a primary way for application developers to create pages in their applications. However, there must be some glue between the existing JSP technology base and the Sun ONE Application Framework. ViewBeans fulfill this role, and are in some sense where JSP technology meets Sun ONE Application Framework technology.

The following sections explain the relationship of ViewBeans to JSPs and the Sun ONE Application Framework.

ViewBeans and Their Relationship to JSPs

If you have ever written a J2EE Web application using JSPs, you probably used what are commonly called helper beans to manage complexity in your JSP. Helper beans are commonly used with JSPs to keep Java code and complex data structures out of the JSP, where they are hard to maintain and debug. Instead, these complex features are put into the helper bean, and the bean is associated with and used by the JSP via a <*jsp:useBean*> tag.

Similarly, in a Sun ONE Application Framework application, developers want to avoid complexity in the JSP because of the inherent difficulties maintaining and debugging JSPs. Therefore, the Sun ONE Application Framework embraces the helper bean pattern and extends it to support the greater feature set required to make the framework functional and productive.

Specifically, the Sun ONE Application Framework uses ViewBeans to manage complexity in the JSP, and to provide a place for application developers to put application-oriented Java code. Like other helper beans, application developers can work with ViewBeans using traditional bean-oriented techniques, since ViewBeans are placed into JSP page scope and can be accessed just like any other helper bean. However, despite the name, ViewBeans are not exactly like helper beans.

The most significant difference is that (most) consumers of a ViewBean (for example, Sun ONE Application Framework JSP tags) do not use bean-like properties to access application data. Instead, Sun ONE Application Framework tags use the Sun ONE Application Framework-specific View API to interact with a hierarchy of View child components. In addition to the richer interactions made possible by this approach, it also represents a significant performance advantage, a clear advantage for enterprise applications that need to scale to massive levels.

ViewBeans and Their Relationship to Views

In addition to fulfilling a role similar to that of JSP helper beans, ViewBeans are special View components that function as root views in the Sun ONE Application Framework View component hierarchy. That is, they are top-level View components that contain other View components, and as such have no parent. In terms of Swing or other client-side development technologies, ViewBeans function like a window, frame, or dialog component.

The ViewBean interface extends from the ContainerView interface. This means that ViewBeans can contain other View components in a nested fashion, just like a root directory on a hard disk contains other directories. In addition to having all the behavior of a regular ContainerView, the ViewBean interface adds methods that are specific to its role as the root of a View component hierarchy.

Because of these attributes, ViewBeans can be thought of primarily as pages in your application. Just as your client-side application consists of a number of windows and dialogs, your Web-based application consists of a number of pages, and each of these pages has a ViewBean associated with it. When you count the number of Sun ONE Application Framework pages in your application, you will find the same number of ViewBean objects.

What makes a ViewBean special, as compared to a regular ContainerView?

- ViewBeans are the only View components that can be directly associated with a JSP (via the <*jato:useViewBean*> tag).

- Because ViewBeans are an abstraction of a single page in an application, they have logic and methods that manage page-specific functions, such as storage of page-specific application values.
- ViewBeans expose methods that allow application developers to initiate rendering a response to a client (they can serve as controllers of the request).
- ViewBeans are the only components in your application that are visible and can be accessed by application clients.

Pagelets and ContainerViews

As mentioned above, the View tier is generally comprised of an arbitrarily nested hierarchy of View components. Many of these components will be one of many types of ContainerViews, such as TiledViews or TreeViews, or other high-level components like data grids, input forms, headers and footers, and more. The general term for all these variations is pagelet, or a part of a page. Where ViewBeans are analogous to a page, ContainerViews are analogous to pagelets.

Pagelets might have their own JSP fragment associated with them, or might use their parent page (or pagelet) for rendering. When a pagelet uses a JSP fragment, instead of declaring a specific reliance on a type like a full JSP does, the association between a fragment and its pagelet peer is done by name. This allows a single pagelet to be rendered by several JSP fragments, dynamically, at the whim of the application. This allows for extremely flexible dynamic page generation.

Controller Tier

The Controller coordinates activity between Models and Views, and for the application as a whole. Whereas Model and View components can frequently be instances of off-the-shelf types, Controllers are typically custom written to an application.

In some J2EE architectures, the Controller role is fulfilled by a monolithic class, mapping, or other store of information using a highly centralized approach. Although this approach can be convenient for newcomers to visualize the application, it has serious drawbacks in terms of maintainability and the ability to compose an application from componentized pieces. Therefore, the Sun ONE Application Framework uses an inherently distributed approach in which the Controller role is further subdivided into Controller objects and logic units that are more closely related to a specific task or application component.

Unlike the View and Model tiers discussed above, there is no one single object or interface that corresponds to the Controller role in the MVC pattern in the Sun ONE Application Framework, at least not in a global sense. The reasons for this are generally described in the remainder of this document as various Sun ONE

Application Framework features are discussed. However, suffice to say at this point that the Sun ONE Application Framework adopts a distributed approach to fulfilling the Controller role in the pattern, and with this come a number of advantages, including the ability to create reusable components and to aggregate an application from components over time.

How Sun ONE Application Framework MVC Differs From Traditional MVC

Traditional (stateful, client-side) MVC architectures generally use a publish-and-subscribe event model to relate objects to one another. However, this approach requires stateful objects, an approach which in the Web tier generally will not scale to large numbers of users. Because the lifecycle of a request in the J2EE Web tier is well described and predictable, and because scalability is a primary requirement, the Sun ONE Application Framework avoids stateful server-side objects. This approach requires moving away from event listeners as a way to relate MVC objects to one another. The Sun ONE Application Framework maintains relationships between the tiers in MVC, but these relationships are not traditional event listener relationships.

CHAPTER 2

Develop an Application

This Sun™ ONE Application Framework *Develop an Application* chapter explains in detail the creation and usage of application components that can then be assembled into a complete functional application.

Create an Application

What is a Sun ONE Application Framework Application?

The Sun ONE Application Framework notion of application is essentially the same as the J2EE notion of a Web-tier application. In short, a Sun ONE Application Framework application is a J2EE Web-tier application with some Sun-ONE-Application-Framework-specific features. As such, Sun ONE Application Framework applications use the standard Web Application aRchive (WAR) structure as their basis. Therefore, when you create a Sun ONE Application Framework application, you are creating a Sun-ONE-Application -Framework-specific WAR file, and implicitly defining a single `ServletContext` for that application. Each Sun ONE Application Framework application has a single standard `web.xml` file, containing some Sun-ONE-Application-Framework-specific data.

Because a Sun ONE Application Framework application is in the end a J2EE application in a WAR file structure, you can use other J2EE features within the same application. For example, it is perfectly feasible and acceptable to use non-Sun-ONE-Application-Framework JSPs or servlets in the same WAR file. To a limited extent and as permitted by the J2EE specification, these non-Sun-ONE-Application-Framework application components can interoperate with Sun ONE Application

Framework application components. For more information, see the section "Interoperating With Sun ONE Application Framework Applications" on page 83 in Chapter 3, Programming Guide.

A Sun ONE Application Framework application is also a logical entity which describes a related set of components that interact to serve a specific function. In more concrete terms, an application is a set of pages, Views, Models, and application specific code that are deployed as a single unit as a single WAR file. Any changes to any of the parts of the application generally require the re-deployment of the entire application (as a WAR file, per the J2EE specification). The application WAR file will contain all relevant libraries, classes, and resources required for that application to function, with the notable exception of those libraries provided by the J2EE container.

Application-Level Entities

In addition to the Sun ONE Application Framework runtime library, `web.xml` file, and other assorted WAR file artifacts, each Sun ONE Application Framework application generally has an application servlet. This servlet serves as a base class for the various module servlets in the application (see below), and therefore provides an opportunity to perform application-level event handling, initialization, or other tasks related to each request.

Modules

Because real-world applications can become extremely large, and have different parts with slightly different functional requirements, the Sun ONE Application Framework adds an application sub-unit called a module. A Sun ONE Application Framework application consists of one or more Sun ONE Application Framework modules deployed as a single unit. All Sun ONE Application Framework applications must have at least one module.

Do not confuse the Sun ONE Application Framework usage of the term module with J2EE use of the term, which can be used as a synonym for a J2EE Web application as a whole. For example, you might see a J2EE Web application or WAR file described as a Web module. In this document, all references to application will refer specifically to a J2EE Web application or WAR file as a whole, and all uses of the module will refer to Sun-ONE-Application-Framework-specific modules, which have no analogue in J2EE.

In physical terms, a module is a single Java package under the application WAR file's `WEB-INF/classes` directory (the remainder of this section will refer to paths relative to this directory). A module package is an arbitrary package chosen by the developer, and can consist of any number of subdirectories. For example, both `main` and `com.mycompany.main` might be module packages.

Perhaps the most important distinction between a module package and any other arbitrary package is that only ViewBeans inside a module package will be accessible to clients of the application (see more below). This is accomplished primarily via a servlet mapping in the application's web.xml file. This mapping maps the module's module servlet (explained below) to a URL path. Generally, this URL path is a simple name and not fully qualified. For example, although a module package might actually be `com.mycompany.main`, to an external client, it might be identified only by the URL path `/main`. This reduces complexity for the users of the application, and reduces the length of URLs. The URL mapping need not share any similarity to the module package. In the example above, a URL path of `/foo` would also have been possible.

Clients can only request logical page names (not class names), which are automatically mapped to ViewBean class names by the module servlet. Therefore, it follows that only ViewBeans are directly accessible to external clients. For security and other reasons, all ViewBeans within a module package must exist directly within that package. ViewBeans within subpackages are not accessible (the subpackage must be declared as a module for those ViewBeans to become accessible), though they can be used in other ways by the application (for example, as subclasses).

Module Servlet

Each module has a single module servlet that handles all client requests for objects within that module. The module servlet acts as a front controller, intercepting all client requests and firing events as needed before dispatching the request to an application component. For example, the module servlet has events related to the beginning and end of a request, session initiation and timeout, and various error conditions. These events are described in the section on Handle Requests.

A module servlet generally derives from the application's application servlet. Because the application servlet is the superclass, the module can override or add behavior specified for the entire application as needed. For example, this becomes useful if a particular module has different requirements for handling certain events, such as session timeout. The module author can specialize the module's behavior as needed, yet still leverage common application-wide front controller behavior.

There is no strict requirement that a module servlet derive from the application's application servlet base class. It can instead just directly derive from the Sun ONE Application Framework's `com.iplanet.jato.ApplicationServletBase` class.

However, there is seldom need to avoid the derivation, and the Sun ONE Application Framework IDE toolset will automatically create module servlets that derive from the application servlet base class.

Package Structure

In general, a Sun ONE Application Framework application's WEB-INF/classes directory looks like the following:

```
/Base application package (may be multiple directories deep)
<Application servlet>
     <Other application classes/objects/files>
     /<module 1>
        <Module servlet>
        <Other module classes/objects/files>
     /<module 2>
        <Module servlet>
        <Other module classes/objects/files>
     ...
     /<module n>
        <Module servlet>
        <Other module classes/objects/files>
```

Each module is a self-contained sub-package of the overall application. Pages and objects in one module can interact with objects in other modules via *cross-module navigation* and/or simple class references. Cross-module navigation is covered later in this document.

Create a WAR File

Sun ONE Application Framework applications are packaged like any other J2EE Web-tier application in a standard Web Application Archive (WAR) file. Sun ONE Application Framework applications have certain assumptions as to where certain files will be located within the WAR file. The following is the recommended layout for Sun ONE Application Framework WAR files:

```
/[base application package]
    /[module 1]
        [Module JSP files]
    /[module 2]
        [Module JSP files]
    ...
    /[module n];
        [Module JSP files]
/[other static resources]
/WEB-INF
    web.xml (the application deployment descriptor)
    /classes
        /[base application package]
            [Application servlet class]
            [Other application classes/objects/files]
            /[module package 1]
                [Module servlet]
                [Other module classes/objects/files]
            /[module package 2]
                [Module servlet]
                [Other module classes/objects/files]
            ...
            /[module package n]
                [Module servlet]
                [Other module classes/objects/files]
    /lib
        [Sun ONE Application Framework runtime jar file]
        [other Sun ONE Application Framework component libraries]
        [other application jar/zip files]
    /tld
        /com_iplanet_jato
            jato.tld (the Sun ONE Application Framework tag library descriptor)
```

There are two parallel directory hierarchies—one for JSP files, and one for module classes. These two directory hierarchies should remain in parallel, although strictly, they need not. Specifically, each ViewBean is configured with the URL of its JSP peer during development, and this URL is arbitrary. However, to avoid naming collisions between modules which might be developed by different groups, the parallel directory structure illustrated above should be maintained.

Use Component Libraries

Although earlier versions of the Sun ONE Application Framework had rich support for certain types of application components, version 2.0 formalizes this support and extends it to most types of framework objects. Beginning with version 2.0, Views, Models, and Commands—as well as supporting classes—can all be designated components. With this designation, these objects become introspectable and manipulatable by the IDE toolset to allow for highly-productive visual development of a Sun ONE Application Framework application.

The Sun ONE Application Framework defines a basic set of components, and Sun Microsystems and other developers are expected to provide additional component libraries that significantly extend the features and power of the core framework. For information on creating your own reusable components and component libraries, see the *Sun ONE Application Framework Component Author's Guide*.

What is a Sun ONE Application Framework Component?

Since its inception, the Sun ONE Application Framework has supported a component model for certain types of objects. However, this component model relied on developers to learn each component's API and write code to use that component in his or her application. Although this level of functionality was sufficient and provided a significant productivity advantage over contemporary competitors (which had, and still have, no component models), Sun ONE Application Framework version 2.0 has significantly extended its component model to encompass all types of primary application objects (Views, Models, and Commands) and to allow rich IDE-based development of Sun ONE Application Framework applications.

In version 2.0 terms, a component is one of the various types of supported component classes in conjunction with metadata information. This metadata is encapsulated in a Sun-ONE-Application-Framework-specific class called a `ComponentInfo` class. At design-time, the development environment can inspect the ComponentInfo and present the component in an easy-to-use visual fashion.

The metadata stored in ComponentInfo classes is intended to enable automated use of the component in a development environment, such as the Sun ONE Studio. Developers can still manually create and use various types of components in their applications without defining a ComponentInfo class.

There are two types of Sun ONE Application Framework components. The first type is referred to as a distributable component. Distributable components are packaged in a component library and are generally deployed as part of a set of components. Distributable components have explicit ComponentInfo classes associated with them, and are specifically developed to be deployed for use by other developers.

By contrast, the second type of Sun ONE Application Framework component is referred to as an application-specific component, or a non-distributable component (do not confuse the term distributable in this context with the J2EE use of distributable to describe applications that run on multiple VMs; the reference is to distribution to other developers in this context). These components are only reusable within the application in which they are defined. They generally do *not* have explicit ComponentInfos associated with them, and exist as first-class objects in the current application. As an example, whenever you build a ContainerView or Model in your application, you are building an non-distributable component for use within that application. Because the IDE toolset knows how to manipulate these components directly, they are usable within the same application without any additional work by the developer. However, it generally takes some additional work to turn non-distributable components into distributable components (such as adding an explicit `ComponentInfo` class).

There is also another special classification of component, which is the extensible component. Extensible components are components that can be subclassed to create new types of components. Non-extensible components can have instances created of them, but cannot be extended to add new behavior. As of the current version of the Sun ONE Application Framework, all non-distributable components are derived from extensible components (though they are not themselves extensible components). Only certain distributable components are extensible components.

If all of this sounds confusing at first, don't worry. The IDE toolset will automatically manage these subtleties for you so you can just concentrate on building your application. As an application developer, you will generally never need to worry about whether a component is extensible or not, or whether it has a ComponentInfo class.

Sun ONE Application Framework Component Library

The Sun ONE Application Framework Component Library contains the core interfaces, runtime classes, and many basic components that you will use to create a Sun ONE Application Framework application. The component library is packaged as a single JAR file, and should appear in your application's `WEB-INF/lib` directory.

When creating a Sun ONE Application Framework application using the IDE toolset, the current version of the component library is automatically added to the application's `WEB-INF/lib` directory. If you open an application created in a previous version of the IDE toolset, you might be prompted to upgrade the application, including the Sun ONE Application Framework runtime library.

Other Component Libraries

In addition to the Sun ONE Application Framework Component Library, you might add additional component libraries to your application simply by placing them in your application's `WEB-INF/lib` directory. The IDE toolset will automatically recognize and mount the component library (it might take a minute or two), after which you will have the library's new components available for use within your application.

