Overview

Sun ONE Application Framework

Version 2.0

817-0447-10
September 2002

Products covered by and information contained in this service manual are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

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.

Les produits qui font l'objet de ce manuel d'entretien et les informations qu'il contient sont régis par la législation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes biologiques et chimiques ou du nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des États-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régi par la législation américaine en matière de contrôle des exportations ("U.S. Commerce Department's Table of Denial Orders "et la liste de ressortissants spécifiquement désignés (" U.S. Treasury Department of Specially Designated Nationals and Blocked Persons "),, sont rigoureusement interdites.

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

About This Document

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.

This preface addresses the following topics:

- Who Should Use This Guide
- Using the Documentation
- How This Guide Is Organized
- Related Information
- Documentation Conventions
- Product Support

Who Should Use This Guide

The intended audience for this guide is the developer who is at least somewhat familiar with building Web applications using existing J2EE Web Technologies (servlets and JSPs), but new to building Web applications with the Application Framework.

This guide assumes you are familiar with software development processes, including debugging and source code control.

Using the Documentation

The Sun ONE Application Framework manuals are available as online files in Portable Document Format (PDF) and Hypertext Markup Language (HTML) formats, at:

`http://docs.sun.com/`

The following Sun ONE Application Framework Documentation Roadmap table lists concepts described in the Sun ONE Application Framework documentation. The left column lists the concepts, and the right column lists the corresponding documents.

For information about	See the following
Late-breaking information about the Sun ONE Application Framework software and documentation	Release Notes
An introduction to the Sun ONE Application Framework Web application and discussion of developmental issues of a J2EE Web application	Sun ONE Application Framework Overview
An introduction to the user of the Sun ONE Application Framework and the Forte tool plugin, with a description of the mechanics and techniques used to build, deploy and test a Web application using this powerful tool	Sun ONE Application Framework Getting Started
A description of the environment and steps required to install the Sun ONE Application Framework within the Sun ONE Studio	Sun ONE Framework Installation Guide
Wizard-based concepts and components	Sun ONE Application Framework Online Help (installed with the product)

How This Guide Is Organized

This guide contains the following documentation components:

- Sun ONE Application Framework Overview

- Sun ONE Application Framework Design and Architecture FAQ

Related Information

In addition to the information in the Sun ONE Application Framework documentation collection listed in Using the Documentation, the following resources may be helpful:

- J2EE Specifications

  `http://java.sun.com/products/`

- Enterprise JavaBeans Specification, Version 2.0

  `http://java.sun.com/products/ejb/docs.html#specs`

- General EJB product information:

  `http://java.sun.com/products/ejb`

- Java Software tutorials:

  `http://java.sun.com/j2ee/docs.html`

- *Enterprise JavaBeans*, by Richard Monson-Haefel, O'Reilly Publishing, ISBN 0-596-00226-2

  `http://www.oreilly.com/catalog/entjbeans3/`

- *Enterprise JavaBeans Design Patterns*, ISBN 0-471-20831-0

- *Core J2EE Patterns*, ISBN 0-13-064884-1

Documentation Conventions

This section describes the types of conventions used throughout this guide:

- General Conventions
- Conventions Referring to Directories

General Conventions

The following general conventions are used in this guide:

- **File and directory paths** are given in UNIX® format (with forward slashes separating directory names). For Windows versions, the directory paths are the same, except that backslashes are used to separate directories.
- **URLs** are given in the format:

  http://*server.domain*/*path*/*file*.html

  In these URLs, *server* is the server name where applications are run; *domain* is your Internet domain name; *path* is the server's directory structure; and *file* is an individual filename. Italic items in URLs are placeholders.
- **Font conventions** include:
  - The `monospace` font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
  - *Italic* type is used for code variables, book titles, emphasis, variables and placeholders, and words used in the literal sense.
  - **Bold** type is used as either a paragraph lead-in or to indicate words used in the literal sense.
- **Installation root directories** for most platforms are indicated by *install_dir* in this document. Exceptions are noted in Conventions Referring to Directories.

  By default, the location of *install_dir* on **most** platforms is:
  - Solaris 8 non-package-based Evaluation installations:

    `user's home directory/sun/appserver7`
  - Solaris unbundled, non-evaluation installations:

    `/opt/SUNWappserver7`
  - Windows, all installations:

    `C:\Sun\AppServer7`

  For the platforms listed above, *default_config_dir* and *install_config_dir* are identical to *install_dir*. See Conventions Referring to Directories for exceptions and additional information.
- **Instance root directories** are indicated by *instance_dir* in this document, which is an abbreviation for the following:

  default_config_dir`/domains/`*domain*/*instance*
- **UNIX-specific descriptions** throughout this manual apply to the Linux operating system as well, except where Linux is specifically mentioned.

Conventions Referring to Directories

By default, when using the Solaris 8 and 9 package-based installation and the Solaris 9 bundled installation, the application server files are spread across several root directories. These directories are described in this section.

- **For Solaris 9, 12/02, bundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/usr/appserver/`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
  - *default_config_dir* refers to `/var/appserver/domains`, which is the default location for any domains that are created.
  - *install_config_dir* refers to `/etc/appserver/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.
- **For Solaris 8 and 9 package-based, non-evaluation, unbundled installations**, this guide uses the following document conventions to correspond to the various default installation directories provided:
  - *install_dir* refers to `/opt/SUNWappserver7`, which contains the static portion of the installation image. All utilities, executables, and libraries that make up the application server reside in this location.
  - *default_config_dir* refers to `/var/opt/SUNWappserver7/domains`which is the default location for any domains that are created.
  - *install_config_dir* refers to `/etc/opt/SUNWappserver7/config`, which contains installation-wide configuration information such as licenses and the master list of administrative domains configured for this installation.

Product Support

If you have problems with your system, contact customer support using one of the following mechanisms:

- The online support web site at:

  `http://www.sun.com/supporttraining/`

- The telephone dispatch number associated with your maintenance contract

Please have the following information available prior to contacting support. This helps to ensure that our support staff can best assist you in resolving problems:

- Description of the problem, including the situation where the problem occurs and its impact on your operation
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem
- Detailed steps on the methods you have used to reproduce the problem
- Any error logs or core dumps

Chapter 1

Sun ONE Application Framework Overview

This document provides an overview of the Sun™ ONE Application Framework and includes the following sections:

- Introduction: The Challenges of Building Web Applications
- What is the Sun ONE Application Framework?
- How Does the Sun ONE Application Framework Work?
- How is the Sun ONE Application Different From Other Web Application Frameworks?
- Conclusion

Introduction: The Challenges of Building Web Applications

Building Web Applications: Pre-J2EE

J2EE, and in particular its Web-tier components (Servlets and JSPs), has been so successful because it addresses the core frustrations of first-generation Web developers. In the pre-J2EE world, these developers had to contend with vastly different programming models, APIs, and server eccentricities just to build simple applications. Enterprise scale applications were all the more difficult because so many factors had to be considered just to select a technology that *might* support an application's requirements. Actually building an application was an additional problem, complicated by platform immaturity, API mismatches, cross-platform integration issues, and lack of highly-scalable development and maintenance models.

Although server vendors solved many of the programming and development scalability issues by providing powerful, high-level application frameworks, these frameworks shared only few basic assumptions, and no common contracts or infrastructure. These frameworks were generally tied to the server vendor's proprietary server infrastructure, and while they made it possible to build highly functional, robust enterprise applications, it was not possible to change vendors or easily take advantage of technology provided by other vendors. Moving from one vendor's platform to another was essentially impossible without rewriting the application.

Enterprise architects adopted several strategies to avoid vendor lock-in, the most widespread of which was adding heavy doses of abstraction into the architecture. Although this strategy solved some of the problems—enterprise business objects and processes could be decoupled from Web-container details—it created others. These abstractions added significant, sometimes massive, complexity to the application, and introduced both development and deployment penalties. Debugging this complex infrastructure became exponentially more difficult as architects tried to distance themselves further and further from proprietary APIs. To make matters worse, Web developer skills became less and less reusable, as each new project introduced new architectures and constraints that were incompatible with those they had previously encountered.

