Getting Started

Sun ONE Application Framework

Version 2.0

817-0446-10
October 2002

Contents

About This Document

This Sun™ ONE Application Framework *Getting Started Guide*, with its associated tutorial, introduces developers to the mechanics and techniques used to build Web applications with the Sun ONE Application Framework tools.

This document and the tutorial assume Java expertise and familiarity with the development and deployment procedures for the specific servlet container and development tools being used.

This preface contains 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:

- About This Document

- Getting Started
- Before You Begin
- Tutorial Sections (Links to)
- Tutorial—Section 1.1 Application Infrastructure
- Tutorial—Section 1.2 Create Login Page
- Tutorial—Section 1.3 Test Run the Login Page
- Tutorial—Section 2.1 Prepare Application to Access SQL Database
- Tutorial—Section 2.2 Create the CustomerModel
- Tutorial—Section 2.3 Create Customer Page
- Tutorial—Section 2.4 Test Run the Customer Page
- Tutorial—Section 2.5 Link Login Page to Customer Page
- Tutorial—Section 2.6 Run Application
- Index

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." on page 11.

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" on page 11 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/supportraining/`
- 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

Before You Begin

Welcome to the Sun™ ONE Application Framework, the J2EE Web application framework and toolset (IDE) for enterprise Web application development.

This section contains the following topics:

- Primary Features of the Sun ONE Application Framework
- QA Certification

Primary Features of the Sun ONE Application Framework

The primary features of the Sun ONE Application Framework are:

- Turnkey J2EE application development.
- High performance, proven J2EE framework runtime.
- Full component-based development.
- Graphical application builder toolset:
    - Logical application tree explorer view.
    - Automated synchronization of changes between application components and JSPs.
    - High-level wizards.
- Support for Web Services Model (Enterprise Edition only).

The Application Framework is used by:

- Large enterprises doing medium-, large-, or massive-scale enterprise Web applications.
- Financial, Manufacturing, Government, Education, Health Care, and Telecommunications sectors.

The Application Framework is a valuable tool to:

- Guide naïve and junior Java/J2EE Developers:
  - Provides exceptional ease of use and an easy learning curve with the graphical development tools.
  - Leverages complex J2EE APIs for those without detailed knowledge.
  - Inexperienced developers learn J2EE as they build high-performance enterprise applications.
- Complement advanced Java/J2EE developers and architects:
  - Advanced developers gain higher productivity by avoiding tedious low-level J2EE development.
  - Architects have well-defined points from which to extend the application architecture.
- Accelerate Web Application development and skill/component reuse by providing easy entree into the J2EE API world.

This document shows you how to use the Application Framework features to:

- Create a Sun ONE Web Application.
- Create a page (ViewBean) and an associated JSP.
- Create and use a Model.
- Link pages together.

QA Certification

- Solaris 8
- Solaris 9
- Windows 2000
- Javasoft RI and Tomcat

- Sun ONE Application Server 6.5 and 7.0, WebLogic, WebSphere (J2EE container testing done via WAR import export)
- Sun ONE Studio 4.1, Enterprise Edition
- Sun ONE Studio 4.1, Community Edition

Chapter 2

Getting Started

This chapter outlines the mechanics of using the Sun™ ONE Application Framework tools to build a J2EE Web application.

This section contains the following topics:

- Introduction
- Writing Sun ONE Application Framework Applications
- About the Sun ONE Application Tutorial

Introduction

This document, with its associated tutorial, introduces developers to the mechanics and techniques used to build Web applications with the Sun™ ONE Application Framework tools.

It is intended for developers who are 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 document and the tutorial assume Java expertise and familiarity with the development and deployment procedures for the specific servlet container and development tools being used.

Because the Application Framework is foremost a design pattern and a set of interfaces, the examples here show only the most basic way of creating an Application Framework application, by extending existing Application Framework implementation base classes and manually constructing certain application objects. Realize, though, that this is only one possible way to create an Application Framework application.

There are two reasons for not showing more advanced techniques in this document. First, starting at a fundamental level is the most direct way to impart how the Application Framework works to someone new to the framework. Being able to see exactly how the framework interacts with the application is critical to getting the most out of the Application Framework.

Second, building an application using these fundamental techniques is a prerequisite to fully understanding the many possible ways to build Application Framework applications. Features that extend the Application Framework to add additional capabilities are built on the techniques demonstrated in this document. After understanding these basic examples, you have a greater understanding of how these features extend and complement the Application Framework core, and you are able to optionally decide not to use them and instead construct your own Application Framework extensions (or simply fall back to a more basic approach where necessary).

The ultimate goal of this document, then, is to introduce developers to the most fundamental way to build Application Framework applications, so they become familiar with Application Framework's interactions with applications built on top of it, and more fluent in the Application Framework itself.

Writing Sun ONE Application Framework Applications

Writing an Application Framework application consists of first laying out an application structure, and then incrementally adding Application Framework objects to that structure. Although this can be done entirely by hand, from scratch, the task has been simplified by creating an Application Framework tools module for the Sun™ ONE Developer Studio that assists developers in writing their Application Framework applications. With the assistance of these tools, creating an Application Framework application becomes a simple process of generating Application Framework components using wizards and customizing them to an application.

Before demonstrating the creation of a simple Application Framework application, you will cover the basics of how an Application Framework application is structured.