Unpacking of Tag Libraries

As part of a Sun ONE Application Framework component library, a library developer might provide one or more tag libraries to support rendering of the library's View components. Tag libraries are declared in the component library's component manifest file, and when the IDE toolset recognizes the component library, its tag library descriptors (`.tld` files) are automatically unpacked from the library JAR file for use by the application. In addition, the IDE toolset automatically adds tag library entries to the `web.xml` file.

Tag library descriptor files are unpacked to a special location under the application's `WEB-INF/tld` directory based on the name of the library to ensure that same-named files from different libraries do not conflict. In this scheme, library names are converted to directory names by replacing dots (".") with underscores ("_"). For example, the component library's internal library name is `com.iplanet.jato`, which is translated to `com_iplanet_jato` when unpacking the tag library descriptor. The Sun ONE Application Framework component library's tag descriptor file ultimately appears under the `WEB-INF/tld/com_iplanet_jato` directory in your application.

The tag descriptor's derived physical directory name is automatically registered to a logical resource name in the `web.xml` file for use by the application. This logical name is chosen by the component library author. In the Sun ONE Application Framework component library's case, the descriptor is registered as the resource `/WEB-INF/jato.tld`.

The tag descriptor unpacking mechanism makes use of timestamps to determine if an existing file should be overwritten when a new version of the library is added to an application. This feature ensures that upgrading of an application's component libraries is just a single step for a developer.

Unpacking of Additional Files

In addition to unpacking tag descriptor files automatically, the IDE toolset can also unpack any other additional files the library author wants when a component library is added to an application. These additional files can include JAR files, static HTML

files, image files, JSPs, or any other type of file. This is completely automatic when a new component library is added to an application in the IDE, but be sure to read each component library's documentation and keep an eye out for additional files that might appear suddenly in your application directory structure after adding a new library.

Create a Page (ViewBean)

As outlined above, a single logical page in your application generally consists of two parts: the ViewBean, and an associated JSP. The emphasis in the IDE toolset is on the ViewBean half, since it is a Java class and is a natural fit for the types of authoring tasks (such as configuration of components) required by the developer. Therefore, the JSP can be considered the subordinate half of a logical page object, and the following sections emphasize the tasks required to create a ViewBean first, and how to create and manage its set of JSPs second.

Create a ViewBean Class

Creating a ViewBean is straightforward. If you are using the IDE toolset, it is as simple as selecting New -> Sun ONE Application Framework (JATO) -> Page (ViewBean) in the IDE, and selecting a choice from the ViewBean list of the New View Wizard.

You must create ViewBeans in a Sun ONE Application Framework module package for them to be accessible to clients of your application. ViewBeans outside a module package can be used as superclasses for other ViewBeans inside module packages, but cannot themselves be accessed by clients.

When you create the ViewBean using the wizard, you will generally create a companion, or peer, JSP file to render that ViewBean. This JSP file will appear under your application's document root, and be automatically associated with your ViewBean by a *<jato:useViewBean>* tag declaration. By convention, the path to the JSP mirrors the package name of the ViewBean, so that a ViewBean placed in a `com.mycompany.main` package will have a JSP created in a `com/mycompany/main` subdirectory of the application document root. See more about a ViewBean's JSP associations in the sections below.

If you are creating a ViewBean manually, you can simply create a subclass of `com.iplanet.jato.view.BasicViewBean` or another ViewBean class in one of your application's module packages. However, a ViewBean created manually is not

manipulatable in the IDE as anything but a plain Java file—you have to manually add all components and event handlers to the ViewBean yourself. You also need to add and associate JSP files with the ViewBean manually.

Naming

Previous versions of the Sun ONE Application Framework used a strict naming convention to map request page names to ViewBean class names. This restriction has been removed beginning with version 2.1 and ViewBeans can be name just like any other class. For existing applications that need to maintain backward compatibility, there is a `web.xml` setting that can toggle the use of the strict convention. For more information, see Chapter 4, "Deploy an Application" on page 85.

By default, the IDE toolset creates applications that allow ViewBeans without the strict naming convention, meaning ViewBean classes can assume any name you want.

Code

If you double-click the ViewBean class in the IDE, or open it with a text editor, you see that the ViewBean generally has some preexisting code from its initial creation, plus some code in protected blocks that cannot be edited (in the IDE at least).

Do not use another editor to edit the protected code blocks in your ViewBean or other Sun ONE Application Framework object. Your changes will just be overwritten the next time you make a change to the class in the IDE. If you need to add code to a particular spot within the protected areas, either rethink what you are trying to do (chances are you don't really need to add code there), or use the Code Generation tab for the ViewBean or one of its child components to add code in a standard way.

You can generally add any other code or methods to the ViewBean that you want, or override some of its methods that do not have a visual IDE representation. Some of the more interesting methods you might want to override for advanced techniques are `getDisplayURL()`, `mapRequestParameters()`, `setRequestContext()`, `securityCheck()`, `beginChildDisplay()`, and `handleRequest()`.

Code within your ViewBean classes generally does not need to be thread-safe, as each request thread gets a private copy of a ViewBean. Also note that ViewBeans are request-scoped objects, so do not try to save data in a ViewBean instance between requests.

Manage JSPs

The most common technique for rendering a ViewBean as a response to a client request is to use what is called a peer JSP. This JSP contains custom Sun ONE Application Framework tags that associate it with its peer ViewBean. At runtime, the JSP and the ViewBean work together to render a dynamic response to the current request. This combination of JSP + ViewBean allows for powerful layout and content capabilities (the strengths of JSP technology) while keeping JSPs easy to maintain and code-free.

Each JSP can only be associated with a single ViewBean, but a ViewBean can be associated with many JSPs at the same time. These other JSPs can contain variations on the content and layout used to render the ViewBean; this is the parallel content feature discussed in the section "Display URLs and Parallel Content" on page 48. However, despite the presence of this feature, ViewBeans predominantly use only a single JSP in most applications.

JSPs have what is frequently referred to as a uses relationship with their peer ViewBeans. In other words, JSPs use their peer ViewBeans while rendering a response. ViewBeans, however, do not use their associated JSPs in the same fashion; that is, their relationship is not symmetrical. During rendering, the JSP pulls data from the ViewBean. The ViewBean is called by the JSP to provide this data, but it never calls the JSP for anything. The JSP is in the driver's seat when it comes to the rendering or display process.

For a JSP to be associated with a ViewBean, it must minimally have a valid *<jato:useViewBean>* tag declaration that calls out the ViewBean's class. At runtime, when the ViewBean is forwarded to display itself, it must select a JSP with a matching *<jato:useViewBean>* tag and return the JSP's URL relative to the application's document root as its display URL (via the `getDisplayURL()` and/or `getDefaultDisplayURL()` methods).

In the Sun ONE Application Framework IDE toolset, each ViewBean node has a sub-category node called *JSPs*. One or more JSP nodes can appear underneath this node, and each of these JSPs are associated with the current ViewBean via a *<jato:useViewBean>* tag declaration that calls out the ViewBean's class name.

Each of these associated JSPs is assumed to be managed by the ViewBean. *Managed* means that these JSPs should be automatically synchronized with the ViewBean as child View components are added. (This idea of managing a JSP is purely a design-time notion and has no runtime meaning.) More on this feature in the sections below.

The primary use to the developer for these JSP nodes is to allow him or her to easily open a ViewBean's JSP file, and assuming there is more than one managed JSP, to select a default for the ViewBean's display. The developer can right-click on any JSP node in the JSPs category and select the Set as Default JSP menu item to select that JSP as the ViewBean's current default. There can be only one default at a time.

Setting a particular JSP as the ViewBean's default results in the URL for that JSP being set as the ViewBean's default display URL, via the `setDefaultDisplayURL()` method. You might notice the generated code that calls this method in your ViewBean in one of the protected code blocks.

When a ViewBean is executed during runtime, the default JSP is used to render the ViewBean unless the developer has overridden the `ViewBean's getDisplayURL()` method to return a different value. Therefore, during development, you can usually switch between JSPs for alternate test runs by setting a different default JSP and recompiling the ViewBean.

Add Child View Components

A ViewBean is generally useless without adding some child View components to it to represent display fields on the page. Therefore, a major part of authoring a page is adding child View components to the ViewBean and configuring them to access Model data.

You can easily add a child View component to your ViewBean by right-clicking on its Visual Components node and selecting Add Visual Component... You will be presented with a component chooser dialog that shows you the currently available View Components in the application and each of its mounted application libraries. Alternatively, you can choose a component to add from the Component Palette.

Child View components basically come in two different types. The first type is Views that implement the `com.iplanet.jato.view.DisplayField` interface and effectively act as leaves on the tree of View components. DisplayFields usually represent a single primitive piece of data, such as a String or Integer, and frequently can be changed by the user and submitted back to the server. A good example of a DisplayField View component is the Basic Text Field component that comes with the Standard Component Library.

Occasionally, some DisplayField components are complex, meaning they also implement the ContainerView interface and have child View components themselves. Generally developers interact with these components as DisplayFields and have no idea that they are themselves compound components.

The various DisplayField components provided in the Standard Component Library all have the ability to be associated with a Model via a ModelReference and a Model field binding. For more information, see Chapter 3, Programming Guide, "Working with Values" on page 63.

The second type of View component is the ContainerView (or, alternatively, pagelet), which really encompasses a couple of subtypes including the TiledView and TreeView, and other types provided by third parties. ContainerViews are special types of Views that can contain other Views (including other ContainerViews), and they function as compound components. Through ContainerViews, View

components can be nested to any arbitrary level in what is commonly called a View component hierarchy. In client-side application terms, a ContainerView is the equivalent of a panel component, like JPanel in Swing. A TiledView component is roughly described as a table component (though it is not limited to that use), and a TreeView component is equivalent to a tree component like JTree in Swing.

Various types of ContainerViews might have custom properties and methods to allow developers to more easily interact with them, rather than needing to work with or understand the complex set of components they contain. ContainerViews are most frequently application-specific non-distributable components, meaning they are generally types within the current application that can be partially implemented or extended by the developer.

In terms of adding a ContainerView component to a ViewBean (or another ContainerView), there is no real difference from adding any other type of View component. ContainerViews expose properties that can be configured just like other View components. The one difference is that ContainerViews generally expose their inner complexity for purposes of interacting with the ViewBean's JSP. Specifically, developers can generally *see* children of ContainerViews and individually synchronize and lay out those components.

After a child View component has been added to the ViewBean or other ContainerView, the developer simply needs to configure it via its property sheet. As each property is filled out, the generated code in the ViewBean will be updated to configure the component at runtime. Also, as each child View component is added to the ViewBean, a matching JSP tag is added to the ViewBean's managed JSP(s). This tag will be minimally configured, and added with only basic regard to the static content layout in the JSP. The developer must add any additional desired attributes to this tag and arrange it to suit the page's layout. See the following section for more details.

Synchronize to the JSP

Just because a child View component is added to a ViewBean, it might not or should not necessarily be rendered when the ViewBean is displayed. For example, some View components should only appear on certain JSPs associated with a ViewBean, or might be used only by application code for their association with a Model field.

Each child View component must have a representation in a JSP associated with a ViewBean to be rendered with that ViewBean. This representation is normally a custom JSP tag provided by the component author. For example, when you add a Basic Text Field component to a ViewBean in the IDE, the toolset automatically adds a <*jato:textField*> tag to the JSP so that component will be rendered when the ViewBean is displayed.

Therefore, inherent in the notion of adding View components to a ViewBean is the idea of synchronizing one or more of the ViewBean's managed JSPs to the set of child View components. Each JSP basically has the ability to use only the fields that are relevant to it, so the developer needs a way to easily manage the JSP representation of each child View component.

The Synchronize to ViewBean... feature aims to make this JSP management easier. By right-clicking on a JSP node in a ViewBean and selecting this menu item, you open a dialog that you can use to select each of the child View components you want to appear in that particular JSP. When you press OK, the changes are made in the JSP—deleted component tags are removed, and inserted component tags are added. You can then open the JSP file itself and arrange these tags to work within your JSP's overall layout (inserted tags are added near the end of the file).

This feature is simply a convenience. You can manually synchronize a ViewBean to a JSP by adding, editing, or removing tags in the JSP file. In fact, some View components might not support automated synchronization, and will require manual synchronization to appear in the rendered output.

JSP tags are added to a JSP based on the content type of the JSP. Developers have the ability to select a tag template for each content type their component supports, so depending on the dialect of your JSP, you might see different JSP tags for the same View component when using different content types. Pay attention to the features each set of tags supports, as they might also differ by content type

Execute a Page from the IDE

Generally, when writing a Sun ONE Application Framework application, you want to be able to run the current page to see how it is working and to test its JSP's layout. Instead of manually packaging the application, deploying it in your container, then opening a browser to the correct URL, you can do this all in one step by simply executing the ViewBean.

First, make sure your application has been fully compiled at least once by selecting Build All... from the application node. (All the files in your current module will be compiled for you when you execute the ViewBean.) Right-click on the desired ViewBean and select the Execute Page (Redeploy) menu item or toolbar button. Depending on which J2EE server you have selected as your default for Web applications (see Sun ONE Studio help), this will automatically deploy your application and load a browser with the correct URL for that ViewBean. If you have made any changes to application code, you *must* execute the page with the Redeploy option, or your changes will not be reflected in the application. The only exception to this is changes to the JSP, which are automatically detected.

You can only execute ViewBeans in a Sun ONE Application Framework module. Remember, ViewBeans outside a module cannot be accessed by application clients.

Create Pagelet (ContainerView) Components

This section describes creating View components from extensible components contained within a component library. Extensible components are components that are meant to be extended (subclassed) to create new types of components. After a new component type has been defined (using the extensible component as the basis), many instances of it might be used within a given application.

Create a ContainerView Class

There is really nothing different about creating a ContainerViews class; the process is analogous to creating a ViewBean class. The only difference is that ContainerViews, like all pagelets, cannot be executed on their own. They must be designated as children of another component and placed inside a page to be run.

One other difference is that ContainerViews and the other types of pagelets might opt not to create a JSP fragment, and instead delegate all rendering to its parent component. This means that custom tags that represent the pagelet and its children are placed in the parent's JSP (or its parent if it does not have one of its own, and so on) instead of in a standalone JSP fragment.

Primary Models

Many types of ContainerViews use the notion of a primary model, or a model that drives their rendering. For example, TiledView implementations generally require a primary model of type DatasetModel to proceed from tile to tile, and to know when to stop iterating. In the same way, TreeView implementations also require a primary model of type `TreeModel` to display a hierarchical view of that data.

Handle Requests

Handling and responding to client requests is inarguably the reason Sun ONE Application Framework applications exist. This section outlines the various features Sun ONE Application Framework provides for handling and responding to client requests.

Request Lifecycle

In general, each request consists of two parts: the submit phase and the display phase.

When a user accesses a page in a Sun ONE Application Framework, he or she generates an HTTP request to the server where the application is running. This client request is initially handled by the application's module servlet, which acts as the front controller for a particular application module. The module servlet fires various request lifecycle events (for example, session timeout detection) and then determines a ViewBean class to invoke based on the information submitted from the client. The ViewBean class the servlet invokes is normally the same class that was responsible for generating the user's previous response. In this way, transitions between pages (ViewBeans) occur in a Sun ONE Application Framework on the server only, as directed by the application developer.

When a page is asked to respond to a request, its ViewBean's `invokeRequestHandler()` method is called by the module servlet. That is the first point of entry to the application proper. This begins the submit phase. During the submit phase, submitted values are mapped to View components and Models, then an event handler is invoked to process the request and prepare a result. The submit phase generally ends when the application (or the framework) calls the `forwardTo()` method on the same or another ViewBean from inside an event handler, thus beginning the display phase.

During the display phase, a number of things happen, the most important of which is that the JSP associated with the ViewBean is rendered. This causes a callback to the associated ViewBean's `beginDisplay()` method. The ViewBean and all its children then render as the JSP is processed, finally completing with the `endDisplay()` method being called on the ViewBean. The request is not yet complete, however, as the `forwardTo()` method returns to whatever code called it. When that code completes, the call stack unwinds and the response is sent to the client.

Just before the call stack unwinds, the module servlet fires more request lifecycle events, and any registered RequestCompletionListeners are called so that they can do things like add last-minute information to the session (but not affect the response in any way). This completes the request.