Each application was a world unto itself, and there was very little consistency, especially when the foundation application framework did not provide a strong direction for developers and architects alike. And while some constraints were helpful in focusing application development efforts, some frameworks became so high-level that developers had to work around features to accomplish advanced, or in some cases routine, tasks.

Building Web Applications: Post-J2EE

The advent of J2EE solved many of the problems endemic to first-generation Web application development. For the first time, developers could depend upon standard contracts between the container and their application components, and all J2EE-compliant containers were guaranteed to provide the same well-designed API. Architects and developers were liberated from the chaotic mix of proprietary frameworks, APIs, and containers.

However, with freedom came responsibility. Although J2EE is a solid foundation for an application framework, it is not one itself; the J2EE specification avoids recommendations in the application development space. J2EE leaves architects and developers the significant task of designing (or adopting) an application infrastructure that suits their application development needs. J2EE alone cannot suffice.

Emergence of the J2EE Application Framework

To develop real-world applications, especially large-scale enterprise Web applications, developers inevitably find that they must create some kind of framework. The underlying Web-specific platform is already provided by J2EE, but the delta required for actual application development must come from somewhere, and developing a framework internally can be both time-consuming and error-prone.

Thus, an abundance of reusable (and not so reusable) Web application frameworks built on J2EE have appeared, each of which tries to address some range of developer needs. For example, some frameworks focus exclusively on the rendering of data to the client, while others focus on validation of input data. Still others attempt to unify fat client and thin client GUI development.

Because J2EE abolished the de facto architectures used during first-generation Web application development, each J2EE project must now evaluate and choose an application architecture best suited to its requirements. Common concepts and terminology have evolved to assist discussions of these architectures, including such examples as *Type I* & *Type II* servlet architectures, *Service to Workers* delegation, and MVC-based UIs. However, while these concepts are fundamental and important, they are very broad, and applicable to the entire range of applications, from the very small to the very large. Furthermore, none of them really address the *details* of how to build a Web application in a repeatable, maintainable, and scalable way.

To underscore this last point, nearly all contemporary application frameworks claim to use a Type II, Service-to-Workers, and MVC-based architecture, yet these frameworks are strikingly different in implementation, extensibility, and the constraints they place on developers. Knowing the underlying architecture only helps introduce the framework to developers; it has surprisingly little ongoing role in helping the developer learn the framework or even compare it to other frameworks, especially when these frameworks target different application scales.

Contemporary frameworks go beyond the space in which there is sufficient terminology to effectively describe their features. Thus, a much more detailed analysis is necessary to really understand what one framework offers over another, and in particular, what a framework offers in regard to enterprise application development. Simply working from a checklist of features is grossly insufficient.

The Criteria of an Enterprise Application Framework

Reliance on tested and proven architectures is extraordinarily important when building Web applications, more so than in other application development domains. For example, fat client applications are quite forgiving in their response to sub-optimal architectural or technological choices. Even the most heavily abstracted client-side application architecture will run with sufficient performance when it has a modern workstation all to itself.

The same is not at all true when that same application has to be run over a loosely coupled network, on shared hardware which is supporting hundreds or thousands of simultaneous users. In this domain, the wrong choice of architecture or technology can make the difference between an application that responds in a timely manner to requests, and one that doesn’t respond at all, or one that scales well in both development and production, versus one that may only do one or the other, but not both.

Failings in some of these factors may be forgivable in the small-to-medium application development space, where a handful of developers work closely on an application. In such cases, architectures can be much more fluid because the scope of future changes and contact with other teams is limited, and team members can easily coordinate to tweak issues as they arise. Coding standards and best practices are easily shared ad hoc, and retrofitting older portions of the application is a matter of a few hours of work. Teams tend to have similar levels of expertise, remain stable, and stay together for an extended period. Performance is usually not a critical factor, as the user base for these applications is small or forgiving of slower response times.

Development of enterprise applications is the antithesis of the small-to-medium application development model. Team communication is unwieldy; application changes have massive ripple effects; best practices are seldom disseminated outside of sub-teams; retrofitting parts of the application becomes simply impossible; developer turnover is high; and developer expertise varies widely. Finally, performance is absolutely critical, since it may play a determining role in whether a user will continue to use the application or abandon it for a perhaps less convenient, but more responsive, alternative.

Any framework facilitating enterprise Web application development must then account for these constraints in both its design and implementation. More importantly, it must minimize—and in ideal cases, eliminate—the impact of these constraints on the application development effort.

Therefore, any enterprise framework must do the following:

- Provide application consistency, so that developer skills can be reused across teams, across projects, and across companies. There should be an obvious starting approach to developing an application, but this approach must not limit developer capability.

- Provide both high- and low-level features, so that teams can find the right balance to suit their requirements and complete a project within time constraints.

- Provide concrete ways of increasing application maintainability, so that architects and developers don't have to do this work themselves (conceivably in any number of different ways).

- Guide the naïve developer, since not all developers are created equal, and some may not have experience in the Web-based application domain, or even the enterprise development domain.

- Complement the advanced developer, so that he or she can create advanced features by using the underlying J2EE platform directly, without hacking around framework features.

- Appeal to the enterprise architect, so that the advanced architectural elements he or she wants to incorporate are readily available in the framework or compatible with the framework. The enterprise architect should be comfortable that the framework makes prudent architectural choices, preferably the same ones he or she would make given the resources.

Most importantly, the framework must be proven, mature, robust, and well-performing in an enterprise setting. The users of the framework must know what to expect, and be confident that the framework meets these requirements before beginning any development work.

The remainder of this document describes how the Sun ONE Application Framework, with its goal of being a truly enterprise-class Web application framework, addresses these issues and meets these criteria.

What is the Sun ONE Application Framework?

Overview

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.

The Sun ONE Application Framework is based upon the collective experience of industry-leading software engineers, consultants, Web application developers, and enterprise Web architects. It has been in development since January 2000, and available to customers since June 2000. Since that time, the Sun ONE Application Framework has been used in dozens of real-world enterprise Web applications, and is being used successfully in production sites supporting millions of users, and millions of dollars in financial transactions every day.

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 wishing to provide off-the-shelf components that can be easily integrated into Web applications. These same features make the Sun ONE Application Framework very 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 on 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 as if they were native components. Components can be arbitrarily combined and reused throughout an application, across applications, and across projects and companies.

Finally, 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.

How Does the Sun ONE Application Framework Work?

Use of Design Patterns

The Sun ONE Application Framework is based on state-of-the-art design patterns and techniques, like Model-View-Controller (MVC), Service-to-Workers, Hierarchical View, Command, Business Delegate, and others. Furthermore, the Sun ONE Application Framework is based on an N-tier JSP / Servlet architecture. The Sun ONE Application Framework has been designed entirely around interfaces and object contracts that reflect these patterns—it is an integrated set of cooperating design patterns first, and an implementation of those patterns second.

Primary among these patterns is the Model-View-Controller (MVC) pattern. Where other frameworks claim to be MVC frameworks, the reality is that they usually focus on one or maybe two of the pattern tiers, but seldom on all three. Furthermore, other frameworks often claim that JSPs, perhaps with a custom tag library, comprise a full and proper View tier. They many times also claim that application-specific business objects comprise a full and proper Model tier.

These claims are specious.

The Sun ONE Application Framework addresses all three tiers of the MVC pattern. It defines formal View and Model entities with concrete relationships, and provides an advanced logical Controller role that allows applications to scope controller logic in appropriate ways. The Sun ONE Application Framework's View tier incorporates JSP technology, but is not synonymous with it. In the same way, the Sun ONE Application Framework's Model tier incorporates other J2EE technologies, but is not synonymous with any of them. For these and other reasons explained below, the Sun ONE Application Framework provides unprecedented extensibility for developers that other frameworks simply cannot match.

Types of Functionality

There are three logical groupings of the Sun ONE Application Framework functionality:

- Sun ONE Application Framework Core
- Sun ONE Application Framework Components
- Sun ONE Application Framework Extensions

Sun ONE Application Framework Core

The Sun ONE Application Framework core is what is usually referred to as simply *The Sun ONE Application Framework*. It defines fundamental interfaces, object contracts, and primitives, as well as the minimal infrastructure required for the Sun ONE Application Framework applications. The Sun ONE Application Framework core does not provide a component library, but provides the enabling technology for component authors. Included in the Sun ONE Application Framework core are View-based primitives like ContainerViews, TiledViews, and TreeViews, as well as Model-based primitives like DatasetModels, QueryModels, and TreeModels. The Sun ONE Application Framework core also provides primitives for request dispatching and reusable Command objects. Using these primitives, developers can easily create application-specific or reusable components that can be shared within or across projects. The Sun ONE Application Framework core also includes high-level features that allow developers to immediately begin building highly functional applications. These features are covered in more detail in the sections below.

Sun ONE Application Framework Components

The Sun ONE Application Framework components leverage the Sun ONE Application Framework core infrastructure to provide high-level, reusable components for application development. These components can come in a variety of flavors intended for different usage scopes. For example, horizontal Sun ONE Application Framework components will tend to be the most generic components available, with their strength being flexibility and customizability. These types of components will be usable by many different Sun ONE Application Framework user populations, across projects and companies, and generally will not be biased toward any particular look and feel. Vertical Sun ONE Application Framework components are tailored to a particular usage scenario, allowing them to provide high-level features and high ease-of-use. These types of components will be less broadly usable, but because their scope is better defined, they can keep parameterization to a minimum and use a particular look and feel. All Sun ONE Application Framework components can use all of the facilities provided by the Sun ONE Application Framework core, and build upon its high-level features like WebActions, SQL-based Model implementations, and TreeViews.

Sun ONE Application Framework Extensions

Finally, the Sun ONE Application Framework extensions provide access to non-J2EE facilities in a Sun ONE Application Framework-compatible way. In many cases, the Sun ONE Application Framework extensions allow container-specific features to be used from the Sun ONE Application Framework applications seamlessly. Extensions differ from the Sun ONE Application Framework components in that they focus on technology integration rather than application development.

Technical Overview

The Sun ONE Application Framework, or more properly the Sun ONE Application Framework core, is pure Java, and comes packaged as an industry-standard JAR file.

The Sun ONE Application Framework defines several top-level packages:

- `com.iplanet.jato`—Request handling infrastructure
- `com.iplanet.jato.command`—Command-related interfaces and implementations
- `com.iplanet.jato.taglib`—Custom JSP tag library
- `com.iplanet.jato.model`—General Model-related interfaces and implementations
- `com.iplanet.jato.view`—General View-related interfaces and implementations

Each of these packages contains subpackages of more specific derivations, such as HTML-specific View implementations, and SQL-specific Model implementations. There are no formal packages or classes for the Sun ONE Application Framework components or the Sun ONE Application Framework extensions, which are purely logical classifications.

In writing a Sun ONE Application Framework application, developers derive application-specific subclasses from existing Sun ONE Application Framework classes, or implement certain Sun ONE Application Framework interfaces in an application-specific way. In most cases, developers will use the existing Sun ONE Application Framework core implementations as superclasses, thus inheriting a great deal of useful behavior. (Component developers may be more likely to implement a set of Sun ONE Application Framework interfaces directly.)

Application objects are organized around the central concept of a *page*. Each page consists of a *rendering specification*—normally a JSP containing static content and markup plus custom Sun ONE Application Framework tags—and one class comprising the root of the page's View hierarchy. Each request to the server returns a page as the result. The page flow through an application is determined by the control logic written by the developer. There is no fixed relationship between one page and another other than that provided by the developer.

In the HTML world, each rendered page generally contains one or more links or buttons which the user can *activate*. Each activation of a link or button sends data back to the server, and results in invocation of a Command object specific to that activation. This Command object can take action itself, or delegate handling of the request to developer-defined event methods. Ultimately, the request is forwarded to a resource that is responsible for rendering a response to the client.

In most cases, this resource is an HTML-based JSP page which uses the Sun ONE Application Framework tag library to render dynamic content. The tag library uses the Sun ONE Application Framework View components to obtain the data it renders. These View objects are associated with one or more Model objects, and draw data from them as needed. Thus, the Sun ONE Application Framework Views act as a hierarchical facade to any number of Models. These Views can be reused across multiple pages and with different Models. Models can generally be used by any number of Views since they have no display or View dependencies.

Once the user receives a response in the form of a page, he or she activates a link or button and a request is sent back to the Sun ONE Application Framework application. The request is sent back to the same objects that rendered the page. This allows the Sun ONE Application Framework infrastructure to map the submitted data back into the same Views (and thus Models) from which it originated, providing virtual persistence of this data. The developer interacts with his or her application objects and the submitted data as if there had never been an intervening response-request cycle. Once the data has been mapped back into the originating objects, the Command object specific to that link or button press is activated, and the cycle begins again.

How is the Sun ONE Application Different From Other Web Application Frameworks?

The following sections outline some major differences between the Sun ONE Application Framework and other contemporary Web application frameworks.

Based on J2EE Standards

Many frameworks adopt servlets as a viable technology while eschewing JSP, or vice versa. Still others say they adopt these standards, but in reality, they are merely proprietary containers that can be run in a J2EE container using rudimentary servlet-level integration.

The Sun ONE Application Framework embraces J2EE standards like servlets and JSPs directly, while still allowing developers to freely use the features J2EE provides. The Sun ONE Application Framework is not a container within a container, nor is it a layer meant to abstract the developer from J2EE. Instead, it adds to J2EE features that facilitate enterprise Web application development, while still letting developers interact with as much as or as little J2EE/Sun ONE Application Framework as they like.

A Familiar Paradigm

The Sun ONE Application Framework provides display fields, application events, component hierarchies, and a page-centric development approach, all of which are time-tested and very comfortable to developers familiar with client-side application development using Swing, Delphi, Visual Basic, or PowerBuilder. While there are differences due to the Web paradigm, these familiar constructs lend a natural feel to the Sun ONE Application Framework for these developers, and significantly speed application development. They also mean that the Sun ONE Application Framework is particularly well-suited to integration with application builders, such as Forte for Java or JBuilder (for more information on the topic of application builder integration, see the Tool-ready section).

Application Consistency

Many contemporary Web application frameworks are extremely flexible, and in some cases, this is the fundamental intent of the design. They consciously strive to be non-prescriptive about certain aspects of an application, like its Model tier. Instead, they focus on one or two areas of application design, most commonly the Controller and View portions of an MVC architecture, and leave the rest to the developer.

Some architects and developers might argue that flexibility is never a drawback, but when considering enterprise development, it certainly can be. While it may initially sound strange to characterize flexibility as a drawback, there is an inverse relationship between flexibility and application consistency. A framework that is maximally flexible, like the J2EE API itself, leads to applications that vary widely in the way in which they are developed.

Unlimited flexibility, or an ill-defined development direction, leave inexperienced architects and developers to discover some technique—any technique—that seems to accomplish the task at hand, even if this technique is ultimately flawed. When a framework fails to provide at least one clear path to follow throughout the full range of development tasks, developers are as likely as not to use a technique that sabotages or offsets the advantages the framework provides. Furthermore, each isolated team will likely find a different technique to use, so that even within the same application, one group cannot easily understand or maintain the work of another group. In the worst case, a flawed technique in one portion of the application will undermine the rest of the application to such a degree that the application suffers performance or scalability issues. This situation easily arises when an inexperienced architect or lead developer chooses a poor global direction for the application. Such a choice may result in intractable architectural issues throughout the application, in the worst case rendering the application ultimately unworkable.

Unfortunately for Web application developers, it turns out that most frameworks are flexible in ways that can easily be counterproductive, in both development and production. As we noted above, a good enterprise framework should guide naïve developers in a positive direction without getting in the way of advanced developers. Although many frameworks achieve the latter, they only do so because they are non-prescriptive about certain aspects of the architecture or application development, either because of design philosophy or due to a design flaw. This leaves the developer to make many choices when starting a project, including many which present significant danger to the overall project if improperly selected.