Front Controller Events

Before each request is dispatched to an individual RequestHandler (the ViewBean or one of its children), it is processed by the front controller. In concrete terms, the front controller is the module servlet. The module servlet normally derives from the application's application servlet, which normally derives from `com.iplanet.jato.ApplicationServletBase` (it is possible that there could be

some variation from this scheme, but ultimately, all Sun ONE Application Framework servlets *must* derive from `ApplicationServletBase`). ApplicationServletBase fires a number of events during request processing. Developers can respond to these events by overriding the event callback methods in the application or module servlet. These events offer the opportunity to consolidate general request processing logic in a single location.

The following table lists events fired by the module servlet (more information is available in the JavaDoc for `com.iplanet.jato.ApplicationServletBase`):

Event Method	Description
`onAfterRequest`	Fired after the current request is complete. Can be used to clean up request-specific resources or finalize data in the session.
`onBeforeRequest`	Fired when a new request is received. Can be used to verify some portion of the request (such as a user principal parameter), or redirect the request upon some condition.
`onInitializeHandler`	Fired to allow application-specific initialization of a RequestHandler instance (normally a ViewBean). Can be used to handle common initialization of ViewBeans, perhaps based on some request attribute.
`onNewSession`	Fired when a request without a session is received and a new session is created for the client. Can be used to prepopulate the user session with data.
`onRequestHandlerNotFound`	Fired when the requested RequestHandler (i.e. ViewBean) was not found. Typically used to respond to.
`onRequestHandlerNotSpecified`	Fired when a RequestHandler (i.e. ViewBean) was not specified in the request
`onSessionTimeout`	Fired when a request with an expired session ID is received. Typically used to reinitialize the client's session or redirect to a login page.
`onUncaughtException`	Fired when an unexpected exception occurs during request processing. Typically used to present an error page to the client.

To use these events, application developers simply override these event methods and perform whatever logic is required. The most common behavior is to redirect the request to a different page than that which was originally requested by the client. This is common for error or session timeout conditions. However, these events are fired as request processing occurs, and they provide no specific mechanism to communicate to the main request processing logic that normal processing should stop.

Therefore, there is a technique commonly used in these circumstances. The Sun ONE Application Framework provides a special exception class, `com.iplanet.jato.CompleteRequestException`, that can be thrown anywhere during request processing, without actually causing an error condition to be raised to the client. The common use of this exception is to redirect processing of a request

to another ViewBean, or perhaps by sending a result directly back to the client using the Servlet API, and then throw a `CompleteRequestException` so that the original request stops in its tracks. This exception essentially tells the request processing infrastructure of the Sun ONE Application Framework that the developer has handled the response already and further processing is not necessary. The following is an example of using this technique to redirect a request to a different ViewBean in the case of a session timeout:

```
public void onSessionTimeout(RequestContext requestContext)
{
    // Obtain a ViewBean in the application
    ViewBean viewBean=
requestContext.getViewBeanManager().getViewBean(SessionTimeoutViewBean.class);
    // Forward the request to a ViewBean
    viewBean.forwardTo(requestContext);

    // Abort normal request processing
    throw new CompleteRequestException();
}
```

If you neglect to throw the `CompleteRequestException` in a scenario in which the response has already been handled, you will generally either get a J2EE or application error, or cause the first response to be completely ignored by the container, thereby defeating the purpose of handling the event.

Unlike other places you place code in your application, code in the module or application servlet must be threadsafe. Therefore, you must avoid using member variables (static or otherwise) to store request-scoped data in the servlet.

Application Events

After the front controller has fired various events for a request and determined which RequestHandler (for example, ViewBean) should dispatch the request, the ViewBean is invoked to complete the handling of the request. Specifically, the ViewBean's responsibility is to determine which of its subcomponents is responsible for handling the request, and to invoke that component to actually handle the event. This process is called request dispositioning.

An argument can be made at this point that the logical Controller role is at least partially filled by the ViewBean, which might seem confusing to some because it has been stated that the ViewBean is (also) a View component. This is part of what makes a ViewBean special, its dual nature as both a View and partial Controller of the request. This situation is actually quite intuitive—much like a window or dialog

object in a client-side application determines which GUI widgest contained within it should receive a mouse click event, the ViewBean is responsible for determining which View component within it should respond to a request event.

This approach follows naturally from the fact that the Sun ONE Application Framework uses a component model to provide aggregation of an application from components. Just like Swing visual components have logic to help them determine which of their child components was clicked, so the Sun ONE Application Framework has similar logic. Without this ability, it would be impossible to create View components that can be assembled and wired together to create a View hierarchy.

During request dispositioning, the ViewBean and each of its contained ContainerViews in turn examine the `javax.servlet.http.HttpServletRequest` object to determine if it is the component responsible for handling this request. Once a responsible component is found, the request is dispatched to it, which generally causes a business logic event to be fired.

This abstract description might leave you wondering what is meant by the term responsible component. The responsible component is the first View component that implements `com.iplanet.jato.RequestHandler` to return a non-null result from the `acceptRequest()` method when it is called during request processing. A component normally decides whether to return an object from this method by examining the request parameters and determining if they contain a name-value pair that corresponds to an event that should be handled by this component. The details of this process are beyond the scope of this document, but the most important part to understand is that CommandFields are the key decision makers in this process. CommandFields is explained in the next section.

CommandFields

`CommandFields` are special types of Views that correspond to a component on a page that can be activated by the client to initiate a request to the server. Although any type of View could theoretically implement the `CommandField` interface, this wouldn't make much sense. Instead, only certain types of Views, such as button display fields, should implement this interface. For example, in HTML terms, both buttons (<*input type="submit"*>) and links (<*a href="..."*>) are represented in the application by CommandFields in the Sun ONE Application Framework, because these are components that users click (activate) and cause another request to be sent to the server.

CommandFields can be added to a ViewBean or other container just like any other View component, and the Sun ONE Application Framework component library contains both button and HREF/link CommandField components. The difference is

that the framework understands that these types of components are special and therefore they are treated in a specific way. Specifically, they are examined during request dispositioning to determine if they were the source of the current request.

The end result of this process is that, for example, when a user clicks a button on a form in an HTML application, the result is that the button component in the application is called upon to determine how to handle that request. The button component has special properties that tell the parent component what to do to invoke business logic for that request. Specifically, an activated button or other CommandField specifies a developer-defined Command object that should be invoked for the current request. Alternatively, a CommandField can specify that a default Command object be invoked to handle the current request (by not specifying an alternative).

Command Descriptor Property

The developer specifies which Command object will be invoked when the field is activated via the field's Request Handler property. A CommandDescriptor is a Sun ONE Application Framework object that encodes the information needed at runtime to construct a particular instance of a Command class and invoke it. Minimally, the CommandDescriptor specifies which Command class should be instantiated at runtime. (the Command class is only constructed if the corresponding CommandField is activated.)

By *omitting* a value for a field's Request Handler property, the developer implicitly indicates to the framework that it should use a default Command object to handle the request.

This choice of Command objects leads to two alternate ways of handling a Sun ONE Application Framework business logic event: via a request event handler method on the parent of the CommandField (if the default Command is used), or via a developer-defined Command object. Both alternatives are explained in the following sections.

Request Event Method Handlers

The most common approach for handling request events is to use a request event method handler. The developer implements a method with a certain naming convention and signature that will automatically be called when the corresponding CommandField is activated. This is much analogous to implementing an event handler method for a client-side application.

The advantages of this approach are that it is easy to understand and maintain, keeps code localized to a relevant object, and allows the developer to write simple procedural logic. The disadvantage of this approach is that this event logic is not reusable between different fields or pages, though of course these methods can call common methods that are reusable in this fashion.

In the Sun ONE Application Framework, this event handler method is implemented in the *parent* of the responsible CommandField. This is more intuitive than it might sound. For example, if you create a new ViewBean and add a button component to it as a child, you would implement the request event handler method in the parent of the button component, the ViewBean.

You could not implement this method anywhere else, because you are creating an *instance* of the button component within the container—you cannot add a new method to an instance! This fact underscores the fact that there is a close relationship between a ContainerView *type* and its child component *instances*, and that both the container and its children work together to form a functional component.

The signature of the request event handling method is the following, where <childName> is the local (unqualified) name of the child component:

```
public void handle<childName>Request(RequestInvocationEvent
event)
```

For example, in the example above, if you added a button component named *submit* to your ViewBean, you would need to implement the following method in your ViewBean in to handle events from the button:

```
public void handleSubmitRequest(RequestInvocationEvent event)
```

(The name of the child component is automatically uppercased as needed to create a conventional method name.)

Although you can add this method to a ContainerView class manually, the advantage of using the Sun ONE Application Framework module for Sun ONE Studio is that you can simply right-click and choose the handleRequest event from the popup-menu. This adds the appropriate method stub to your class for you.

Important: This style of event handling is only used if the developer does *not* specify a value for the CommandField's Request Handler property. If the developer specifies a descriptor for a CommandField, the framework assumes that the Command object specified in the descriptor will handle the request completely. The IDE might still allow you to add an event handler stub to your View, but it will not be invoked if a descriptor is specified!

See the sections below for information on how to actually implement business logic in your event handler methods.

Command Event Handlers

The alternative to a request event handler method is to implement a Command object directly, and associate it with a CommandField directly by supplying a value for the field's Request Handler property.

The advantages of this technique are that it allows reuse of application-specific or library-provided behavior, provides point-and-click application assembly, and to some, keeps View components *cleaner* by eliminating event handler methods. The disadvantages of this approach are that it causes a proliferation of Command classes (potentially one for every CommandField instance), it decouples event handling logic from objects that generate those events (less encapsulation of component behavior), requires more up-front design, and might require more code to accomplish the same task as a request event method handler.

To use this approach, developers simply create a Java class that implements the `com.iplanet.jato.command.Command` interface (by hand or by using the wizard in the Sun ONE Application Framework toolset). Once this work is done, the Command is available to be specified in the Command Class property of a CommandField's Request Handler property.

See the sections below for information on how to actually implement business logic in your Command objects.

Which Event Handling Approach Should I Use?

A general recommendation on deciding which request event handling approach to use is to be practical about which techniques solves the application requirements. It has been found that most applications will actually use both styles.

In general, when you want to encapsulate event handling logic within a component, are creating a component for reuse, or simply want to create a functional object quickly without worrying about abstracting every detail of the event handling, request event handler methods might be the more appropriate choice.

If, by contrast, you want to abstract away the details of event handling into common classes, need reusability of complex logic, or want to provide pre-built request event handling logic that can easily be assembled into an application, use Command objects.

Another thing to consider is that, in general, Command objects take more work up front to create, but might pay off later in terms of reusability. Request event handler methods are quick, easy, and familiar to many application developers that have used Swing or other client-side application development technologies, but might require more care to make important logic reusable.

In the end, there are no absolute rules, and no need to decide one way or another, even within the same object. The Sun ONE Application Framework gives you the flexibility to decide on a per-field basis which approach works best, and developers are encouraged to be open to the best approach for the job at hand.

Write Event Handling Logic

Regardless of how you elect to handle a request event, using either an event handler method or a Command object, you must write some logic to have your application do something meaningful. In general, your event handling logic will follow a routine pattern:

1. **Execute business logic**

   a. **Obtain values submitted by the user**

   b. **Process these values**

   c. **Prepare objects with results and for display**

2. **Render a response to the client**

Event handling logic is typically procedural in nature—each request is primarily handled by a single method body, though this logic might make references to any objects or methods it wants (just like any other Java code).

Forward References

Sun ONE Application Framework objects such as ViewBeans can be forward-referenced within event handling logic, to allow configuration of these objects for the forthcoming display cycle. For example, it is possible for a developer to obtain a value from a field on the current page and then set this value on a different field on a different page within his event handling logic. This technique should seem familiar to developers who have used client-side application development technologies, where GUI widgets are stateful and can be referenced from more or less anywhere in the application at any time.

Business Logic

With exception of the use of the Sun ONE Application Framework API, it is assumed that most developers are already familiar with the general concept of writing application business logic. Because the use of the Sun ONE Application Framework

API is covered in Chapter 3, "Programming Guide" on page 51, this skips to the discussion of how the developers reply to a client request after their business logic executes.

In addition to using the Sun ONE Application Framework API, developers have full freedom to use the J2EE APIs provided by their container from within their event handling logic. For example, it is perfectly feasible and acceptable to use Servlet, JDBC, JNDI, EJB, JavaMail, or other J2EE technologies or APIs within the scope of a Sun ONE Application Framework event handler. However, in some cases, the Sun ONE Application Framework provides features that make using these APIs considerably easier. Some of these capabilities are described in Chapter 3, "Programming Guide" on page 51.

Render a Response

After executing business logic in response to a request, the developer must determine the response that should be sent to the client, and depending on the desired technique, actually render this response. You will always be sending some kind of response to a client request, be it another page or an error message. In most cases, the response will be another page that follows logically from the executed request. For example, if the user clicks a button to add an item to his shopping cart, the response would probably be to show a page with the details of the user's shopping cart.

Every request comes to a crucial point in which it transitions from focusing on the request, and begins to focus on sending a response to the request. In Sun ONE Application Framework terms, this turning point in the request lifecycle is commonly referred to as the switch from the submit cycle to the display cycle. During the submit cycle, submitted values are mapped to View components and Models, and an event handler is invoked to process the results and prepare a result. During the display cycle, the result is rendered, or displayed, to the client.

Pageflow

The ubiquitous thin-client style of chaining pages together is frequently called pageflow. The user *flows* from one page to another by invoking an action on the first page, which sends a request to the server and results in a response page being sent back to the client.

There are many approaches to addressing pageflow. Some frameworks take the position that pageflow is a first-class requirement that should be directly addressed by a prospective framework. For example, these frameworks might specify that business logic return a JSP URL that should be used to render the response. A common problem with these approaches is that the response rendering mechanism then becomes very brittle—it is hard to send a response using anything except this

mechanism. If this mechanism assumes a JSP or URL will be invoked to render a response, it becomes hard to satisfy certain application requirements that are at odds with this approach. For example, some application requests might result in sending back a binary response (such as a PDF file). These scenarios are difficult to solve if the framework assumes that the application cannot directly respond to the client.

By contrast, the Sun ONE Application Framework enables an easy pageflow mechanism, but leaves the decision of how to render a response in the hands of the developer. Specifically, Sun ONE Application Framework developers have the ability to easily forward to or include any ViewBean in the response, forward to or include any arbitrary URL, or send a result back directly to a client using the standard Servlet API.

Display a ViewBean

The most common approach to rendering a result is to forward the request to another ViewBean (this terminology derives from the underlying J2EE Servlet feature which allows forwarding of a request to response URL). In Sun ONE Application Framework terms, this is frequently referred to as displaying the ViewBean. Because ViewBeans generally have a URL for a peer resource such as a JSP associated with them, the framework can handle the details of actually invoking the appropriate JSP for a ViewBean if it simply knows which ViewBean should be used to display a response.

Therefore, after executing business logic for a given request, the developer might simply obtain a reference to the desired target ViewBean using the ViewBeanManager, and call the ViewBean's `forwardTo()` method to cause that ViewBean to be rendered as a response to the client:

```
public void handleSubmitRequest(RequestInvocationEvent event)
{
    // Do business logic here (this is still the submit cycle)
    ...

    // Display a ViewBean as a result
    ViewBean targetViewBean=
        event.getRequestContext().getViewBeanManager().getViewBean(
            ResponseViewBean.class);
    targetViewBean.forwardTo(event.getRequestContext()); // Start display cycle

    // Finish up (this occurs during the display cycle)
    ...
}
```

As you can see, the transition between the submit cycle and the display cycle is triggered by the developer's call to `forwardTo()`. This represents a significant opportunity—the developer could trigger this call at any point during his event handler. Furthermore, the `forwardTo()` method eventually returns, after which the developer has the opportunity to do some additional work during the display cycle (however, this is fairly uncommon).

Despite the emphasis on the submit and display phases, these phases are essentially logical in nature—they are just aspects of a single request to the application.

Display URLs and Parallel Content

So, how does a ViewBean know which JSP it should use to be displayed? There are two methods that allow the ViewBean to specify which URL should be used to render it when `forwardTo()` is called.

The first method is `ViewBean.getDisplayURL()`. This method returns a String containing the URL that should be forwarded to using the Servlet API RequestDispatcher mechanism. (this URL need not call out a JSP; it could instead call out a static HTML page (if it did not need to dynamically display data) or a page written using another (non-JSP) dynamic rendering mechanism. However, in practice, the majority of display URLs will refer to JSPs as these are the readily available J2EE mechanism.)