The Sun ONE Application Framework, by contrast, provides an implicit, proven direction for both Web application architecture and application development, without precluding the use of other approaches. It does this by providing well-defined points of interaction with an application, as well as clearly defined ways in which to extend, augment, or override existing behavior. The difference, then, between using the Sun ONE Application Framework and another framework to develop a Web application, is that someone new to the Sun ONE Application Framework need not make a (potentially bad) choice in order to get started. He or she can see from the outset a general approach, and as he or she becomes more advanced and fluent in the Sun ONE Application Framework and J2EE, other approaches and techniques become apparent. Furthermore, whatever work the developer has done up to that point will still be consistent with more advanced techniques used later. As a result, applications written in the Sun ONE Application Framework will resemble one another more so than applications written using other frameworks. They will be more consistent, both in use of high-level and low-level features, and thus be more maintainable.

Symmetrical Display/Submit Handling

Many contemporary application frameworks evolved from custom tag libraries, a very well-received and popular technology. In some cases, they are little more than a custom tag library and perhaps one or two additional interfaces. As a result, these frameworks are myopic in that they are heavily biased toward the display of data *to* the user, but provide little assistance for handling data *from* the user.

These frameworks perhaps address one set of developer needs well, but at the expense of others. In a Type II architecture, rendering technologies like JSP have zero involvement during the submit (request) cycle. This means that if the View representation is defined only in a JSP, the submit-cycle logic cannot take advantage of it. This logic instead just receives a raw list of parameters as inputs. Developers are then left to use these values in their raw form, with little or no assistance. They have suddenly stepped off the deep end into the most basic servlet techniques.

Frequently, in these frameworks, clear relationships between application objects are unspecified and hard to maintain. Because these frameworks provide little or no structure for incoming data, invoked components are forced to work *in the dark*, not able to reliably know what data they are receiving on any given request invocation. This may place a burden on the project that may not be readily apparent when the project is started, but quickly becomes a major factor as the application grows.

The lack of symmetrical display and submit cycles commonly leads to a proliferation of inter-object dependencies. Generally, this proliferation of relationships is reflected in a proliferation of low-level controller logic necessary to do nothing more than manually shuffle input data to a target object or backend. This can lead to an asymmetric notion of a backend object or model being used to render a page, but not used directly when handling a request from a previously rendered page. This asymmetry places yet more burden on developers to micro-manage backend components and concern themselves with the low-level details of running in a Web application container.

Productivity and maintenance are the casualties of a display-centric architecture. By contrast, the Sun ONE Application Framework assists with both the display and submit cycles in a symmetrical fashion, by virtue of its formal View tier. Whereas other frameworks loosely define their View tier as a JSP or some other kind of content rendering technology, the Sun ONE Application Framework makes a distinction between rendering specification (JSP) and View components. Only together are these considered the full View tier. A Sun ONE Application Framework application defines primarily a hierarchy of View components, and then references these components from the rendering specification. The developer interacts with these View components in the same way during both display and submit cycles. The View components are the canonical View form.

Formal Model Entity

As noted in the previous section, many frameworks focus heavily on technology to assist display of data to the user. The most common species of this type of framework are those that focus on XML and XPath. Although enticing to developers looking to use the latest *cool* technologies, these frameworks have little to offer the developer during the submit cycle of the application, and frequently require representation of application data in XML or some other display-oriented format. The coercion of application data to a framework-centric representation is burdensome at best, and in some cases, a fatal shortcoming.

Instead, the Sun ONE Application Framework perspective is that the application should be able to represent its data in a View-agnostic way, and provide a formal mechanism for obtaining that data without implying a particular data format. Therefore, the Sun ONE Application Framework provides a formal Model entity that defines a handful of standard methods that all Models must implement. Using an arbitrary, Model-specific key, Model consumers (including the Sun ONE Application Framework Views) can obtain Model data in a standard way, without any assumptions about how that Model internally represents its data.

For this reason, the Sun ONE Application Framework components can interact with any Model in the same way, allowing a different Model to be plugged into the same View. Models become interchangeable, and by this virtue, so does the data they represent. Marshaling of data to a particular format purely for display becomes unnecessary, and the View tier need not understand the specific type of data with which it interacts. Different types of Models can coexist within an application, without the View tier being cognizant of any difference between their native data formats. XML / XPath, JDBC, JDO, and other enterprise data all look the same to a Model consumer, and thus the Sun ONE Application Framework is able to subsume the development approach of any framework concentrating on one of these data formats.

Finally, the interposition of a Model structure on an enterprise-tier resource enforces a level of abstraction that not only makes the application design far more consistent, but significantly eases the burden of maintenance. In formally defining the data available from the enterprise tier, developers also define a formal yet loosely-coupled contract between tiers of the application. This contract allows the application to be easily modified in the future, and in a well-defined way. The incidence of regressions is lower, and regressions are more readily apparent if they occur.

Application Events

The Sun ONE Application Framework provides developers with a number of events for application-related occurrences. There are three types of events: general request events, specific request events, and display events.

General request events include events like `onBeforeRequest()`, `onSessionTimeout()`, and `onUncaughtException()`. Developers can use these events to respond to general application and request lifecycle occurrences as well as error conditions. By default, error-related events use a consistent, localized mechanism to report errors to users, and can be overridden by developers to take application-specific action.

Specific request events occur based on user action. When a user activates a link or button (also known as a CommandField in the Sun ONE Application Framework) on a page, the request results in the invocation of a Command object on the server corresponding to that activation. Although users can provide their own Command objects in response to such actions, the default Command implementation delegates handling of the request to a request handling event method of the form `handle`<*name*>`Request()`, where <*name*> is the name of the CommandField the user activated. This event is invoked on the parent container of the CommandField, and thus is scoped to the component that originally rendered the link or button. Within this event handler, developers can take any action they like, either handling the request as they wish, or delegating the handling of the request to another object.

The main difference between this Sun ONE Application Framework feature and similar request-handling features provided by other Web application frameworks is that the event is invoked on the component to which it pertains, and is fine-grained per link or button. Other frameworks generally provide only one coarse-grained event handler per HTML form, and the developer is left to make that code conditional based on the user’s action. This is both messy and hard to maintain as the set of fields changes. That approach also makes use of modular, self-contained components difficult, because the single event handler must be changed each time a new link or button is added to or removed from the form, regardless of whether it is contained within a component (for more information on this drawback, see the Hierarchical Views and Component Scoping section).

Lastly, the Sun ONE Application Framework provides fine-grained, field-level display events. Display events are invoked during the rendering of a page, and give the developer hooks into the rendering process that simply would not otherwise be possible. From these events, developers can access the tag handlers as well as the JSP page context and output stream. Display events can be used to skip rendering of a field or abort the currently rendering page altogether. They can also be used to tweak the outgoing content rendered by the JSP, providing advanced content filtering capabilities. Furthermore, display events encapsulate display logic pertinent to a component inside that component, thus providing a high degree of reusability for components even if they use advanced rendering techniques.

Most importantly, display events keep Java code or program-like structures out of the JSP. Any kind of programmatic construct in the JSP is generally a maintenance problem, both because it exposes application functionality to the JSP author, and because parallel content must duplicate this functionality in potentially many places (for more information on parallel content, see the Support for Parallel Content section). Although these kinds of features may be a productivity benefit for small- to medium-sized applications requiring little or no significant maintenance, or having a limited lifespan, such applications are not typical in the enterprise. Many frameworks emphasize this kind of application development, and many of their features are targeted to filling out these capabilities with a full range of programmatic constructs that mimic a traditional programming language. By contrast, the Sun ONE Application Framework recognizes and leverages the advantages of JSPs without compromising maintainability or the ability of the application developer to finely control rendering of the JSP.

Hierarchical Views and Component Scoping

Most frameworks use a flat namespace for data field names in an HTML form. This flat namespace severely limits how View components can be combined—for example, two components using a field called *name* cannot be used on the same form. The only resort is to contrive unique names for all fields, globally, throughout all components and forms. Clearly, this workaround will not scale during development, and curtails the development of a global component market.

The situation is even worse for frameworks that rely on tightly-coupled form-object concordance. In this situation, an HTML form corresponds to a Java object, usually a JavaBean, with accessor and mutator methods for each form field. Simple form field names like *name* easily map to Java methods like `getName()` and `setName()`. But, as noted above, developers will seldom be able to use these simple names if they want to employ reusable components, and will instead need to use globally unique, contrived names. Mapping of complex, contrived field names like `com.foo.componentA.name` to Java method names is particularly inelegant. Such names must comply with Java method naming standards, so the only viable options would be `getCom_foo_componentA_name()` and `setCom_foo_componentA_name()`. Workable, perhaps, but less than elegant or maintainable.

To make matters worse, any framework that relies on a single object as a facade for form field names precludes the use of View components altogether—all data used on that page or form must be reflected by a single object interface, regardless of whether portions of that form are used on multiple pages. A developer could create an object, solely for use by a form, which then delegates to other more reusable objects, but this requires tedious and hard-to-maintain data-shuffling code and is not a true component architecture. Furthermore, it requires a compilation step to make changes in the form or page, a critical shortcoming of frameworks that want to work with application builders.

The Sun ONE Application Framework, by contrast, provides a hierarchical namespace for HTML form fields that is not based on tightly-coupled form-object concordance. Each display field View is created separately as a child of a parent container View, and uses a simple local name within that container. It thus implicitly inherits a qualified, unique global name. These qualified field names are guaranteed never to conflict with other field names, even if local names are identical in other containers. Therefore, independent View components can be arbitrarily combined and will never conflict with one another. The Sun ONE Application Framework automatically manages the mapping of form data associated with these qualified field names back into components during the submit cycle, so developers never have to think about how they combine components. They simply use them and the Sun ONE Application Framework takes care of the details.

Furthermore, developers do not use these qualified names during authoring of a JSP page. Instead, the Sun ONE Application Framework provides what are called *context tags*. These tags define nested container and component scopes. Developers use local names in the JSP within these scopes, and these names are automatically and transparently translated to qualified names at runtime using the current context. Not only can View components then be arbitrarily combined, but rendering specification fragments (JSP fragments & pagelets) can be arbitrarily combined in a parent page. The Sun ONE Application Framework developers have then two types of View component reuse at their disposal, and these types can be combined in several permutations. This is simply not possible in other frameworks.

Efficient Object Management

Many frameworks focus very heavily on object reuse within the application, with the intent of being more efficient and scalable because they avoid object allocation. Unfortunately, this approach is today wrongheaded, and has been debunked in several well-know forums. While it may not have been true in the JDK 1.0 timeframe, object allocation in modern JVMs is extremely cheap, especially when compared to process-wide synchronization points. For maximum scalability, a framework must avoid synchronization between concurrent threads as much as possible.

Frameworks that go to lengths to share objects are unknowingly limiting their scalability. Furthermore, they increase their complexity significantly, and require great care to avoid bugs related to multithreading. In many cases, these frameworks also impose multi threaded programming concerns on application developers, who are more often than not unequipped to undertake such tasks. Perhaps scariest is that these bugs may not reveal themselves until the application built on the framework is in production and under heavy load.

For these reasons, the Sun ONE Application Framework takes a pragmatic approach. It reuses objects where it makes sense, but allows other objects to be allocated as needed. The common request handling infrastructure of the Sun ONE Application Framework relies on shared object instances managed by the container, but objects used by the developer during normal request handling are allocated lazily as needed. Not only does this approach reduce complexity and eliminate an entire class of potential bugs for the Sun ONE Application Framework itself, it does the same for application code. Developers need not worry about stomping on shared data, and debugging becomes much easier.

The Sun ONE Application Framework has proven that this approach is maximally effective in production deployments, in which hundreds of requests per second are handled without significant latency or memory effects due to object allocation.

Support for Parallel Content

Most contemporary frameworks provide internationalization support by giving developers access to Java resource bundles. JSP authors replace static content in the JSP with custom tags that instead obtain localized content at runtime from a resource bundle backed by a property file.

While this is a useful approach, it has significant drawbacks when used alone, among these being that the JSP page author cannot author a page in a natural way using a standard HTML editor, but must instead edit content in property files. Furthermore, this approach largely assumes that the markup surrounding the localized content is unchanged, when in reality it may be heavily influenced by the device or language being targeted. Therefore, certain types of internationalization are best addressed using an alternative approach called *parallel content*.

The Sun ONE Application Framework provides full support for parallel content, which is the use of parallel sets of JSPs, with each JSP in the set customized to a particular language, target device, output markup (for example, XML, HTML, or WML), or any combination of these. Each of these JSPs references the same View components, and thus contains only variations of content and markup. The application can then choose the most appropriate JSP to render at runtime based on user preference or any other desired criteria.

Parallel content works very well when trying to localize content for both Western and Asian languages, where page layout may be affected heavily, or when trying to render to different device types like a standard browser or an Internet-enabled cell phone. The advantage is that the business logic and View structure remain consistent across localized versions of the page, while allowing for (sometimes significant) rendering differences.

Some frameworks assume a static association between JSP and application component, or try to automate page flow using a declarative specification of the component-JSP relationship. While this latter approach has its advantages in certain, limited cases (yet many more significant drawbacks), it does not allow the flexibility needed for use of parallel content. Other frameworks that emphasize programmatic constructs in the JSP make the use of parallel content extremely difficult. Developers using these frameworks must copy and maintain programmatic constructs across multiple parallel JSPs. Because the Sun ONE Application Framework provides display events to keep programmatic constructs out of the JSP, display logic never has to be replicated across parallel JSPs in a Sun ONE Application Framework application.

The Sun ONE Application Framework provides full support for parallel content, making it extremely easy for applications to select at runtime a JSP to render based on any developer-defined criteria. The lookup for parallel JSPs is also developer-defined, so parallel content can be organized in a way that makes sense to the application. Together with resource-bundle-based internationalization strategies, the parallel content feature of the Sun ONE Application Framework provides the most flexible internationalization support possible.

Ready-to-Use, High-Level Features

The Sun ONE Application Framework provides not only low-level infrastructure for use by applications and components, but also high-level features that developers can use to rapidly build highly functional applications.

Among these features are WebActions, which allow developers to perform common, high-level tasks with a minimum of code. For example, developers can invoke the Next and Previous WebActions to automatically paginate through rows of data in a DatasetModel, across requests. The dataset position is automatically managed across requests by the WebAction infrastructure, with no additional code necessary from the developer. Any model implementing the DatasetModel interface can be used with these WebActions.

Another high-level feature that the Sun ONE Application Framework provides is a set of SQL-based Model implementations that automatically manage Model-oriented access to JDBC resources. These implementations use SQL queries and stored procedures to retrieve and persist data in an RDBMS, all without the developer worrying about detailed JDBC use or the inconsistencies in JDBC driver usage. Of course, developers can use JDBC directly from within a Sun ONE Application Framework application if they wish, but the presence of these value-added implementations in the Sun ONE Application Framework core allows developers to very rapidly build functional, enterprise applications out of the box.

Other frameworks simply do not provide this level of functionality, out of the box or otherwise. Although developers can use object-relational mapping tools with any framework, including the Sun ONE Application Framework, they minimally require a conscious decision to use complex business objects in the application architecture. By contrast, the Sun ONE Application Framework SQL-based Models allow developers to abstract these details away from the application domain and put them behind a standard Model interface.

The rest of the application is not directly dependent on JDBC or SQL, and thus becomes far more maintainable and consistent.

Finally, the Sun ONE Application Framework provides TreeView and TreeModel primitives that drastically simplify development of hierarchical data displays. These primitives are complemented by a set of look-and-feel-agnostic custom tags, which allow developers to structure a JSP document into portions that will be selectively rendered for a given tree node. Since these tags output no markup themselves, they can be used in JSP fragments and pagelets to provide a pluggable and customizable component look and feel. No other contemporary framework has anything rivaling these components.

Tool-ready

Unlike other frameworks, the Sun ONE Application Framework was designed from the ground up to be ultimately used with GUI application builders to create Web applications. Almost all other contemporary frameworks lack the features that make highly functional application builder integration feasible. Because they define no formal fields, components, or Models, nor provide a page-centric development approach, there is only limited opportunity for manipulation in a GUI builder. Instead, such integration is likely limited to one-way code generation from templates, with nothing but simple manual code editing to follow.

While tool-readiness has been part of its fundamental design, the Sun ONE Application Framework does not yet provide this capability, for several reasons. First, before ever focusing on GUI application development, the framework must be correct, robust, and amenable to API-based manipulation. Developers must be able to do everything they need, including using very advanced techniques, via a well-designed API. Frameworks that focus on tool support from the beginning are generally biased toward only that type of manipulation, and fail to provide a well-designed, easy to use, and flexible API underlying that infrastructure. This means that developers cannot go below the tool-oriented layer to do things like build reusable components, or manipulate application objects in advanced ways.