The value of the display URL is solely under the discretion of the ViewBean. For example, a ViewBean might use some information in the request, such as the client's locale, to decide between two different URLs, one in Spanish and another in French. In the core BasicViewBean implementation, if the developer does not provide alternative logic for the `getDisplayURL()` method, the method will simply return the value of the `getDefaultDisplayURL()` method. This value can be set by the developer in code, or automatically in the IDE by simply selecting a JSP under a ViewBean node, right-clicking, and selecting the Set Default option.

This ability for the ViewBean to decide on a particular URL from a set of URLs is the Sun ONE Application Framework's parallel content feature. Parallel content refers to the fact that developers frequently want to render the same dynamic data in several different ways. Using a single JSP, there is no way to accomplish this without using heavy conditionality or pulling static content out of the JSP. These approaches usually result in productivity and maintenance problems. Therefore, the Sun ONE Application Framework provides the ability to have multiple JSPs (or other URLs) associated with the same ViewBean. This allows the developer to provide *parallel* pages that present essentially the same dynamic data, but perhaps in radically different ways.

One use of this feature is to create pages that contain static content in different languages. For example, it would be difficult to create a single JSP that could present a ViewBean data in both English and Japanese. Not only are the character sets and

languages extremely different, but the layout of the Japanese page will differ considerably from its English counterpart. Instead of writing a single, highly complex JSP, the developer can instead write two much simpler JSPs. Each one will refer to the same ViewBean, but will present the data in different ways. The ViewBean can make a decision on which JSP to use to display itself by examining the locale or the user's language preference and return the appropriate URL from its `getDisplayURL()` method.

Another use for parallel content is to target different device types. For example, a ViewBean could be rendered as either an HTML document or a WML document. It is not possible to create a common JSP to render both because the markup languages have incompatible differences. Instead, a developer can create JSPs in each markup language and allow the ViewBean to decide which to use. Again, the ViewBean would use some information to decide which URL to return from its `getDisplayURL()` method.

It would be remiss to imply that this decision making machinery is already built and ready to use in the core `BasicViewBean` class, when in fact it is not. There is a good reason for this: it is not possible for the Sun ONE Application Framework to know the range of application-specific variables that might play into the decision to display a particular URL over another. Furthermore, the URL scheme used to differentiate different versions of a JSP (say, English versus Japanese) is a decision the application developer must make. For these and other reasons, being prescriptive has been avoided in this area.

Therefore, when parallel content is required, the `getDispayURL()` method should be overridden by developers to supply the requisite decision logic. Also, it is expected that application developers and component providers will provide ViewBean components that build in a particular set of heuristics and schemes for determining and locating parallel JSPs. The users of such components would buy into a particular mechanism, with full knowledge of that decision, and with the ability to use different components with different mechanism as desired.

Although it is possible for other objects to call a ViewBean's `setDefaultDisplayURL()` method to change the URL that ViewBean will use during that request, this technique is discouraged because it violates the semantics of this method and leads to dependencies between objects that are hard to maintain.

J2EE Restrictions

The Servlet specification declares that it is illegal to call `RequestDispatcher.forward()` within a given request more than once, if the request has previously been committed to the client. This has implications for the `ViewBean.forwardTo()` method.

Generally, this restriction means that it is not possible to call `ViewBean.forwardTo()` multiple times within a single request. The only exception is if the current response has not yet been committed to the client. The response is committed to the client's output stream when the response buffer is filled to capacity and must be flushed to make room for more response data. After the first flush to the client, the response is considered committed, and trying to forward again will cause an `IllegalStateException` to be thrown by the servlet container. When using JSPs to render a response, the response buffer is configurable by setting an attribute in the page, so the developer has limited control over this restriction.

However, it would typically be uncommon to forward more than once within a given request—the developer chooses the response once and only once—so this restriction will not affect most applications. One possible exception is if the application encounters an error during the display cycle, *after* a forward, and needs to display an error page. When these conditions occur, be sure to set the response buffer size large enough to accommodate this eventuality.

CHAPTER 3

Programming Guide

This Sun™ ONE Application Framework *Programming Guide* chapter describes common programming scenarios and explains how to use certain fundamental objects in the Sun ONE Application Framework.

In general, the information here is supplemental to the Sun ONE Application Framework JavaDoc. Refer to the JavaDoc for detailed API usage information.

Using the RequestContext

The RequestContext is the primary object that provides Sun-ONE-Application-Framework-related services during a request. It can be obtained from virtually anywhere and at anytime within a Sun ONE Application Framework request, and used to access both J2EE and Sun ONE Application Framework features.

Getting the RequestContext

There are two main ways of getting the current RequestContext instance. First, if the current object such as a ViewBean or Model in which you are writing code implements the `com.iplanet.jato.RequestParticipant` interface, it likely already has an instance of the RequestContext before control is passed to your code. Both the ViewBeanManager and ModelManager will automatically set the RequestContext on ViewBeans and Models that implement the `RequestParticipant` interface before returning these objects to the caller. BasicViewBeans, BasicContainerViews, BasicTiledViews, and BasicTreeViews in particular all have a method called `getRequestContext()` that can be used at virtually any point inside these objects to obtain the RequestContext. One notable exception is that this method cannot be used during the construction of one of these objects, since the RequestContext cannot be set on these objects until they are fully constructed.

The second main technique for obtaining the RequestContext (and some of its primary sub-objects) is to use the static methods in the `com.iplanet.jato.RequestManager` class.

The following table lists these methods and their descriptions.

Method	Description
`getRequestContext()`	Returns the current request's RequestContext object. This method can only be used within the scope of a request.
`getRequest()`	Returns the current request's `javax.servlet.http.HttpServletRequest` object. This method can only be used within the scope of a request.
`getResponse()`	Returns the current request's `javax.servlet.http.HttpServletResponse` object. This method can only be used within the scope of a request.
`getSession()`	Returns the current request's `javax.servlet.http.HttpSession` object. This method can only be used within the scope of a request.

These static methods can be used to obtain the Sun ONE Application Framework RequestContext from inside any object or method in the context of a request, and is most useful to avoid having to add a RequestContext parameter to your object's method calls. These methods can be used without any performance impact on your application; specifically, they do not cause any thread synchronization.

None of these methods can be used outside the scope of a request. For example, you cannot use them to obtain a session object inside an object's static initializer, or inside a servlet's `init()` method.

The servlet event methods specify the RequestContext as a parameter, and many of the event objects used when firing other application events have a RequestContext member. You can use the instance provided in this fashion instead of using one of the other techniques described above.

Getting the Servlet Request and Response Objects

Because a Sun ONE Application Framework application is ultimately a servlet-based application, it has access to the current request's `javax.servet.http.HttpServletRequest` and `javax.servet.http.HttpServletResponse` objects. You can obtain the request and response objects through the RequestContext's `getRequest()` and `getResponse()` methods. You can also conveniently obtain these objects via the RequestManager methods described above.

With only a few cautions (on the response object primarily, explained elsewhere), you can use the request and response objects just as you would within any servlet- or JSP-based J2EE application.

Getting the Session Object

You can obtain the `javax.servlet.http.HttpSession` object if you have either the current request's HttpServletRequest object (see above), or directly via the RequestManager method `getSession()` described above.

Other Available Objects

In addition to the objects mentioned above and a few other conveniences, the RequestContext also provides access to three key Sun ONE Application Framework-specific manager objects.

The following table shows these three key objects.

Method	Description
`com.iplanet.jato.ViewBeanManager`	Obtained via a call to getViewBeanManager(). The ViewBeanManager helps manage ViewBean instances within the current request. All ViewBeans should be obtained via this object.
`com.iplanet.jato.ModelManager`	Obtained via a call to getModelManager(). The ModelManager helps manage Model instances within the current request. All Models should be obtained via this object (or through a ModelReference).
`com.iplanet.jato.SQLConnectionManager`	Obtained via a call to getSQLConnectionManager(). For applications that use JDBC to communicate to RDBMS backends, the SQLConnectionManager provides a thin utility layer on top of the J2EE container's database connection support. Use of the SQLConnectionManager is not mandatory, but is recommended.

RequestCompletionListener Interface

Objects that are interested in being notified of the completion of the current request can register themselves with the RequestContext as RequestCompletionListeners. Registered listeners will receive notification at the end of the current request processing, regardless of the outcome of the current request (for example, regardless of whether the request completed normally or ended in an error).

Objects can use this mechanism to perform last minute tasks, such as saving state in the session, closing open request-scoped resources, or virtually anything else, *provided the task does not affect the output stream*. Objects must register themselves as RequestCompletionListeners on each request if so interested.

Using the Message Writer

The RequestContext also provides a useful tool called the message writer. The message writer is a `java.io.PrintWriter` whose contents are accumulated during the course of a request and then appended to the rendered page right before it is sent to the client. This mechanism works much like the console does in a non-server based application, and is extremely useful for debugging purposes. Interested objects can write to the message writer via the `getMessageWriter()` method on the RequestContext. ViewBeans also have a convenience method called `appMessage()` that can be used to obtain and use the message writer in a single step.

The output of the message writer is necessarily HTML-specific, and so might be incompatible with pages rendered in a different markup language. Furthermore, the message writer is automatically appended by the ViewBean JSP tag at the end of a page, and so will not appear if a page is rendered using some other non-JSP mechanism. See the sidebar for information on how to turn off the message writer.

The appearance of messages written to the message writer can be turned on or off using a configuration parameter in the `web.xml` file. This allows you to turn on messages during development, but turn them off during deployment. For complete information, see Chapter 4, "Deploy an Application" on page 85.

Using ViewBeanManager

When two different parts of the application want to work with a ViewBean instance, they always want to use the same instance. Within a given request, ViewBeans are singletons, meaning that there is only one instance per type of ViewBean in any

given request. To ensure singleton instances, ViewBean's are managed by the ViewBeanManager, which is an instance of `com.iplanet.jato.ViewBeanManager`.

The ViewBeanManager is available from the RequestContext by calling the `getViewBeanManager()` method. Each request receives a new ViewBeanManager instance. Because the ViewBeanManager is available via the RequestContext, and the RequestContext is widely available using one of several techniques (described previously), you can generally obtain a ViewBean instance from anywhere in your application code, not just inside Sun-ONE_Application-Framework-specific objects or methods.

After obtaining the ViewBeanManager, getting a ViewBean instance is as easy as calling the `getViewBean()` method with the class of the desired ViewBean. For example, this code returns the singleton instance of the Login ViewBean:

```
requestContext.getViewBeanManager().getViewBean(Login.class);
```

The ViewBeaManager also has a method that will return a ViewBean by class name, but this method does not provide the compile-time safety of the method shown above.

The `getLocalViewBean()` method deserves additional explanation, although it is not generally used by application developers. This method obtains a ViewBean instance that corresponds to the supplied logical name (this name is also referred to as the ViewBean's local name). A local view bean name is the unqualified name of the ViewBean class without the *ViewBean* suffix. For example, if a module has a ViewBean class `com.foo.Page1`, the local ViewBean name is simply `Page1`. Because the local ViewBean name is automatically qualified by the current module's base package name, local names can only be used to obtain references to ViewBeans within the module whose servlet initially handled the current request.

Using ModelManager

When two different parts of the application want to work with a Model instance, they frequently want to use the same instance so that they access the same data. To allow for sharing of Model instances within a request, the RequestContext contains a ModelManager object, which should be used to obtain nearly all Model instances (possible exceptions to this are noted in a following section).

The ModelManager provides a convenient lookup mechanism for Model instances based on their type and an optional instance name. All lookups within the same request will return the same Model instance specified by type and instance name. With the exception of Models that are specifically stored in the session (see section below), Models obtained via the ModelManager are instantiated and shared on each

request. This is generally more efficient for the application when Model data does not need to be stored between requests and can be easily obtained again when/if needed, because storing objects in the session is an expensive operation.

The key method in ModelManager is `getModel()` and its variations. In general, it is recommended that you provide a Model class as the parameter to `getModel()` versus providing a Model class name—this allows for compile-time checking of the method call.

Although it is possible, it is fairly uncommon to provide an instance name when retrieving a Model from the ModelManager. If you use one of the `getModel()` method variations that does not specify an instance name, the ModelManager will use a default instance name based on the Model's class name, which is generally sufficient for most applications. However, there are specific cases when you want to use two different instances of the same Model type within the same request, and so this feature can be very useful.

Getting and Saving Models in the Session

Occasionally, you will find that you need to store Model data between requests in the HTTP session. The ModelManager provides an easy way to get and save Models in the session for you as you look them up.

In general, you should bias your application to *not* store Model data between requests because storing data in the session is typically expensive (in relative terms) and can limit your application's scalability if overused.

There are several `getModel()` variations that take one or two additional session-related boolean parameters. The first is the `lookInSession` parameter, which if true will cause the ModelManager to check first its local Model cache, and then the HTTP session, before instantiating a new Model instance. The second parameter is `storeInSession`, which if true, will cause the ModelManager to schedule the returned Model instance for setting in the session. The Model instance is not actually set in the session until the end of the current request, to ensure that all changes to the object in the current request are written to the session regardless of the J2EE container's session implementation.

If you have a Model instance and decide that you want to add or remove it from the session manually, you can call the ModelManager's `addToSession()` or `removeFromSession()` methods with the Model instance. Of course, you can also set a Model in the session yourself using the session API, but the `addToSession()` method helps ensure that the object added to the session is later accessible using the `lookInSession` parameter.

ModelTypeMap

Although this feature is seldom used in later versions of the Sun ONE Application Framework (and not in the IDE toolset at all), it deserves some mention. Keep in mind that this feature is completely optional in your application and you should only use it if you expect it to bring you some concrete advantage.

Early in the design cycle of the Sun ONE Application Framework, it was decided that it would be useful to be able to lookup a Model instance (implementation) by specifying just a Model interface. This ability would allow application developers to create implementation-neutral Model interfaces and work exclusively with those throughout the application, instead of relying on implementation details of the Model in their application code. This provided the utmost in loose coupling between the View and Model tiers, and allowed developers to plug in different Model implementations without affecting the View tier at all.

Therefore, the `com.iplanet.jato.ModelTypeMap` class was added to allow for mapping of interface type to implementation type within the ModelManager. When a developer calls the ModelManager's `getModel()` methods, the ModelTypeMap is used to transform the provided Model type to an implementation type. If no mapping has been specified, the original type is returned without changes. This originally provided a fallback mechanism, through which developers could specify implementation types and still obtain valid Model instances.

In reality, most developers found the ModelTypeMap feature cumbersome and limiting, and is therefore seldom used any more. However, it is still an assumed part of the architecture of the ModelManger, and the application servlet still provides an instance of an empty ModelTypeMap to the ModelManager when it is created. Except for those rare cases where you think the ModelTypeMap will provide some value to your application, you can essentially ignore it as just framework infrastructure.

If you do find that you want to use the ModelTypeMap, you should add a static initializer to your application servlet class and add one entry to the ModelTypeMap for each mapping you want. For example:

```
ModeTypeMapBase.addModelInterfaceMapping(
     apppkg.modulepkg.CustomerModel.class,
     apppkg.modulepkg.CustomerModelImpl.class);
```

Then, in your application code, you would call `ModelManager.getModel(CustomerModel.class)` to get the Model, but get back an instance of `CustomerModelImpl`.

Exceptions to using the ModelManager

Most Model types provided with the Sun ONE Application Framework require some initialization to be useful. The normal approach for configuring these Models is to create a subclass that configures the Model appropriately when it is instantiated. However, it might simply be easier to use some of the Model types as configured instances (especially since the IDE toolset does not currently support some Model types as extensible components.) These Models are `com.iplanet.jato.model.BeanAdapterModel`, `com.iplanet.jato.model.SessionModel`, `com.iplanet.jato.model.MultipleModelAdapter`, and `com.iplanet.jato.model.ResourceBundleModel`.

You might occasionally find it useful to instantiate some of these models in your code, but then add them to the session using the `ModelManager.addToSession()` method. This is a good fit for Models that require only one-time initialization but are long-lived, and will allow these Models to be obtained from the ModelManager seamlessly in subsequent requests so they can easily be bound to DisplayFields or other Views.

Using SQLConnectionManager

A large segment of Sun ONE Application Framework applications use an RDBMS to store at least some application data. J2EE provides the JDBC Standard Extension API to make obtaining a database connection in scalable way feasible from within a J2EE application. However, there are some additional concerns when actually developing a real-world application.

Therefore, the Sun ONE Application Framework's SQLConnectionManager adds one additional feature to help developers more easily work with database connections through the lifetime of an application in development. Specifically, it is common for an application in development to run in a container that does not readily support database connection pools. Furthermore, using a connection pool requires additional configuration of that pool in the container, separate from the deployment of the application. This can make it hard to simply deploy a utility or demo application.