Furthermore, focus on tool support tends to lead designs in a direction that may incur performance penalties. Perhaps the most common implementation approach leading to performance issues is the use of shared objects. Shared runtime objects are common in application-builder-oriented designs, but because they require significant synchronization at runtime, they ultimately limit scalability. Projects should not need to give up production scalability for development scalability; instead, the two should both be achievable.

The Sun ONE Application Framework perspective is that if the framework is first designed around interfaces and object contracts, GUI builder support will be easy to add at a later stage (and in fact, this work is being pursued today in the Sun ONE Application Framework). What will result will be a framework that provides both development productivity using application builders, but also supports advanced uses that truly make the framework ready for the enterprise. This also means that scalability and performance will not be compromised by a tool-centric design early in the process, and that even if application builder support requires some performance penalty, developers will have the ability to make a conscious choice of development versus production scalability.

Finally, application-builder focused frameworks are usually consumers of additional, sometimes nonstandard, technologies, meaning that they require additional libraries outside their scope that cannot be guaranteed to be the right version, or even available on a given platform. These other technologies must then be made available manually by the project team, sometimes with difficulty because of version clashes. The Sun ONE Application Framework perspective is that it is better to be lean in this regard rather than require additional libraries which might conflict with a project's deployment architecture.

Enterprise-class Performance

Because the Sun ONE Application Framework has been optimized to eliminate *all* synchronization points, applications built on the Sun ONE Application Framework are as scalable as the J2EE container in which they run. The Sun ONE Application Framework introduces only a small, fixed amount of overhead to each application request, whereas other frameworks that do costly synchronization may exact exponentially more overhead or incur increasingly longer latencies as load increases.

Conclusion

The Sun ONE Application Framework provides features that have either no equivalent or no equal in other contemporary Web application frameworks. The vast majority of available frameworks focus on rendering of data, using various technologies like JSP and XML. Only a very few actually attempt to address a significant range of developer needs, and only the Sun ONE Application Framework attempts to address the broad range of enterprise application development requirements. The Sun ONE Application Framework has been designed through and through to complement enterprise development, and minimize the impact of the unique challenges enterprise development presents.

Thus, the Sun ONE Application Framework meets the criteria of an enterprise framework in the following ways:

- The Sun ONE Application Framework provides application consistency by enforcing and encouraging the use of proven, state-of-the-art design patterns. For this reason, developer skills can more easily be reused across teams, across projects, and across companies. The Sun ONE Application Framework offers an obvious, proven approach to developing applications, but also allows low-level interaction with the underlying J2EE and Sun ONE Application Framework platform for advanced developers.
- The Sun ONE Application Framework provides both high- and low-level features, from ready-to-use Model implementations to fundamental extensibility facilities, so that teams can find the right balance of features that suit their requirements.
- The Sun ONE Application Framework provides concrete ways of increasing application maintainability, including a state-of-the-art request dispatching mechanism that eliminates tedious data shuffling; enforcement of consistent design patterns through well-defined object contracts; advanced component development facilities, fine-grained application events and override points; and a page-centric development model.
- The Sun ONE Application Framework guides the naïve developer to create well-designed, high-performance Web applications by offering a clear, understandable, and proven application development approach, using already familiar application development concepts.
- The Sun ONE Application Framework complements the advanced developer by not getting between him or her and the underlying J2EE platform, and by providing a well-defined mechanism for extensibility.
- The Sun ONE Application Framework appeals to the enterprise architect with its enterprise-class design patterns, proven high-performance, and formal mechanisms for abstracting and encapsulating access to enterprise-tier resources.

The Sun ONE Application Framework is mature, robust, stable, and extremely well-performing. But most importantly, the Sun ONE Application Framework is proven. It is being used successfully in production enterprise applications supporting millions of users and millions of dollars in financial transactions every day, and has had ringing endorsements from enterprise developers, architects, and project managers alike. Above and beyond all the other reasons outlined here, the fact that the Sun ONE Application Framework has already been proven in the enterprise most of all assures those adopting the Sun ONE Application Framework that it meets all the criteria of an enterprise-class Web application framework.

Chapter 2

Sun ONE Application Framework Design and Architecture FAQ

This document provides answers to a number of questions often asked by people new to the Sun™ ONE Application Framework about its design and architecture.

The questions included in this document are as follows:

- Who should be interested in the Sun ONE Application Framework?
- Why use the Sun ONE Application Framework when we already have J2EE?
- Isn't the Sun ONE Application Framework just another proprietary Web application framework (JAPWAF)?
- How is the Sun ONE Application Framework different from other J2EE frameworks?
- The Sun ONE Application Framework has the notion of a "display field". This isn’t like the J2EE Blueprints or other J2EE architectures I’ve seen—why not just pull values directly from a helper bean?
- Do the Sun ONE Application Framework applications require the use of EJBs?
- How are the Sun ONE Application Framework applications structured?
- How are the request flow and URL format implemented?
- How does a view bean relate to a session or entity bean?
- With the JSP scope set to "request" to simplify threadsafe coding and force beans to be constructed and destroyed with each request, will there be negative performance impact?

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 who build medium, large, and massive-scale enterprise-strength Web applications. The Sun ONE Application Framework combines robust design patterns with equally robust implementations of those patterns to provide an enterprise Web application foundation. Because the Sun ONE Application Framework provides core facilities for reusable JavaBean-like components, it is also suited for third party developers wishing to provide off-the-shelf components that can easily be integrated into Web applications. These same features make the Sun ONE Application Framework very suitable as a platform for building vertical Web applications or offerings. This is because its horizontal extension capabilities provide a well-defined way for both end users and original developers to extend or leverage existing vertical features.

Why use the Sun ONE Application Framework when we already have J2EE?

J2EE is a relatively young technology, and though it is very exciting, it doesn’t yet provide the richness and rapid development model that some non-J2EE Web technologies have developed over time. This is not necessarily a bad thing—being free of non-standard application APIs is a huge benefit, and the freedom that J2EE provides can make many Web development tasks easier and quicker, and the result more maintainable.

However, because of the ongoing industry need to quickly build richly functional Web applications, and the only minimal level to which J2EE (rightly) specifies such tasks, there is still a need for additional application design patterns and added functionality beyond J2EE for nearly any real-world Web development project, especially enterprise Web applications. This is where the Sun ONE Application Framework steps in: easy to understand and close to the metal, yet it provides unprecedented design flexibility and consistency. Best of all, it is based entirely on the pure, standards-compliant J2EE platform, so you don’t sacrifice J2EE to use the Sun ONE Application Framework; instead, you benefit from both.

Isn’t the Sun ONE Application Framework just another proprietary Web application framework (JAPWAF)?

No. With proprietary Web application frameworks, you are tied not only to the framework API—for which you don't have the source code—but also the vendor's underlying application server platform. If you want one and not the other, you’re out of luck, and moving an application from one vendor's solution to that of another means a rewrite of your application from scratch.

The Sun ONE Application Framework is different. First, the Sun ONE Application Framework is based entirely on the J2EE platform. There is no container-specific code in the core Sun ONE Application Framework classes, which means you can use the Sun ONE Application Framework in your favorite J2EE container, without changes. We've tested the Sun ONE Application Framework in containers such as Apache Tomcat, Caucho Resin, Allaire JRun, the Sun ONE Application Server, the Sun ONE Web Server, and IBM WebSphere, and it works the same in all of them (barring container bugs, which may necessitate slightly different usage). As far as we're concerned, J2EE has delivered, and we take full advantage of that fact, with you as the beneficiary.

Second, you have the full Sun ONE Application Framework source code. This means that should you wish, you can investigate, change, fix, tweak, configure, or otherwise mess with every nook and cranny of the framework, including even the underlying design pattern, to make it suit your exact needs (although we hope this won't be necessary). We want you to see how the Sun ONE Application Framework works, not only because it will help get bugs fixed more quickly and easily, but because we believe that the Sun ONE Application Framework can only benefit from detailed technical review and discussion.