SQLConnectionManager adds a thin abstraction to the task of obtaining a JDBC connection. It provides the ability to map datasource names to other values, and easily switch between JNDI and `java.sql.DriverManager` connection lookup techniques, for those cases where it is convenient to be able to do so.

When obtaining a connection from SQLConnectionManager, you provide a JDNI-like datasource name that follows the JDBC Standard Extension convention, such as `jdbc/mydb`. If the SQLConnectionManager is currently using a JNDI lookup to

obtain JDBC connections, it will use this name as the key in the lookup. If instead the SQLConnectionManager is using the DriverManager to obtain JDBC connections, it will use this name to perform a local lookup in its datasource mapping table to obtain a JDBC URL that can be used to obtain a connection.

You set the current connection lookup mode using the `setUsingJNDI()` method of the SQLConnectionManager class. This method is normally called from the static initializer of the `SQLConnectionManagerImpl` class in your application package. If you provide a true value for this parameter, the SQLConnectionManager will lookup JDBC connections using the standard JNDI/JDBC connection lookup technique. This assumes that database connection pools have been already configured and registered in your J2EE container.

If you provide a false value in the call to `setUsingJNDI()`, the SQLConnectionManager will use its own local set of datasource mappings to obtain a JDBC URL that can be sent to the `java.sql.DriverManager` to obtain a connection. This technique is at least an order of magnitude less efficient that using a connection pool, but is generally acceptable in development, or in certain classes of applications that need to work standalone without additional container configuration.

When using DriverManager connection lookups, you can add a datasource mapping using the `SQLConnectionManagerBase.addDataSourceMapping()` method. This mapping will specify a logical datasource name that should be mapped to a physical JDBC URL. For example (using a PointBase URL):

```
SQLConnectionManagerBase.addDataSourceMapping("jdbc/sample",
    "jdbc:PointBase://localhost:9092/sample");
```

The `addDataSourceMapping()` method is normally called from the static initializer of your application servlet class in your application package, so that the mapping is initialized when the application is loaded by the container. (Theoretically, you could add mappings at a later time, but there is little reason to do so since the mapping needs to be consistent for the life of the application.) When using JNDI connection lookups, there is no need to add datasource mappings, though you can if for some reason you want to map one datasource name to another.

Under no circumstances should you use the DriverManager to obtain JDBC connections in a production application! Such usage causes a new database connection to be opened for every use of the connection, and will cause enormous performance and scalability problems with your application. Make sure you use SQLConnectionManager's JNDI lookup mechanism along with a database connection pool in your container when you finally deploy your applications into production.

In your application, you can obtain a JDBC connection from SQLConnectionManager directory using its `getConection()` or `obtainConnection()` methods. The static obtainConnection() method is used to obtain connections outside of the scope of a request, such as during application initialization. You can also bypass SQLConnectionManager and go directly to a JNDI lookup or to DriverManager if you want, but the Sun ONE Application Framework classes that use JDBC, such as `com.iplanet.jato.model.sql.QueryModelBase`, will always use SQLConnectionManager to obtain database connections.

Using the RequestManager

The RequestManager provides a handful of static methods you can use to obtain key request-scoped objects.

The following table shows these methods.

Method	Description
`getRequestContext()`	Returns the current request's RequestContext object. This method can only be used within the scope of a request.
`getRequest()`	Returns the current request's `javax.servlet.http.HttpServletRequest` object. This method can only be used within the scope of a request.
`getResponse()`	Returns the current request's `javax.servlet.http.HttpServletResponse` object. This method can only be used within the scope of a request.
`getSession()`	Returns the current request's `javax.servlet.http.HttpSession` object. This method can only be used within the scope of a request.
`getHandlingServlet()`	Returns the `javax.servlet.Servlet` instance that initially handled the current request. For the foreseeable future, this servlet is expected to be a subclass of `com.iplanet.jato.ApplicationServletBase`. This method can only be used within the scope of a request.

Note – The name *RequestManager* is used for the class that is a placeholder for a number of features that are to be added in a future version of the Sun ONE Application Framework. Its current status as just a collection of static utility methods is temporary.

Logging

Logging support in the Sun ONE Application Framework is built on top of the standard ServletContext-based logging feature, which itself is a fairly minimal. Therefore, the Sun ONE Application Framework's logging feature is only intended to add minor functionality to this existing baseline mechanism, and is not intended to supply a full-featured logging solution such as that provided by Log4J or JDK 1.4's logging package. Instead, it is meant to be lightweight and convenient to use for developers who want to use their J2EE container's native logging features.

You can access the Sun ONE Application Framework's logging via the static methods in the `com.iplanet.jato.Log class`. In addition to these methods, this class provides the ability to filter messages based on log levels and to echo the log to the standard out.

Logging Messages

To log a message, simply call one of various static `log()` methods in the `com.iplanet.jato.Log` class, optionally providing a log level parameter. If a log level is provided, the Log class determines if that message should be logged based on the currently enabled log levels. If a log level is *not* provided, the currently enabled log levels are *not* considered, and the message is always logged.

The messages that are allowed to pass are sent to the container via the ServletContext's various `log()` methods, and generally appear in the container's log and/or console.

Log Levels

The primary value added to the baseline ServletContext logging mechanism is the ability to log messages using levels and to filter the current log output by these levels. Log levels fall into a few major categories.

The following table shows these log level categories.

Log Level Category	Description
Error levels	Levels that specify errors or warnings in the application. Includes the WARNING, ERROR, and CRITICAL levels. All levels in this category can be enabled via the ANY_ERROR level.
Debug levels	Levels that are used by the developer for informational or debugging purposes. Includes the STANDARD, TERSE_DEBUG (less information) and VERBOSE_DEBUG (more information) levels. All levels in this category can be enabled via the ANY_DEBUG level.
Trace levels	Levels that are used by the developer and/or the Sun ONE Application Framework to trace request execution. Includes the JATO_TRACE, JATO_QOS_TRACE and APP_TRACE levels. All levels in this category can be enabled via the ANY_TRACE level. The JATO_TRACE and JATO_QOS_TRACE levels are not for developer use and are only used to view trace information logged by the framework itself.
User levels	Levels that are defined and used by the developer in whatever fashion desired; the meaning of each of these levels is application- and developer-defined. Includes the USER_LEVEL_1, USER_LEVEL_2, and USER_LEVEL_3 levels. All levels in this category can be enabled via the ANY_USER_LEVEL level.

Except for the JATO_TRACE and JATO_QOS_TRACE log level, all log levels are primarily for developer use. In other words, the core runtime will generally not log at any of these log levels (with the exception of some of the error levels). However, Sun ONE Application Framework components written by other authors might log at these levels, so do not expect to have full control over them. The one exception to this rule is the several USER_LEVEL_* log levels, which by convention should be reserved for current application usage only. Distributable components should not use these levels once published.

Multiple log levels can be enabled at a time by logically ORing multiple levels together in a call to Log.setEnabledLevels(). For example, the following code enables several log levels at once:

```
Log.setEnabledLevels( STANDARD | ERROR | CRITICAL | JATO_TRACE );
```

You can check if a level is currently enabled by calling the isLevelEnabled() method with the level to check. In addition, there are several convenience log levels that automatically include several others, such as the various ANY_* log levels, and the DEFAULT_LOG_LEVELS level.

Finally, you can change log levels at any time from application code, a servlet init() method, a static initializer, or basically anywhere you have static access to the Log class at runtime. However, there is only one set of Log settings per deployed

application, and the current settings are shared by all currently executing request threads. Therefore, it generally wouldn't make sense to change the log levels on a per-request basis.

There are no current plans to significantly improve the Sun ONE Application Framework's logging facility, as it is meant to satisfy minimal requirements using only baseline J2EE logging features. If you need a richer logging facility for your application, you are encouraged to use Log4J or JDK 1.4's built-in logging package.

Logging to Standard Out

Because it can be difficult to easily access the J2EE container's log file, the Log class has the ability to echo whatever is sent to the ServletContext log to the console's `System.out` stream. This feature is frequently helpful during development since the current container's console is readily visible on the developer's machine.

Making Log Messages Stand Out

Because the Sun ONE Application Framework logging outputs to the J2EE container's log, which is probably also filled with other log messages, the `Log` class provides the ability to set a message prefix via the `setMessagePrefix()` method. This prefix will be appended to every messages logged via the `Log` class. With proper choice of prefix, Sun ONE Application Framework log messages can be made to stand out from other messages in the same log. The default prefix is three dashes followed by a space, "--- ".

Working with Values

The following sections describe several ways for working with values in a Sun ONE Application Framework application.

Working with DisplayField Values

The first, and most common, approach for obtaining values is to get them from DisplayFields directly. For example, you can ask a TextField for its value by obtaining a reference to the TextField and then calling its `getValue()` method. This

approach is comfortable for View developers, because they can interact directly with the View hierarchy they are building, and all the information needed to get the data is generally contained in a single file.

You can get a reference to a DisplayField or any other View (with the exception of ViewBeans) by calling its parent's `getChild()` method, providing the unqualified local name of the child View component. (This method is defined in the `ContainerView` interface, and so is present for all types of container Views, including ViewBeans.) Because `getChild()` returns an instance of View, and not just display fields, you must cast the result to a usable type before using it.

The following is an example of getting a BasicDisplayField child from within a ViewBean and getting its value:

```
View childView=getChild("textField1");
Object value=((BasicDisplayField)childView).getValue();
```

Because this technique can require lots of casting, there is a convenience method within the core ContainerView implementations (BasicContainerView, BasicTiledView, and so on):

```
DisplayField field=getDisplayField("textField1");
Object value=field.getValue();
```

Because the value returned is a DisplayField instance, you can call `getValue()` on it directly, even without casting it to the specific component type.

Finally, you can call another convenience method, `getDisplayFieldValue()`, to get a value in a single step:

```
Object value=getDisplayFieldValue("textField1");
```

There are also variants of this method that return specific types of objects, such as `getDisplayFieldStringValue()`, `getDisplayFieldBooleanValue()`, and `getDisplayFieldIntValue()`.

The IDE toolset automatically generates child View component name constants in your ViewBean or ContainerView classes, using the pattern `CHILD_<childName>`. Therefore, it is not necessary to use string literals in the calls to `getChild()` or its variations as shown above. Also, the IDE generates child View component accessor methods such as `get<childName>Child()`, so that you can obtain a reference to a View component without casting.

Setting DisplayField values is similar to getting them. Once you have a DisplayField instance, you can call its `setValue()` or `setValues()` methods to change its value. The core ContainerView implementations also provide a convenience method `setDisplayFieldValue()` with several variants to allow developers to easily set values.

The following is an example of getting DisplayField values, processing them, and then setting the result on another DisplayField:

```
int value1=getDisplayFieldIntValue("intField1");
int value2=getDisplayFieldIntValue("intField2");
int result=value1+value2;
setDisplayFieldValue("message",value1+"+"+value2+"="+result);
```

Some ContainerView variations, such as TiledViews, might require the use of additional methods to properly obtain and set data. These details are covered elsewhere in this guide.

Working with Model Values

The second technique for getting application values is to go to a Model directly. Normally, DisplayFields are bound to Models, which provide storage for their values. This binding occurs via a ModelReference object, and a field name. Generally, a DisplayField is bound to a Model field.

Thus, it follows that if you have a reference to a Model and know which field to get, you can ask the Model for data directly using the `Mode.getValue()` or `Model.getValues()` methods. The easiest way to get a reference to a Model within an IDE-created View component is to use a ModelReference object. ModelReference objects are created automatically as developers define bindings between DisplayFields and Models, and they are available to methods in the class to use also (this allows everyone to share the same Model reference).

Once you have a ModelReference instance, simply call its getModel() method to obtain a Model reference. You can then get values from the Model:

```
Model model=modelReference1.getModel();
Object value1=model.getValue("field1");
Object value2=model.getValue("field2");
```

The IDE toolset automatically generates field name constants in Model classes created within the IDE. You can use these constants instead of string literals to more easily refer to fields within the model.

You can also call the `Model.setValue()` and `Model.setValues()` methods to set values on a Model. You simply provide the field name along with the new value in the call.

Some Model specializations, such as DatasetModel, might require the use of additional methods to properly obtain and set data. These details are covered elsewhere in this guide.

Getting Values Using the J2EE API

Finally, developers can obtain application data using the low-level J2EE/Servlet API. The Servlet API defines the `javax.servlet.http.HttpServletRequest` class, with methods such as `getParameter()` and `getParameterValues()` that can be used to get values that were directly submitted in the request. You simply need to know the names of the appropriate parameters to get their values.

Sun ONE Application Framework View components use a naming scheme to automatically generate qualified names for DisplayField components. This qualified name uses the name of the DisplayField prefixed in order of upward traversal to the root by the names of its parents. These names are qualified by the dot (".") character by default (see the deployment section for information on using a different value).

For example, if I have a ViewBean named *Foo* which contains a ContainerView named *bar*, which further contains a text DisplayField named *bat*, the qualified name of the field would be `Foo.bar.bat`. If this field existed within an HTML form, the developer could get the value of this field by looking for the parameter named *Foo.bar.bat* in the request.

Some ContainerView variants like TiledView add additional information to the qualified name, to allow decoding of multiple values from the request. TiledView adds a subscript to the field name to distinguish the row upon which it appears. Depending on the number of tiles submitted in the request, there might be several similar parameter names distinguished by subscripts. If in the above example, if *bar* were a TiledView, you might see the following parameters in the request:

```
Foo.bar[0].bat
Foo.bar[1].bat
Foo.bar[2].bat
```

See the documentation for each component to see how it generates child names that are contained within it.

As you might suspect, the qualified names generated for components are what allows the framework to automatically map parameters in the request back to display fields during the submit cycle, and keep components from different authors

distinct from one another. If field names instead used a flat namespace, it would not be possible to create ready-to-use components that could assembled to create an application.

The core DisplayField implementations (BasicDisplayField, etc.) have methods called `getRequestValue()` and `getRequestValues()` that can be used to obtain the values that correspond to those fields from the request parameters. This is generally a far more convenient way to obtain the value submitted for a field than going to the request parameters directly using the field name. These methods can be used to refer to the values originally submitted in the request, even if the current field's value has changed.

You cannot set values in the request, as they are read-only.

Using Display Events

Many times when rendering a page, you will find that you want to modify the way a View renders, or even skip displaying it at all. In other frameworks that are JSP-centric, this is difficult or impossible to do without using complex control or other logic in your JSP, where it is inherently hard to debug and maintain. Even solutions like the JSP Standard Tag Library can result in complex code-like structures in your JSP, essentially trading programming in Java for programming in JSP tags.

By contrast, the Sun ONE Application Framework provides what are called display events to allow developers to perform complex display-oriented logic, but outside of the JSP and instead in their View classes, where Java code makes sense. There are two main display-related events, one triggered for the beginning of display of a component, and one triggered for the end of display of a component. These events differ based on whether the component is a ContainerView itself, or a child View.

Container Display Events

When a ContainerView is displayed during rendering of a JSP, its `beginDisplay()` and `endDisplay()` methods are automatically called when the corresponding JSP tags begin and end its display, respectively. While developers can override these methods in subclasses, many superclasses have implemented these methods to perform useful and necessary tasks upon display notification. For this reason, developers must always call the super version of the method when they override these methods, thereby making them harder to use during application assembly.

Therefore, the event methods reserved for application developer use (at least in components derived from `com.iplanet.jato.view.ContainerViewBase`) are the `beginComponentDisplay()` and `endComponentDisplay()` methods. These methods will be triggered as appropriate by the superclass, and are more or less guaranteed not to have any implementation in the superclass (or more precisely, no implementation that can't be completely ignored by the application developer). Application developers can find these methods listed as events in the Events context menu on ContainerViews added as children to other ContainerViews.

Component developers should directly override the `beginDisplay()` and `endDisplay()` methods to respond to these events, thereby leaving the component versions for application assemblers.

Child Display Events

The container-related display events described in the section above allow the ContainerView component itself to respond to display notifications, but frequently, containers need to respond to rendering of their children. Thus, the ContainerView defines the `beginChildDisplay()` and `endChildDisplay()` methods to be used during rendering to tell the container that one of its children is about to be rendered. The `beginChildDisplay()` method returns a boolean result, where true indicates that the child should be rendered, and false indicates that the child should be skipped.

Component developers can override these methods directly, but application developers have an easier mechanism for responding to fine-grained child display. The various extensible pagelet components included in the Sun ONE Application Framework component library look for methods of a certain signature when responding to the begin or end child display notification. If these events exist, they are called each time the corresponding child is rendered. The signature of these methods looks like the following:

```
public boolean begin<child name>Display(ChildDisplayEvent event)
    throws ModelControlException;
public String end<child name>Display(ChildContentDisplayEvent
event)
    throws ModelControlException;
```

where *<child name>* is the capitalized name of the child view. For example, if a page has a child called *foo*, the application developer could respond to its rendering by simply implementing the `beginFooDisplay()` method with the signature show above.

The `boolean` return value in the case of the `begin<child>Display()` method allows the developer to optionally skip rendering of the child. The `String` return value in the case of the `end<child>Display()` method is the actual markup that will be rendered for that child. The event object contains the markup calculated for the child during JSP rendering, but developers can tweak or completely override the markup in the end display event.

Use of the child display events is a powerful technique for creating dynamic pages, and ensures that JSPs are kept as code free as possible. Common uses of display events are: to skip display of a child based on user role or other per-user information; to calculate a child's value just before it is rendered (each time it is rendered); to execute a model to determine the number of rows it returned; to add additional markup to a rendered HTML control; to dynamically change the color of a table row; to periodically insert header information during TiledView rendering; and many other users. Developers will find display events to be an indispensable tool when writing enterprise applications, and because display-related code is kept in one place in the ViewBean or ContainerView, every JSP that uses that component can use the same logic.

Content Tag

In many cases, it is useful to be able to associate Sun ONE Application Framework display events with arbitrary sections of a JSP, whether they include dynamic content, static content, or a mixture of both types. The *<jato:content>* tag provides exactly this feature. The name attribute of this tag is used to determine the display event callback to the enclosing ContainerView.

When using the content tag, you implement display events in exactly the same fashion as you would for any child View. The difference is that there is no child View, and the content that will be rendered will come from whatever the content tag encloses (or whatever the endDisplay event for that tag returns).

A common use for content tags is to conditionalize rendering of a portion of a page, or to inject arbitrary markup into a page at a specific point. For example, content tags are one easy way to change the color of each tile of a TIledView as it is rendered.

Using ViewBeans

ViewBeans are just special types of ContainerViews and inherit most of their behavior from the superclass. The section "Using ContainerViews" on page 71, contains most of what developers need to know about using ViewBeans. The following sections detail the additional ViewBean specifics.

forwardTo() Method

ViewBeans, as pages, are the primary artifact developers create in their applications. As such, they have key methods for controlling the request. Although developers are free to directly use servlet features such as `RequestDispatcher` to render a response to a client, in practice it is much easier to use the `forwardTo()` method on a ViewBean to begin rendering of a page. The primary reason this approach is easier is because each ViewBean knows which JSP it is associated with, and so can handle the details of forward the servlet request for you. Furthermore, it keeps application developers thinking about *pages* instead of lower-level J2EE primitives.

The `forwardTo()` method offers an opportunity for a ViewBean to perform some logic before beginning the display phase. The most notable technique is for the ViewBean to dynamically select the JSP it wants to render before actually forwarding the request. For example, a ViewBean can determine that the current request comes from a WAP device, and so it instead will render a WML JSP as the response instead of an HTML JSP. This ability to render the same ViewBean using multiple associated JSPs is called parallel content, and is covered in Chapter 2, Develop an Application, under "Display URLs and Parallel Content" on page 48.

Page Session

Often, developers need to retain some information between requests, but putting this information in the server-side HTTP session can get the application out of sync if the user uses the browser's Back button. Instead, ViewBeans have what is called a page session to help solve this problem. Page session works much like the HTTP session, except page session attributes are stored in the rendered response to the client and resubmitted on the next request. If the user uses the Back button, the page session is automatically kept in sync because each version of the page session for the specifically rendered page is cached in each page on the client.

Page session is most often used to track a user's context in an application, much as an HTML hidden field might be used by traditional Web applications. However, the advantage of page session over hidden fields is that it is automatically appended to

all links and forms, and might contain complex objects, not just strings. Furthermore, because browsers have a limitation on the length of URLs sent to them during an HTTP GET request, the page session mechanism can compress the page session if it reaches a certain size (the compression is normally around 40-50% effective).

A given set of page session attributes is specific to a single ViewBean. Page session attributes are not shared between ViewBeans. That application will need to copy page session attributes from the handling page to the displaying next if the attributes should be preserved for the next request.

The page session API is covered in detail in the `com.iplanet.jato.view.ViewBean` JavaDoc.

Important: The page session is no more secure than any other unencrypted value send to the client, such as values stored in an HTML hidden field. It is merely encoded, not encrypted, when sent to the client, so a determined malicious user could craft a request that contained a modified or bogus page session. Be sure to take this fact into account when deciding what your application stores in the page session.

Client Session

The client session is similar in concept to the page session, but it is shared by all pages (and other objects) in the application. Like the page session, it is automatically appended to all Sun ONE Application Framework URLs, and might be automatically compressed if it becomes large.

The client session is available from the RequestContext. The API is detailed in the `com.iplanet.jato.ClientSession` JavaDoc.

Important: The page session is no more secure than any other unencrypted value send to the client, such as values stored in an HTML hidden field. It is merely encoded, not encrypted, when sent to the client, so a determined malicious user could craft a request that contained a modified or bogus client session. Be sure to take this fact into account when deciding what your application stores in the client session.

Using ContainerViews

ContainerViews are analogous to panel components in other visual development environments. They provide a way to group a set of contained components so that they can be manipulated as a group. ContainerViews also form the basis for most complex components (both distributable and non-distributable).

There are several standard specializations of ContainerView that add additional behavior, such as the ability to repeatedly render its contained components. For more information, see the section "Using TiledViews" on page 76.

The ability to add additional behavior to ContainerViews beyond the basic ability to contain other components is what makes them so powerful and versatile.

IDE Support for ContainerViews

The Sun ONE Application Framework IDE toolset helps developers build ContainerViews as easily as selecting them from a list and adding a ContainerView subclass into their application. However, if you have studied the `com.iplanet.jato.view.ContainerView` interface, you might have noticed that it does not contain the methods that would allow an automated tool to describe components that should be created as children. The interface is, rather, written from the perspective of runtime framework requirements, and the framework does not care how child components are actually created and managed by the ContainerView at runtime.

Therefore, IDE support for creating ContainerView-based components relies upon a convention. To create a new extensible ContainerView component that the IDE can manipulate, the component *must* provide the following methods. However, if the component ultimately derives from `com.iplanet.jato.view.ContainerViewBase`, these conventions are already satisfied, and you need not worry about them. This convention also applies to *all* specializations of ContainerView, including ViewBeans and TiledViews.

The following table shows the methods provided by the component to create a new extensible Container View component that the IDE can manipulate.

Method	Description
`void registerChild(String name, Class type)`	This method will be called during initialization of the component to register all its child component's names and types. This information can then be used to satisfy certain ContainerView interface and implementation methods without requiring instantiation of the child components.
`View createChildReserved(String name)`	This factory method will be called to create the named child component as needed. The implementation of this method will automatically be generated by the IDE toolset as the developer adds child components to and manipulates child components in the container.
`View createChild(String name)*`	This alternate factory method will be called to create the named child component as needed. The implementation of this method will *not* be automatically generated by the IDE toolset, but is conventionally left available for the developer to manually add components to the container through code. *Unlike the other two methods in this list, this method is *not* required by the IDE toolset, but its inclusion is strongly encouraged in new component types, as Sun ONE Application Framework developers are generally accustomed to its presence.

At a future time, these methods might be included in an additional interface to be implemented by IDE-supported ContainerView components. However, this interface is not a requirement at this time.

ContainerView API

The discussion below focuses on the default ContainerView implementation (`ContainerViewBase`) provided as part of the Sun ONE Application Framework component library, as it implements the conventions described in the previous section, IDE Support for ContainerViews. However, the `ContainerView` interface is the minimal requirement for a ContainerView components, and implementations might differ. See the JavaDoc for the `com.iplanet.jato.view.ContainerView` class to understand the minimum requirements for implementing a ContainerView component.

The baseline ContainerView API focuses on the ability to contain other View components, including other ContainerViews. Inherent in this feature is the notion of parent and child components. A parent component contains one or more child

components. A child component can also be a parent to its own children, etc., so that an arbitrarily complex containership hierarchy can be established simply by adding components as children of other components. Methods such as `ContainerView.getChildNames()`, `ContainerView.getChildType()`, and `ContainerView.getChild()` are the core methods that allow consumers of the ContainerView (both developers and the framework) to work with its child components.

To define a component as a child of a ContainerView, there are generally two steps. If you are using the IDE toolset, these steps will automatically be managed for you (via code generation) as you visually add child components to your ViewBeans or other visual components.

The first step is to invoke the `registerChild()` method on the parent component to register a name-type mapping. This step is necessary to help the ContainerView make decisions about its child components without actually needing to instantiate them. This feature is important for efficiency reasons, as the framework does not want to have to create child component objects on every request, regardless of whether they will be used or not during that request. The information gathered via registration is used to satisfy calls to the `ContainerView.getChildType()` and `ContainerView.getNumChildren()` methods without instantiating all child components.

The second step is to tell the ContainerView how to instantiate a given child by name. Each ContainerView has two methods for this purpose, `createChild()` and `createChildReserved()`. These methods are identical in behavior, but the latter is reserved for automated tool use. These methods are factory methods that instantiate, configure, and return the child specified by the name parameter. This method is called at most once per child during a request, as the implementation of the component will cache the component returned by this method. Not all child components will be instantiated on every request, or at the same time during a request. This method is called lazily to instantiate the child component on an as-needed basis. For example, if you call the `getChild()` method to obtain a reference to a child and it hasn't been instantiated yet within that request, `createChild()` (or `createChildReserved()`) will be called to obtain a child component instance. Additional calls to `getChild()` within that request will not result in a call to `createChild()`, since the component has already been instantiated and cached by the container.

You should *never* call `createChild()` or `createChildReserved()` directly to obtain a child component instance. Instead, call `getChild()`, `getDisplayField()`, or an IDE-generated accessor method (for example, `getFooChild()`) to obtain a reference to a component. Also, in general, you cannot reliably call any of these methods or obtain a child component reference from within the ContainerView's constructor. The only caveat to this prohibition is if the RequestContext has been set on the ContainerView before the call to obtain a child component (or if all child components have static access to the RequestContext). This

helps ensure that the ContainerView and all its children have access to whatever request resources they require (such as Models) to be properly instantiated and configured.

Using ContainerViews in Your Application

ContainerViews are in many ways just like any other child component you might add to a page (or other ContainerView)—they have properties that can be set and have display events associated with them.

Default Model

All ContainerViews have what is called a default model. This is a model that should act as the *default* storage for any child components that do not have Model storage otherwise specified. Unless otherwise specified, the default model is an instance of `com.iplanet.jato.model.DefaultModel`, which implements a basic in-memory storage mechanism.

The advantage of using the default model is that it does not require a complex Model binding, and allows the field to take any value the developer wants. Furthermore, DisplayField values are submitted and stored in the default model as is, so the developer has considerable flexibility in inspecting and manipulating the values before sending them to a backend system or another Model.

In the current Sun ONE Application Framework implementation of the various BasicDisplayField types, if a ModelReference is not explicitly set on the component, it uses its parent's default model.

Child View Paths

This name of a child in a call to `ContainerView.getChild(String)` might be a qualified view path, using forward slashes ("`/`") as delimiters. All components in the path except the last must refer to a ContainerView or a derivative of ContainerView (such as TiledView). Both relative and absolute paths are possible. If a name path begins with a forward slash, the name is assumed to be relative to the root (the ViewBean). If the path does not begin with a forward slash, the name is assumed to refer to a child relative to the current container. Two dots ("..") can be used to refer to the container that is the parent of the current container.

Using TiledViews

TiledViews are special types of ContainerViews that render their children repeatedly. Each repetition is called a tile. Tiles are most commonly used to generate rows in a table, but they can also be used to generate a set of tabs, a breadcrumb control, or any other structure that requires an iterative rendering. In addition to the standard ability to manage children, TiledViews provide methods such as `next()`, `first()`, and `last()` to control iteration through a set of tiles.

Each TiledView must be associated with a primary model of type `DatasetModel`. DatasetModels provide data in a tabular fashion, and when a TiledView is rendered, it essentially acts like an iterator over the associated primary model's data. At each row of the model, child Views are rendered. If these children (particularly DisplayField children) are bound to the primary model, their values will change with each tile iteration.

The TiledView's `nextTile()` method is used during rendering to move to each tile as needed. This method returns a boolean value that indicates whether the TiledView moved to a new tile, and the default implementation uses the primary model's `next()` method to determine if any more data is available to render. A common technique is to override the `nextTile()` method to perform additional checks or to initialize data needed for that tile's rendering.

During the submit phase, a TiledView must first be positioned to a given tile before that tile's DisplayField values can be read. If you place a CommandField component inside a TiledView and want to read its value when handling a request initiated by its activation, you must use the following code snippet to position the TiledView to the correct row before getting the CommandField's value:

```
getPrimaryModel().setLocation(
    ((TiledViewRequestInvocationEvent)event).getTileNumber());
```

where event is the RequestInvocationEvent parameter provided to your request event handler.

Nesting TiledViews (putting a TiledView within a TiledView) works for rendering of read-only data, but generally does not work as expected if data from the inner TiledView is submitted back to the application. The submitted data is present in the request parameters, but cannot be mapped properly to target models for reasons beyond the scope of this guide. Instead, a technique that might work for your application is to set the inner TiledView to use a SimpleCustomModel (or its own default model) as its primary model. Then, ignore that data that is pushed into the model, and instead iterate over the nested TiledViews while calling the `getRequestValue()` method on each `BasicDisplayField` child. This will return

the servlet parameter value submitted for that combination of TiledView positions without you having to do a lot of string parsing to manually construct the parameter names.

Using TreeViews

TreeViews are special types of ContainerViews that render children in a hierarchical way. In addition to the standard ability to manage children, they provide support for determining the current node location when rendering a hierarchy of data, and have state data associated with them that determines which nodes are expanded or collapsed.

Each TreeView must be associated with a primary model of type `TreeModel`. TreeModels provide data in a hierarchical fashion, and when a TreeView is rendered, it essentially acts like an iterator over the associated primary model's data. At each node of the tree, child Views can be shown, hidden, or customized based on rules developers declare in the associated JSP tags. For this reason, TreeViews are relatively complex compared to other types of ContainerViews (but are considerably easier to use than trying to accomplish the same tasks manually).

The TreeView works heavily in conjunction with the Sun ONE Application Framework tag library, and has several tags defined just for it's special requirements. The tag library reference documentation covers these tags in detail.

Using Executing Models

In addition to containing data, most models act as business delegates and need to support operations to obtain data from or modify data in some back-end system. Therefore, the Sun ONE Application Framework defines the notion of executing models as models that can also have operations invoked on them.

There are several types of executing models in the Sun ONE Application Framework. The most basic type is represented by the `com.iplanet.jato.model.ExecutingModel` interface. This interface exposes a single method, `execute()`, that can be used to invoke arbitrary name operations on the model. Each model that implements this interface must document the operations it supports so that developers can call the `execute()` method with the appropriate operation name.

However, there are a handful of operations common to most models. These operations fall into four types: retrieve, insert, update, or delete. (These types are akin to SQL operations, but have no formal relationship, though they might be mapped to SQL operations in the case of a JDBC-based model.) The ability to process these operations is represented by the `RetrievingModel`, `InsertingModel`, `UpdatingModel`, and `DeletingModel` interfaces. Models that implement these interfaces declare that they support this type of execution interaction; thus no operation name is required to be specified by callers.

Although the most general type of executing model interface, `ExecutingModel`, allows nearly any operation to be invoked on the model, the more specific interface types allow the framework to work with models in a generic but well-understood way. In fact, these specific types are the foundation for the WebActions feature. However, they also provide significant value to Sun ONE Application Framework developers in that they clearly define the semantics of the models they build or with which they interact.

In general, executing models (of all types) will be executed by developers at specific points in their applications. For example, a request event handler might get some data from the current request, put it into a model that implements `UpdatingModel`, and then call `update()` on that model to push that data into the back-end system represented by that model. However, WebActions (covered below) can also execute specific types of models at certain points in the application automatically, freeing the developer from having to write code to move data into and out of models.

Using BeanAdapterModel

The BeanAdapterModel allows developers to use one or more JavaBeans as the backing datastore for a model. This allows DisplayFields to be bound to JavaBean properties, and is a convenient approach when you have an application object model and want to leverage automatic binding of these objects to a view.