Finally, the Sun ONE Application Framework is fundamentally a design pattern, based entirely on interfaces and object contracts. We provide a default implementation of these interfaces, but if you don't like the way this implementation works, you can override the portions you don't like. Or, reimplement the interfaces yourself to create a new type of Sun ONE Application Framework object that integrates seamlessly into the rest of the framework. Your new Sun ONE Application Framework objects can interact with other types of Sun ONE Application Framework objects because they all obey the same contracts. Try that in your favorite proprietary Web application framework.

How is the Sun ONE Application Framework different from other J2EE frameworks?

In our survey of J2EE frameworks, both before and after the inception of the Sun ONE Application Framework, we've found that other J2EE frameworks typically don't address the full range of enterprise J2EE developers' needs. Instead, these frameworks only try to solve limited portions of the broad range of enterprise development needs, and thus come up lacking when used to build large, real-world enterprise Web applications.

Perhaps the most common failing we've seen is the predominant focus on JSP rendering and tag libraries. Apache Struts is perhaps the most well-known example of such a framework. Although JSPs are an integral part of any J2EE Web application, and tag libraries are crucial to reduce JSP authoring costs, they cannot be the primary focus of a framework that wishes to minimize developer work while maximizing application maintainability, both of which are critical for real-world enterprise development. For example, any kind of programmatic construct in the JSP is a maintenance problem, because it exposes application functionality to the JSP author, and because parallel content must duplicate this functionality in potentially many places. Additionally, these constructs can seldom be as rich or powerful as Java code, leading to an even worse problem of needing to create scriptlets in the JSP to handle complex, but relatively common, situations. Apache Struts emphasizes this kind of application development, and many of its features are targeted to filling out such capabilities.

Although we feel that these kinds of features may be a productivity benefit for small- to medium-sized applications that don’t require significant maintenance or an extended lifespan, clearly such applications are not typical in the enterprise. By contrast, the Sun ONE Application Framework recognizes and leverages the advantages of JSPs without compromising maintainability or the ability of the application developer to finely control rendering of the JSP. It accomplishes these goals by separating the view tier into a rendering specification (the JSP) and rendering logic components (`View` components). The combination of these entities simultaneously keeps programmatic constructs out of the JSP, where they are mixed with content and are hard to maintain, while providing even greater control over rendering by using fine-grained, view-related events.

Another common failing of other frameworks is the lack of any formal notion of a model or a view tier interface to backend components. In order for developers to quickly build extensible Web applications, there must be a defined contract between the view components that present application data and those that generate it. What we most commonly see in other frameworks is no specification of such a contract or interface, so that developers are either forced to provide data in a view-tier specific format (such as a concrete object instance), or write tedious code to marshall data from the backend to the view. Again, given a small project, or a project that doesn’t require future changes to its backend tier, this might be acceptable.

The Sun ONE Application Framework, however, provides a formal contract between the view and the backend via its `Model` interface, so that view components can be fully independent from backend components. This ability also allows developers to seamlessly change the backend associated with a view without any changes to the view itself. This means that a view could render directly from a SQL query early in an application's life span, but later render data from an EJB as the application's enterprise tier matures, all without the view components knowing the difference. For this reason, the Sun ONE Application Framework provides a full model-view-controller architecture, while most other frameworks, in effect, only provide a view-controller architecture.

Finally, it’s common for other frameworks to simply not consider the submit cycle of a Web application, so that interlinks and relationships between application components are unspecified and hard to maintain. This omission places a burden on the developer that, unfortunately, may not be readily apparent when the project is started. For example, although many frameworks provide structure for outgoing data, they provide little or no structure for incoming data, so that invoked components are forced to work *in the dark*, not able to reliably know what data they’re receiving on any given request invocation.

Worse yet, multiple application paths to the same component force the preparation of the data required by the target component into the callers. This greatly hinders maintainability due to proliferation of inter-object dependencies. Commonly, this proliferation of relationships is reflected in a proliferation of low-level controller logic necessary to do nothing more than manually shuffle input data to the target component or backend. This can lead to an asymmetric notion of a backend component or model being used to render a page, but not used to directly handle a request from a previously rendered page. This asymmetry places yet more burden on the developer to micro-manage backend components and concern himself with the low-level details of running in a Web application container.

Instead, the Sun ONE Application Framework incorporates this class of functionality into its core design pattern and implementation. This frees the developer completely from concerning himself with the population of data to and from the rendered view and the backing models. The result is that from the developer's perspective, models remain stateful between requests without imposing the burden of actual statefulness on the application (which would not scale).

In summary, the Sun ONE Application Framework addresses the full range of enterprise developer needs, and avoids focusing on only one technology or a subset of those needs. Other frameworks tend to take a narrower approach that might address one or two aspects of enterprise development, but seldom all of the key aspects that are necessary to build large-scale, real-world enterprise Web applications.

The Sun ONE Application Framework has the notion of a "display field". This isn’t like the J2EE Blueprints or other J2EE architectures I’ve seen—why not just pull values directly from a helper bean?

The display field paradigm offers unique advantages over more primitive techniques. Before explaining these advantages, first understand that there is no reason to use *display fields* in an application as we’ve envisioned them. The container and child view mechanisms are based entirely on the notion of embeddable arbitrary view objects. A child view object could be as coarse grained as an entire shopping cart display or application menu, or as fine-grained as individual display fields. This flexibility allows application composition from modular pieces, as well as a more traditional display field oriented approach. In short, you *can* just pull values directly from a helper bean in the Sun ONE Application Framework, but we hope to convince you that there is a better way.

Each top-level `ViewBean` instance (or *root view*) is an instance of `ContainerView`, and can contain any arbitrary set of sub-views, some of which can be display field views. `TiledViews` are also sub-views, and can be arbitrarily nested, in addition to containing child views themselves.

This hierarchy of views is somewhat more intricate than what the typical Web-tier developer might expect. For example, many such developers might simply create a helper bean which declares all of the methods necessary to obtain the values needed to render a companion JSP. The source of those values would be encapsulated inside the helper bean's methods. They would use the "*" notation in a *<jsp:setProperty>* tag that would automatically map submitted request parameters to bean fields. This is straightforward, but has some significant disadvantages for development and maintenance.

Although the Sun ONE Application Framework is similar, the use of sub-views becomes very important to maintain a strict model-view separation. For example, all display field views are *bound* to a model. They have no notion of a value contained within them. Furthermore, all display fields are now *bound*, though not all are *data bound*. In some cases, this model is an instance of `DefaultModel`, which is simply in-memory storage. In other cases, the bound model is a SQL query, a stored procedure, an EJB, a business object, an XML DOM, or a SOAP procedure. The display field views are completely separate from the storage and management of data and business behavior.

Display fields are also model-agnostic, in that they can be bound to any model and work without knowledge of the type of that bound model. This means that you need not write application code to move values from some value source to a value consumer—what we call *data shuffling*. Instead, you get this for free in the Sun ONE Application Framework, unlike in the plain helper bean scenario in which you would need to write (and maintain) target-specific code to get values from an EJB, a JDBC ResultSet, and so on, inside each of the bean's accessors. In short, display fields provide the minimal indirection needed to allow seamless pluggability of arbitrary models. Also, because the interaction of display fields with their associated models occurs via the very general `Model` interface, the backend data can be represented in its native format without requiring expensive marshalling to or from a data format required only by the view tier.

In addition, display fields significantly improve the programming model from the typical helper bean / taglib approach. Not only do they allow for intuitive manipulation by the controlling logic, but they provide type-specific operations and an interface to the HTML rendering, none of which is possible with other approaches.

For example, one of the drawbacks to the typical taglib approach is that the helper bean has no real input to the HTML rendering process. If one needs to control this rendering, as is common, this failure frequently leads to a burdensome amount of application-related logic and properties in the JSP as scriptlets. For example, you may want to skip the rendering of a field because the current user isn't authorized to see it. In the typical JSP, the developer would have to provide a scriptlet to circumvent the display, which puts Java code in the JSP. An alternative would be to enhance the tag handler to conditionalize its display based on some standard mechanism, like checking a condition variable. The problem with either approach is that there is either no consistency or no partitioning of application-related data vs. display-related data. In other words, it's messy and less maintainable.