The JavaBeans to use in the model can be set by the developer explicitly via the `setBean()` or `setBeans()` methods, or obtained via lookup mechanism in the standard J2EE request, session, or application scopes. This feature allows for easy interoperation of the model with other J2EE components. To automatically have the model look up its beans, specify the scope using the Bean Scope property or the `setBeanScope()` method.

The BeanAdapterModel supports the `DatasetModel` interface, allowing arrays of JavaBeans to be used in the model, one element per row. It also supports pagination via WebActions. Developers that create subclasses of BeanAdapterModel can also implement alternative executing model methods to allow them to obtain beans from EJBs, object-relational mapping layers, or some other system that provides data as

JavaBeans. This allows BeanAdapterModel to be used in a wide variety of situations and as a flexible but simple way to integrate non-Sun-ONE-Application-Framework objects into an application.

Using ObjectAdapterModel

See the JavaDoc for `com.iplanet.jato.model.object.ObjectAdapterModel` for usage information.

Using WebActions

WebActions are special behaviors built into the Sun ONE Application Framework View components that allow them to provide automatic model execution and pagination. Specifically, the BasicViewBean, BasicContainerView, and BasicTiledView components support WebActions as implementations of the `com.iplanet.jato.view.WebActionHandler` interface.

Hand in hand with the WebAction View components are WebAction models. These are not new types of models, but rather the association of a model with a WebAction View component. Models are associated with WebAction View components via the following properties: Auto Deleting Models, Auto Executing Models, Auto Inserting Models, Auto Retrieving Models, and Auto Updating Models. These properties mirror the standard categories of executing models, meaning any models that implement one or more of the standard `ExecutingModel` interfaces can be used as WebAction models.

Once a model is associated with WebAction component, it can be executed in the context of a WebAction by calling its handle `handleWebAction(int)` method. Developers can call this method directly from event handlers, or it can be called automatically, such as by a WebAction Command component. Here is a typical example usage of a WebAction within a request event handler method:

```
public void handleButton1Request(RequestInvocationEvent event)
     throws Exception
{
     handleWebAction(WebActions.ACTION_NEXT);
     getParentViewBean().forwardTo(getRequestContext());
}
```

The generally correct behavior after executing a WebAction is to reload the current page, though for certain WebActions such as ACTION_UPDATE, ACTION_INSERT, and ACTION_DELETE, it might make sense to forward to a different page.

WebAction Types

Developers must specify the type of WebAction that should occur when calling the handleWebAction(int) method.

The following table outlines the available types of WebActions.

WebAction Type	Description
ACTION_FIRST	Execute all auto-retrieving dataset models, and move to the first row of the dataset.
ACTION_NEXT	Execute all auto-retrieving dataset models, and move to the next group of rows (as defined by a previous action).
ACTION_PREVIOUS	Execute all auto-retrieving dataset models, and move to the prior group of rows (as defined by a previous action).
ACTION_LAST	Execute all auto-retrieving dataset models, and move to the last group of rows.
ACTION_UPDATE	Execute all updating web action models.
ACTION_INSERT	Execute all inserting web action models.
ACTION_DELETE	Execute all deleting web action models.
ACTION_CLEAR	Execute no models, resulting in a blank resulting page.
ACTION_EXECUTE	Execute all executing web action models.
ACTION_REFRESH	Re-execute the current auto-retrieving dataset models.

WebAction Events

Because WebActions can occur in contexts where the application developer might have no way to prepare a model for execution or respond to error conditions (such as within a try...catch block), there are a number of WebAction events that developers can implement.

The following table shows these WebAction events.

WebAction Event	Description
`beforeWebActionModelExecutes()`	Called before each WebAction model is executed, regardless of the WebAction context (for example, retrieve, insert, update, delete). Developers can use this method to prime the models for execution, perhaps by setting request-specific values on them.
`afterWebActionModelExecutes()`	Called after each WebAction model is executed, regardless of the WebAction context (for example, retrieve, insert, update, delete). Can be used to check, summarize, or even sort model data before it is rendered by the View.
`afterAllWebActionModelsExecute()`	Called after all WebAction models are executed for the current invocation of `handleWebAction()`, regardless of the WebAction context (for example, retrieve, insert, update, delete). Can be used to take last minute action before the View is rendered in association with a WebAction.
`onWebActionExecutionError()`	Called when an error occurs executing a WebAction model. Can be used to take corrective action or to abort the request.

Auto-Retrieving Models

The most common use of WebActions is to automatically execute a model when a WebAction View component renders. Rather than requiring the developer to insert code into the application at a specific point to execute a model for display of a page or pagelet (such as the `beginComponentDisplay()` event), associating one or more models as retrieving WebAction models allows the component to retrieve data automatically when display begins.

The various WebAction View components automatically try to execute any retrieving WebAction models when they begin rendering. If there are no retrieving WebAction models, no models are automatically executed. Similarly, auto-retrieval of models can be manually controlled via the `setAutoRetrieveEnabled(boolean)` method. Setting this value to false skips auto-execution.

There might be cases where you might want to manually execute an auto-retrieving model instead of waiting for it to be executed when its associated WebAction View component is displayed. (For example, this is common if you want to know the number of rows of data in a model before you begin rendering a child component with an associated auto-retrieving model). In such cases, you can execute the model manually and then call `setAutoRetrieveEnabled(false)` on the associated View component. The View component will render normally.

Auto-retrieving models are executed each time the associated View component is rendered on a page. To see any benefit to auto-execution, you generally need to bind a component's child DisplayField components to the model that will be auto-executed.

Pagination Using WebActions

One of the most valuable WebAction features is the ability to paginate through sets of data using arbitrary models. The pagination WebActions (`ACTION_FIRST`, `ACTION_NEXT`, `ACTION_PREVIOUS`, and `ACTION_LAST`) use retrieving WebAction models to allow users to *page* through sets of data that might be too large to display on a single page.

As long as a model implements the `RetrievingModel` and `DatasetModel` interfaces (or the combined `RetrievingDatasetModel` interface), it can be used with the pagination WebActions. This means that you can essentially any kind of model with these actions, regardless of its data's origin. For example, models using JDBC, XML, Web services, CICS mainframe data, or just about any other data source can be paginated using WebActions—providing they implement the `RetrievingModel` and `DatasetModel` interfaces.

There is nothing special you need to do to enable pagination support; just use these WebActions like any other WebActions, keeping in mind the additional model type restrictions. Among the components shipped with the Sun ONE Application Framework component library, the various custom model components, JDBC QueryModel, BeanAdapterModel, ObjectAdapterModel, and WebServiceModel generally support pagination as both `RetrievingModel` and `DatasetModel` implementations.

When to Use WebActions

WebActions are a purely value-added feature of some of the standard Sun ONE Application Framework components. As such, they are at the developer's discretion to use as desired. In general, the auto-retrieval and pagination WebActions are the most useful, the former because of the lessening of developer code, and the latter because of the general difficulty of writing pagination code.

As a general rule of thumb, use WebActions when they make things easier. However, if you feel constrained by them for some reason, feel free to do things manually—all of the behavior the WebActions provide can be replicated using existing Sun ONE Application Framework features, though arguably without the same level of convenience.

Interoperating With Sun ONE Application Framework Applications

Because Sun ONE Application Framework applications are ultimately J2EE Web applications, other Web applications can interoperate with them in well-defined ways. The techniques for interoperation with Sun ONE Application Framework applications depend on how the application components will be accessed.

Interoperating From an External Application

An *external application* is a completely separate application in a different servlet context, application server, Web server, J2EE or non-J2EE container, and so on.

This type of interoperation is the same as it would be for any other Web application—Sun ONE Application Framework applications are invokable through a standard HTTP URL. If you want to pass parameters to a Sun ONE Application Framework application, you can append query parameters to the HTTP invocation and access these from within the invoked Sun ONE Application Framework component.

However, this type of invocation will result in a *first-touch request*, a request that does not invoke a request event handler. In most cases, this is fine. In some cases, though, you might want to simulate a button press or HREF (hyperlink) click. If so, you need to include at least one name-value pair as a query parameter or posted value.

Each request resulting from a Sun ONE Application Framework button or HREF includes a parameter that signals to the application's request handling infrastructure that it should invoke a request event handler. The name of this parameter is the fully-qualified name of the button or HREF. For example, a button name `SubmitButton` on `PgFoo` would generate a parameter like this:

```
PgFoo.SubmitButton=Submit
```

Therefore, to simulate this button press, you would send an HTTP request to the Sun ONE Application Framework application that looks something like this:

```
http://<host>/fooapp/main/PageFoo?PageFoo.SubmitButton=
Submit&...
```

This would typically result in the `handleSubmitButtonRequest()` method being invoked on `PageFoo`. If you want to populate any of the other fields on the same page, as most request event handler methods would expect, you need to append additional parameters to the query string or posted content.

If you want to invoke a request event handler within a child TiledView of a page, you need to also include a row number subscript in the parameter name:

```
PageFoo.FooTiledView[3].SubmitButton=Submit
```

This would invoke the request event handler in the TiledView named `FooTiledView` as if the button press had occurred on the fourth tile (tile numbers are zero-based).

Interoperating From Within the Same Application

In some cases, a single Web application might contain both Sun ONE Application Framework and other J2EE components, such as non-Sun-ONE-Application-Framework servlets and JSPs. The advantage of interoperating with the Sun ONE Application Framework from within the same application is that the application components can share session and other objects via the request and servlet context attributes. This makes it much easier to move seamlessly between components.

In most situations, you can simply follow the same basic guidelines for interoperating from an external application. However, you instead need to forward to or include the Sun ONE Application Framework component in the request using the standard `javax.servlet.RequestDispatcher` mechanism. When forwarding or including in this manner, you must be sure to request the standard Sun ONE Application Framework URL and not any of the Sun ONE Application Framework JSPs directly (the request must proceed via the module servlet to provide the proper Sun ONE Application Framework context):

```
String target="/fooapp/main/PageFoo?PageFoo.SubmitButton=
Submit&...";
RequestDispatcher dispatcher=
    request.getServletContext().getRequestDispatcher(target);
dispatcher.forward(request,response);
```

CHAPTER 4

Deploy an Application

This Sun™ ONE Application Framework *Deploy an Application* chapter covers the preparation of a Sun ONE Application Framework application for deployment in most J2EE containers, as well as deployment-time configuration of a Sun ONE Application Framework application.

It assumes general familiarity with Sun ONE Application Framework, as well as Web Application Archive (WAR) files, deployment descriptors, and your J2EE container's specific deployment method.

Note – Sun ONE Application Framework applications can be run in any Servlet 2.2/JSP 1.1-compliant (J2EE 1.2) Web container.

Configure the Application

In addition to packaging a Sun ONE Application Framework application, it must be configured before it can be deployed.

If you are using the Sun ONE Application Framework IDE toolset, all of the following configuration is automatically managed for you.

This section is intended to provide information to developers who want to create Sun ONE Application Framework applications or objects by hand, or for developers who want to understand the details of how the framework operates.

Module Servlet Configuration

General Configuration

The Sun ONE Application Framework servlet infrastructure established by `ApplicationServletBase` includes the ability to configure arbitrary properties on each module servlet using reflection. Parameters are specified in the application deployment descriptor (`web.xml`) as either context or servlet init parameters using a special name format:

```
jato:<class name expression>:<param name>
```

For example:

```
<context-param>
    <param-name>jato:fooapp.main.MainModuleServlet:foo</param-name>
    <param-value>bar</param-value>
</context-param>
```

The specified class name might contain an asterisk wild card character to allow parameters to be set on more than one object:

```
<context-param>
    <param-name>jato:fooapp.*:foo</param-name>
    <param-value>bar</param-value>
</context-param>
```

Each parameter might only contain a single asterisk, and will match any classes whose class name matches the string before and/or after the asterisk.

The class name expression might also be omitted if the parameter should apply to all module servlets:

```
<context-param>
    <param-name>jato:enabledLogLevels</param-name>
    <param-value>ALL</param-value>
</context-param>
```

To have a parameter set on it, the module servlet class must have a setter method that conforms to the JavaBeans method naming convention. For example, a parameter named *foo* would cause the servlet to call a method called `setFoo()` to set the parameter value. The value is converted to the appropriate type for the setter method. If the type cannot be converted, an error message is written to the servlet context indicating the exception.

The following table shows the module servlet parameters that are currently available.

Parameter Name	Type	Description	Required
allowShortViewBeanNames	`boolean`	Versions of Sun ONE Application Framework prior to 2.1 required that ViewBean classes be named with a *ViewBean* name suffix, for example, *FooViewBean*. Beginning with version 2.1, ViewBeans do not require this suffix. The mechanism that looks up ViewBeans from the request's path info must know whether to enforce the *ViewBean* suffix or not. By default, this parameter is false by default for backward compatibility, but newer application can enable this feature to avoid this requirement. **Note:** It is possible to set this parameter on a per-module basis, so parts of the application using the older and newer naming conventions can coexist.	No

Parameter Name	Type	Description	Required
enabledLogLevels	`String`	A comma-delimited list of log levels that should be enabled. Possible values: NONE MANDATORY STD STANDARD VERBOSE_DEBUG TERSE_DEBUG ANY_DEBUG JATO_TRACE JATO_QOS_TRACE APP_TRACE ANY_TRACE WARNING ERROR CRITICAL ANY_ERROR USER_LEVEL_1 USER_LEVEL_2 USER_LEVEL_3 ANY_USER_LEVEL ALL DEFAULT	No
echoLogToSystemOut	`boolean`	Sets the state of logging to System.out in addition to the servlet context.	No
enforceStrictSessionTimeout	`boolean`	During development of an application, redeploying an application generally times out any sessions associated with it, which can make iterative development difficult. When set to false, this parameter allows session timeouts to avoid causing the user's browser to be unusable for the current application until closed and reopened.	No
generateUniqueURLs	`boolean`	Enables or disables generation of unique URLs during page rendering. Unique URLs help defeat problematic caching strategies of proxies and browsers. For example, if the browser is caching dynamic pages inappropriately, enabling this feature will generally cause the browser to avoid showing inappropriately cached dynamic pages by making it think that every page comes from a unique URL. This feature is disabled by default because it incurs a small runtime cost.	No

Parameter Name	Type	Description	Required
logMessagePrefix	`String`	Sets the log message prefix (used to make log messages stand out better).	No
moduleURL	`String`	The URL for the module servlet. See detailed description below.	Yes
qualifiedViewNameSeparator	`char`	Since the initial version of Sun ONE Application Framework, the period (".") has been used to separate qualified view names. This separator might make it more difficult for developers to use JavaScript in their HTML pages, however, so instead application developers might set an alternative value here, such as underscore ("_"). Developers must take care not to use the separator character in normal child view names.	No
showMessageBuffer	`boolean`	Enables or disables showing the application message buffer at the bottom of the rendered HTML page. This feature is useful for debugging, but should be turned off in deployment.	No
useTaglibTEI	`boolean`	Previous versions of J2EE containers had problems generating Tag Extra Information (TEI) declared in the Sun ONE Application Framework TLD. Beginning with version 2.1, TEI has been enabled by default, but developers with applications running in older container versions might want to disable this feature to maintain compatibility.	No

Module URL Configuration

Each module servlet has an associated *module URL*. This URL is a standard servlet URL path mapping, and must be configured in the application's deployment descriptor (or equivalent). When each page in a Sun ONE Application Framework application is rendered, the Sun ONE Application Framework tag library renders references to the current module's URL into the HTML output, so that submitted forms or activated links return to the module and ViewBean that rendered the original output. Therefore, the module URL for each module must be configured in a standard way that makes it accessible to the Sun ONE Application Framework infrastructure. Additionally, to allow cross-module navigation within the application, the module URLs of each module servlet must be available to each of the other module servlets. Therefore, the module URL is the only module servlet parameter that must be configured before the application can be run.

The moduleURL parameter is configured like other module servlet parameters, but has some additional restrictions, as follows:

- The moduleURL parameter must be only package-specific (must use the asterisk wildcard style of parameter naming)

Because the moduleURL parameter is used also by the ViewBeans in each module, each of which share the same package name, the moduleURL parameter must be named as follows:

```
<param-name>jato:[module package name].*:moduleURL</param-name>
```

- The moduleURL parameter must be configured as a context parameter, not a servlet init parameter

  Whereas other module servlet parameters can be configured as either context parameters or servlet init parameters, the moduleURL parameter must be configured as a context parameter (see below for an example). The reason for this restriction is that the moduleURL parameters for each module servlet must be available to all other module servlets.

- The value of the moduleURL parameter must correspond to the module servlet URL mapping

  The URLs rendered into a Sun ONE Application Framework page are relative URLs. Because of the standard Sun ONE Application Framework URL format, which specifies the page name at the end of the URL path, the moduleURL must be configured as follows:

```
<context-param>
    <param-name>jato:[app package].[module package].*:moduleURL</param-name>
    <param-value>../[module package]</param-value>
</context-param>
```

Similarly, the module servlet URL must be mapped to the following URL:

```
<servlet-mapping>
    <servlet-name>[servlet name]</servlet-name>
    <url-pattern>/[module package]/*</url-pattern>
</servlet-mapping>
```

Technically, the URL path of the module need not be named after the module package—it could be any arbitrary name. Although this is what is recommended for simplicity, the main constraint is that the URL path used in the moduleURL parameter be the same as that used in the servlet mapping URL pattern.

Given the above rules and assuming the following,

- A base application package of `fooapp`
- A module in package `fooapp.main`

a basic application would have the following deployment descriptor:

```
<web-app>

    <context-param>
        <param-name>jato:fooapp.main.*:moduleURL</param-name>
        <param-value>../main</param-value>
    </context-param>

    <servlet>
        <servlet-name>MainModuleServlet</servlet-name>
        <servlet-class>fooapp.main.MainModuleServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>MainModuleServlet</servlet-name>
        <url-pattern>/main/*</url-pattern>
    </servlet-mapping>

...

</web-app>
```

Adding a second module named Module2 to this application would result in the following:

```
<web-app>

    <context-param>
        <param-name>jato:fooapp.main.*:moduleURL</param-name>
        <param-value>../main</param-value>
    </context-param>

    <context-param>
        <param-name>jato:fooapp.module2.*:moduleURL</param-name>
        <param-value>../module2</param-value>
    </context-param>

    <servlet>
        <servlet-name>MainModuleServlet</servlet-name>
        <servlet-class>fooapp.main.MainModuleServlet</servlet-class>
    </servlet>

    <servlet>
        <servlet-name>Module2Servlet</servlet-name>
        <servlet-class>fooapp.module2.Module2Servlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>MainModuleServlet</servlet-name>
        <url-pattern>/main/*</url-pattern>
    </servlet-mapping>

    <servlet-mapping>
        <servlet-name>Module2Servlet</servlet-name>
        <url-pattern>/module2/*</url-pattern>
    </servlet-mapping>

    ...

</web-app>
```

ViewBean Display URL Configuration

Each ViewBean in a Sun ONE Application Framework application has a peer JSP, and the URL of this JSP is referred to as the *display URL* of the ViewBean. Each ViewBean has a default display URL it expects to use, and this URL is available/settable via the `ViewBean.getDefaultDisplayURL()` and `ViewBean.setDefaultDisplayURL(String)` methods. Developers are normally expected to set the default display URL in a ViewBean's constructor.

However, in some cases, it might be useful or necessary to override the default display URL of a ViewBean at application deployment time. In this case, the deployer can set a context parameter in the deployment descriptor that will set the a ViewBean's default display URL at runtime:

Parameter Name	Type	Description
defaultDisplayURL	`String`	The display URL for the specified ViewBean.

This parameter is set using the naming convention established above for module servlet parameters (however, it is not a module servlet parameter):

```
<context-param>
    <param-name>jato:fooapp.module1.FooViewBean:defaultDisplayURL</param-name>
    <param-value>/fooapp/module1/Foo-Alternate.jsp</param-value>
</context-param>
```

Note – Using this mechanism to override ViewBean display URLs is not generally recommended, as it introduces some overhead. Instead, the affected ViewBeans should be recompiled with the appropriate display URL directly, if possible.

SQLConnectionManager Configuration

The Sun ONE Application Framework contains a manager object called `SQLConnectionManager`. An instance of this object will be available to an application from the `RequestContext` on any given request. The purpose of this class is to make the task of obtaining a JDBC `Connection` object easier for developers and the built-in Sun ONE Application Framework objects.

The basic usage of the `SQLConnectionManager` is that callers ask it for a JDBC `Connection` using what is called a *datasource name*. This datasource name is an arbitrary, logical name for a pre-configured database connection. This class also standardizes this task regardless of the technique used to obtain the connection. Finally, the class provides a level of indirection that can be useful when switching between development, test, and deployment environments.

In standard Java applications, JDBC `Connections` are normally obtained via a call to the `java.sql.DriverManager`. Callers provide a JDBC URL specific to the database they want to use, and the `DriverManager` matches an available JDBC

driver with the specified URL, obtains a database connection from the driver, and returns it to the caller. The caller then uses the connection as long as it wants before closing it.

In Web applications, the use of the `DriverManager` is highly inefficient because it requires a new database connection to be opened and initialized for each application request that requires database access. However, the JDBC 2.0 Standard Extension (the `javax.sql` package) defines a standard J2EE mechanism for database connection pooling. This allows connections to be reused over multiple requests, and avoids the inefficiency of repeatedly opening connections to the database.

The JDBC 2.0 Standard Extension provides a JNDI mechanism through which Web application developers can obtain a pooled JDBC Connection object. This mechanism consists of allocating a JNDI context and asking it for a *datasource* by name. This name, also called a datasource name, is of a standard form which begins with `"jdbc/"`. The idea is that instead of embedding JDBC URLs directly in an application, the application deployer pre-configures a JDBC datasource with all the necessary information—host, protocol, username, password, etc.—and makes it available under a logical datasource name that begins with the prefix `"jdbc/"`. Application developers only reference this logical datasource name, which they assume will be mapped appropriately in whatever container the application is deployed.

Unfortunately, not all containers provide this datasource mechanism, and/or they impose certain limitations on the datasource name. For example, one container might allow arbitrary names after the standard `"jdbc/"` prefix, where another container might require the addition of the application context name after the prefix. This might not be a problem with an application developed and deployed in the same container; however, it's fairly common for an application to be developed and deployed under different containers, making this situation a potential problem.

This is where the `SQLConnectionManager` seeks to provide assistance. `SQLConnectionManager` contains its own datasource mapping mechanism, which allows the developer to use truly arbitrary datasource names in their application, yet still allow these names to be operational within any given container. Additionally, the `SQLConnectionManager` allows mapping of datasource names to either JNDI datasource names or plain JDBC URLs. This feature allows a Sun ONE Application Framework application to run in a container that does not support the JDBC 2.0 Standard Extension, albeit without the benefit of connection pooling.

In your application, you can obtain a JDBC connection from SQLConnectionManager directly using its `getConnection()` or `obtainConnection()` methods. The static obtainConnection() method is used to obtain connections outside of the scope of a request, such as during application initialization. You can also bypass `SQLConnectionManager` and go directly to a JNDI lookup (or to `DriverManager`) if you want, but the Sun ONE Application Framework classes that use JDBC, such as `QueryModelBase`, will always use `SQLConnectionManager` to obtain database connections.

Setting the SQLConnectionManager to use JNDI or the JDBC DriverManager

As mentioned above, the `SQLConnectionManager` allows either use of plain JDBC URLs (via the JDBC `DriverManager`) or JNDI datasource names when obtaining a JDBC Connection using an arbitrary datasource name. These two modes are mutually exclusive, and the current mode is selected by calling the `SQLConnectionManagerBase.setUsingJNDI(Boolean)` static method. This method need only be called once; call this method from the static initializer of your application servlet class.

Calling the `setUsingJNDI()` method with a value of `true` will enable JNDI lookups of datasource names. In this mode, all datasource names are assumed to map to JNDI datasource names of the general form `"jdbc/..."`. If the `setUsingJNDI()` method is called with a value of `false`, the `SQLConnectionManager` will assume that datasource names map directly to JDBC URLs, and will attempt to obtain a JDBC `Connection` directly form the JDBC `DriverManager`.

Under no circumstances should you use the `DriverManager` to obtain JDBC connections in a production application! Such usage causes a new database connection to be opened for every use of the connection, and will cause enormous performance and scalability problems with your application. Make sure you use `SQLConnectionManager`'s JNDI lookup mechanism along with a database connection pool in your container when you finally deploy your applications into production.

Add Datasource Mappings

Either of the above modes assumes that a mapping exists for the provided datasource name. These mappings are set by calling the `SQLConnectionManagerBase.addDataSourceMapping(String, String)` static method. This method should generally be called from within the static initializer of the application servlet class.

For example:

```
static
{
    setUsingJNDI(true);

    // Anyone asking for a connection under the name "jdbc/MyDS"
    // would cause the SQLConnectionManager to do a JNDI lookup for
    // a datasource under the name "jdbc/Foo/MyDS-Test"
    addDataSourceMapping("jdbc/MyDS","jdbc/Foo/MyDS-Test");
}
```

or

```
static
{
    setUsingJNDI(false);

    // Note, if we are not using JNDI, we will need to initialize our
    // JDBC drivers in the standard way
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Class.forName("oracle.jdbc.driver.OracleDriver");

    // Anyone asking for a connection under the name "jdbc/MyDS"
    // would cause the SQLConnectionManager to obtain a connection from
    // the JDBC DriverManager with the URL "jdbc:odbc:northwind"
    addDataSourceMapping("jdbc/MyDS","jdbc:odbc:northwind");
    addDataSourceMapping("jdbc/HerDS","jdbc:oracle:thin:@10.0.0.1:1521:foo");
}
```

If a Datasource Mapping Cannot be Found

The `SQLConnectionManager` provides a fallback mechanism if a mapping cannot be found for a given datasource name. If a mapping cannot be found, it assumes that the datasource name is the JNDI datasource name/JDBC URL that should be used to obtain a connection. Therefore, the default mechanism is to use the literal datasource name passed to the `SQLConnectionManager`. This means that unless the developer adds mappings, the use of `SQLConnectionManager` is completely transparent.

Package the Application

Packaging of Sun ONE Application Framework applications follows the standard J2EE guidelines for Web Application aRchive files. If you've developed your application by following this guide's recommendations, you need do nothing more than use a JAR or Zip tool to archive your application directory structure into a WAR file. The WEB-INF directory and its peer directories and files should be at the root of the archive. The name of the WAR file is arbitrary, though it needs to have a `.war` file extension.

Deploy the Application

After configuring the application as described above, deploying a Sun ONE Application Framework application is no different than deploying any other WAR-based J2EE Web application. J2EE containers differ on the details of deploying such an application; refer to your container's deployment documentation for specifics. However, in many cases, you simply need to copy the WAR file to the container's `/webapps` directory.

Some additional notes on deployment:

- Each deployed application needs its own copy of the Sun ONE Application Framework runtime in its WEB-INF/lib directory. In addition to being the only supported configuration, this measure ensures that the application is entirely self-contained and can be immediately installed in any container. It also ensures that Sun ONE Application Framework version changes do not affect multiple applications. If a shared copy of the runtime were used for several applications, all of these applications would have to be upgraded to a new version of the runtime at the same time.
- The application source code should be packaged in the deployed WAR file when possible (under the /WEB-INF directory). This is not a formal recommendation, but it has several advantages. First, it aids troubleshooting efforts by providing an exact image of the source that is actually being used in the application. Second, it ensures that critical fixes can easily be made to deployed applications without requiring a full development/source control environment.
- If you use JDBC in your application, check before it is deployed to ensure that it is using JNDI mode in the SQLConnectionManager.
- You might want to configure your container or Web server to deny access to all `.jsp` files from external clients if possible, since these files are only meant to be accessed through Sun ONE Application Framework servlets.

Access a Sun ONE Application Framework Application

After deploying your Sun ONE Application Framework application, it is available to be accessed by HTTP clients. Sun ONE Application Framework applications use a standard URL format of the following general form:

```
http://<host + container-specific path>/<servlet context
name>/<module name>/<page name>
```

where:

- The servlet context name is the name of the deployed WAR file (in most J2EE containers).
- The module name is the name of the module package within the application.
- The page name is the name of the page to display within that module.

For example, given the following:

- WAR file name: `HelloWorld.war`
- Module name: `greeting`
- Page name: `Hello`
  - ViewBean class: <*app package*>`.greeting.Hello`
  - JSP name: `Hello.jsp`

the URL to access the Hello page would be the following:

```
http://<host + container-specific
path>/HelloWorld/greeting/Hello
```

The base application package name does not factor into the URL. Additionally, the requested page name must exist within the specified module. Trying to access a page outside of the requested module is illegal. For more information, see "Cross-Module Navigation" on page 98.

Many J2EE containers do not impose container-specific paths for access to the servlet engine. For example, Sun ONE Application Server, Apache Tomcat, or Caucho Resin would use the following URL:

```
http://<host>/HelloWorld/greeting/Hello
```

However, because some J2EE containers use a multi-tiered architecture with a conventional Web server for external access, the URL might require additional path information. For example, the example URL in iPlanet Application Server 6.x would be this:

```
http://<host>/NASApp/HelloWorld/greeting/Hello
```

Cross-Module Navigation

Cross-module navigation refers to handling a request in one module but responding with a page from another module. This scenario normally arises when moving from one logically related area of the application to another.

For the most part, the Sun ONE Application Framework runtime manages this switch automatically for you, but it is important to understand the basic mechanics. The crucial task in moving across modules is in making sure that the page ultimately rendered to the client contains references back to the module in which that page is contained. Otherwise, a request for a page in a different module would be sent back to the server, causing a security exception. Once a request crosses module boundaries, the target module's servlet will be the next servlet to handle a request from the user. For example, if a user request triggered an event in `PageOne` of `module1`, and this event forwarded the request to `PageTwo` in `module2`, `module2`'s module servlet would be the servlet to handle the next and all subsequent user requests (until another request crossed another module boundary).

There is one subtlety of which developers should be aware when crossing module boundaries within their applications. Normally, developers use the `ViewBeanManager` available from the `RequestContext` to obtain ViewBean references. The `ViewBeanManager` allows developers to provide a short name for the `ViewBean` they want to retrieve via the `getLocalViewBean()` method. However, this method must prepend a package name to the provided name to derive a class name. This package name is always the package name of the module servlet which handled the request. Therefore, it is not possible to obtain a `ViewBean` from another module using this method; only ViewBeans within the current module are available through this shortcut method. To obtain a reference to a ViewBean in another module, use one of the other `ViewBeanManager` methods that expects a class or fully-qualified class name.

APPENDIX **A**

Troubleshooting

This Sun™ ONE Application Framework *Troubleshooting* Appendix A outlines known Troubleshooting issues including symptom, probable cause, probable solution, and comments for each known issue.

Symptom

```
javax.servlet.ServletException: Invalid request - request
handler "X" not found at
com.iplanet.jato.ApplicationServletBase.onRequestHandlerNotFound
(...)
...
```

Probable Cause

Misspelled the target page.

Probable Solution

In this example, the target page is spelled *Login*, not *login*.

Case sensitivity and plural form are common mistakes.

Comments

Because page names are always related to ViewBean class name, they generally begin with an uppercase letter.

Symptom

```
javax.servlet.ServletException: The request context is null -
this page must be accessed through a servlet at
org.apache.jasper.runtime.PageContextImpl.handlePageException(Pa
geContextImpl.java:457)
...
```

Probable Cause

Attempting to access a Sun ONE Application Framework JSP directly using the JSP's URL.

Probable Solution

All access to Sun ONE Application Framework Web applications must go through a module servlet (the front controller). This exception indicates that the user tried to access the wrong resource.

Instead of this URL:

```
http://localhost:8081/JatoTutorial/jatotutorial/main/Login
```

use a URL like this:

```
http://localhost:8081/JatoTutorial/main/Login
```

Comments

The actual valid URL might vary depending on how the URL/servlet mappings are configured in the application's `web.xml` file. If the application was created without manual modification of the `web.xml`, the URL pattern should be as follows:

```
http://<server>:<port>/<ServletContext>/<module>/<targetPageName
>
```

Symptom

```
com.iplanet.jato.NavigationException: Exception encountered
during forward
Root cause = [java.sql.SQLException: Cannot find the user
"ADMINISTRATOR".]
...
```

Probable Cause

This is specifically a database access SQL exception. In this case, the model attempted to access the database without the proper username and password.

Probable Solution

If you are using JDBC URLs (rather than the JNDI lookup technique), provide the appropriate username and password by setting them explicitly in the Model's constructor (PointBase sample database example):

```
setDefaultConnectionUser("pbpublic");
setDefaultConnectionPassword("pbpublic");
```

In some cases, you can also provide the user name and password on the JDBC URL.

Comments

This problem can be avoided by using JNDI database connection lookup. See "Using SQLConnectionManager" on page 58 in Chapter 3, Programming Guide.

Index

R

S

T

U

V

W