The Sun ONE Application Framework gets around this limitation by providing the best of both worlds. For example, if a developer wants to conditionalize the display of a field, or customize the output of the field's HTML at the last minute, he can implement a display event handler in the parent view and that handler is automatically invoked during HTML rendering. The developer can then skip the display of that field or manipulate the HTML output directly (for example, changing a text box into plain text). Or, he can call methods on the display field view that indicate the necessary action, and this action will automatically be taken into account when the field is rendered by the HTML rendering subsystem (the JSP / taglib combination).

Thus, rather than placing display-related code in the JSP along with page content, or worse, in controlling business logic or in the business-oriented model, developers can either augment the display field rendering process or easily direct it. This not only keeps Java code out of the JSP, but it is far more powerful than scriptlets or other approaches to controlling HTML rendering. Another benefit is that it is far more consistent.

Display fields also allow the developer to work with an HTML page as if it were a stateful server-side object. When the user clicks a button or href on the HTML page, the request is ultimately routed back to the view that rendered that button or href, and an event handler corresponding to that object is invoked. But, before invoking the event handler, the Sun ONE Application Framework repopulates all the display fields and views with the submitted request parameters. The effect is that from the developer's point of view, the page remains stateful and simply responds to commands from the user. The developer handles the event for the most part like he or she would in a *fat client* application, by implementing an event handler and taking action based on the request and its updated field values. Realize that because display fields are always bound to a model, any changes in the value of the field are automatically propagated to the model. This allows developers to choose the most productive compromise between traditional *fat client*-like and formal MVC-like programming styles.

It is important to understand that the alternative approaches of using helper beans and using display fields both implement the facade design pattern, in that the interface to and from the HTML page is handled by these objects. The data values provided via this interface can come from a number of sources, including multiple backing EJBs, business objects, result sets, and so on, all of which can be referred to in abstract terms as *models*. However, whereas the traditional helper bean would manage these models via custom code, with little or no consistency or reusability, the Sun ONE Application Framework abstracts them to a formal definition of a `Model`, and specifies a clear contract between a model and a view bound to it.

This formal contract has several advantages. First, it allows display fields to provide an easily mutable mapping between the facade presented by the view and the set of models backing that facade. As we include XML-based declarative features into the Sun ONE Application Framework (work already in progress), display fields will allow declarative changes to this facade, greatly simplifying application development and drastically reducing development time.

Second, because this facade is so easily changeable, changes to backing models can be propagated to the view portion of the application with extreme ease, and with declarative support, without recompilation. Compare this to the traditional helper bean approach in which developers would need to add accessors and mutators to their helper bean and/or modify the code needed to obtain values from or send values to the backing model. Because of these fine-grained changes and the recompilation they require, development time, productivity, and object reuse suffer significantly.

Do the Sun ONE Application Framework applications require the use of EJBs?

No. Of course, like in any other J2EE application, you can simply obtain a reference to an EJB and use it directly from within a Sun ONE Application Framework application. But, while EJBs are a component of J2EE, they are not a mandatory component. Furthermore, they are a relatively complex addition to an application's architecture, and currently have some significant drawbacks. There are a large number of Web developers who are not prepared to move to an EJB-based architecture, as it presents many unique challenges and requires a significant additional investment in many areas of application design and implementation. We felt it would be a disservice to customers for us to require their use.

However, this isn’t to say that we don’t support and facilitate the use of EJBs. The Sun ONE Application Framework provides valuable features based on a flexible and pluggable model-view architecture, which compliments both Web-oriented and EJB-oriented applications. As an alternative to using an EJB directly, for example, one could use a model that is either backed or directly implemented by an EJB. Integration of EJBs is as seamless as it is for any other kind of model in the Sun ONE Application Framework, and provides automatic data binding and other high-level features.

As an initial way to get started with EJBs, we include in the Sun ONE Application Framework a class called `BeanAdapterModel` that maps the standard `Model` interface onto arbitrary JavaBean properties. If you have a Customer EJB, you can wrap it in this adapter model so that display fields can access it's properties / values directly without any additional code. Using this adapter, values are rendered from the Customer bean on display and are pushed back into the bean on submit automatically. You could also use this class to wrap access to local business objects as well—the adapter does not care whether the objects it encapsulates are remote or local.

How are the Sun ONE Application Framework applications structured?

The Sun ONE Application Framework applications are fully independent entities comprised of one or more *modules*. Each module is a functional slice or logical grouping of the overall application. At least one module is required in an application, but other modules are optional and may be added at any time.

Each Sun ONE Application Framework application defines what we call the *application servlet*, which is the base class from which all module-specific servlets are expected to derive. Only the module servlets are accessed by clients of the application; the application servlet serves only as a common base class for the module servlets, providing the opportunity to consolidate common application-level event handlers in a single class. Module servlets allow for module-only specialization of these events. Together, these servlets form the request handling infrastructure of each Sun ONE Application Framework application.

Each module corresponds to a sub-package of the application, and in addition to containing a minimal servlet infrastructure specific to that module, contains one or more logical pages. Each module may also contain supporting models and other non-Sun ONE Application Framework classes (of course, classes in a module can use classes outside the module as usual).

Each logical page in the module consists minimally of one JSP and one ViewBean. The ViewBean is the helper bean for the corresponding JSP, and in conjunction with the Sun ONE Application Framework tag library, provides the display/application event infrastructure. Each ViewBean contains an arbitrary hierarchy of reusable child view objects which can be assembled to create full-fledged, data-bound pages within minutes.

How are the request flow and URL format implemented?

All requests are initially handled by a controller servlet (one per module, several per application), and the URL of this servlet is chosen by the developer. This servlet dispatches a request to a request-specific controller object (the ViewBean), which then ultimately forwards the request to another resource (a JSP, ViewBean, or other Web resource). The source code for the servlet and its dispatching mechanism is fully under developer control, and we encourage developers to learn the details of this mechanism by reading the well-commented source.

How does a view bean relate to a session or entity bean?

There is no direct relation. A view bean is a JSP worker or helper bean (a *usebean*). Each view bean acts as the central support for its peer JSP. Session and entity beans can be accessed from within the view bean or any of its associated models or views if desired.

With the JSP scope set to "request" to simplify threadsafe coding and force beans to be constructed and destroyed with each request, will there be negative performance impact?

In general, we believe that the overhead of supporting persistent application objects is typically greater than that present in the current approach. In our testing, we have found that the overhead from the current approach is insignificant when compared to what proprietary, non-J2EE containers previously did, as well as the typical behavior associated with a Web application. In essence, allocation of objects in modern JVMs is a sufficiently cheap operation that we feel justifies the benefits provided by this approach. However, this doesn’t mean that we’ve ignored the implications of such an approach.

Specifically, although some non-J2EE containers provided persistent application objects for efficiency reasons, these objects were not stateful to any client. Thus, each client's stateful information had to be regenerated on each request. We have been careful to design our equivalent application objects with as little overhead as possible, which is, in most cases, equivalent to the useful, stateful information that must be recreated on each request anyway. In addition, the Sun ONE Application

Framework supports optimized access to these objects so that they are only created when needed. Thus, the Sun ONE Application Framework-based applications are generally more efficient than non-J2EE applications in both processing and memory consumption—in most cases, significantly so.

Furthermore, it is not necessarily possible to design an infrastructure that could make use of persistent application objects in a useful way. Both the Sun ONE Application Server and other highly scalable servlet containers make use of multiple JVMs, and a user's requests are not necessarily routed back to the same address space. This means that persistent application objects would have to be reinitialized for each request anyway, which is at least as much overhead, if not more, than simply creating the objects anew. Additionally, placing application objects in the session is not a viable option, as it is an extremely expensive operation and minimally requires deserialization, which is more expensive than simple object allocation.

Finally, we feel that adding the layer of functionality to the Sun ONE Application Framework to support persistent application objects would significantly separate developers from the underlying container. This generally violates our design principles of keeping things as close to standard J2EE as possible.

Having made these objections, however, we may in the future still provide the ability to use persistent application objects if we find that the possible benefits outweigh the drawbacks. However, we expect that such functionality, if present, would be applicable to only a narrow range of applications and not helpful to the typical Sun ONE Application Framework application.

Index