J2EE/Sun ONE Application Framework Terminology

In this document, you will come across terms such as *application*, *module*, and *components*. These terms can be confusing, because they are also used in more general Web architecture and development discussions.

The following table contains a list of the most important terms that you will find in this document.

Term	**Description**
J2EE component*	Sometimes referred to as J2EE *application* components; concrete software components which are deployed, managed, and executed on a J2EE server including EJBs, Servlets, and Java Server Pages (JSPs); there are components including HTML and Applets which are also J2EE components but these are not relevant to the Sun ONE Application Framework Web application discussion.
J2EE module*	Represents the basic unit of composition of a J2EE application. A J2EE module consists of one or more J2EE components and one component-level deployment descriptor. J2EE modules can be deployed as stand-alone units or can be assembled with a J2EE application deployment descriptor and deployed as a J2EE application. Servlet and / or JSP components are packaged as a J2EE module and deployed as a WAR file. EJB components are packaged as a J2EE module and deployed as a JAR file. An arbitrary number or WAR files and JAR files may be combined to form a J2EE application and deployed as an EAR file. WAR files (J2EE modules which are also known as J2EE Web applications) may be deployed stand-alone on a J2EE server.
J2EE Web application*	Stand-alone J2EE modules containing J2EE components deployable in a J2EE servlet container (Web application container). Depending on the context of the term *application* or *J2EE application*, the intent may be to refer to a J2EE Web application; there are products such as Sun ONE Application Server 6.x and Apache Tomcat which support J2EE *Web* applications in that they can manage J2EE modules consisting of Servlets and JSPs, but they cannot manage a complete J2EE application which may have EJB J2EE modules.

Term	Description
J2EE application*	Consists of one or more J2EE modules and one J2EE application deployment descriptor, packaged using the Java archive (JAR) file format into a file with a `.ear` (enterprise archive) filename extension.
Sun ONE Application Framework module	Refers to both a logical and physical partition of content and components within a Sun ONE Application Framework application (not to be confused with a J2EE module).
Sun ONE Application Framework application	In informal terms, a Sun ONE Application Framework application is a J2EE Web application that has been written using the Sun ONE Application Framework. It consists of at least one J2EE module (the Web application), but may also include other standard J2EE components or modules. A minimal Sun ONE Application Framework application is a J2EE Web application consisting of one WAR file. In formal terms, a Sun ONE Application Framework application is a collection of related Sun ONE Application Framework modules, all running in the same servlet context. In this sense, *Sun ONE Application Framework application* refers only to this logical Sun ONE Application Framework abstraction.

* Refer to the Java 2 Platform Enterprise Edition Specification v1.2 (J2EE) section J2EE8.1 for a detailed explanation of this term.

How Sun ONE Application Framework Applications Are Organized

The Application Framework provides formal *application* and *module* entities. An Application Framework application is a base Java package that contains one or more sub-packages (Application Framework *modules*). It is perfectly acceptable for an application to consist of only one module, and it is likely be the common situation for smaller applications. Each module inherits behavior from its parent application-level components, and may also customize this behavior separately from other modules.

In J2EE Web application container terms, an Application Framework application corresponds one-to-one with a servlet context, and thus is subject to the constraints enforced by the container for servlet contexts.

Before starting to develop your application, you should first decide how it should be organized:

- Determine which modules will be grouped together into your Application Framework application.

  Avoid over-categorizing your application into several modules simply because the Application Framework provides this capability. In many cases, one module is sufficient.

- Decide on an application package name.

  The application package name can be arbitrarily complex and will likely reflect your organization's packaging strategy. Each of your modules becomes a package beneath this application package.

- Assign a deployment-time or published Web application name.

  In Apache Tomcat, the directory immediately beneath the `/webapps` directory would bear this name. In the Sun™ ONE Application Server, the directory immediately beneath the `/ias/APPS/modules` directory would bear this name. The deployed application name is the same as the name WAR file name.

For example, suppose you have two Application Framework modules (named *module1* and *module2*) that comprise an Application Framework application. You would call this application *myapp*. The full application package name is `com.mycompany.myapp`.

- The application package would be `com.mycompany.myapp`
- The module1 package would be `com.mycompany.myapp.module1`
- The module2 package would be `com.mycompany.myapp.module2`

One final note about application and module naming conventions: In general, the application package name should be different from that of any of its modules.

For example, your first instinct might be to name both your application and its primary module *foo*. This can easily lead to confusion for someone trying to understand your application, as well as your application development tools. Instead, consider naming the application package something like *fooapp*, or calling the primary module something like *main* or *module1*. This makes your application structure much easier to understand, especially when you add to it in the future.

About the Sun ONE Application Tutorial

You will now develop a simple application to get a taste of using the Sun ONE Application Framework and the tools. This application consists of two pages: a login page, and a customer account page, and demonstrates the following:

- Retrieving field values submitted by the user.
- Returning a status message to the user.
- Using a QueryModel to retrieve customer information.
- Using a QueryModel to update customer information.
- Coordinating user input with QueryModel SQL WHERE criteria.
- Moving from one page to another.

This tutorial breaks the steps required to develop the application into *chapters* and *tasks*. Each chapter addresses a broad topic, at the end of which you have an application that you can run. Each task within a chapter is a relatively self-contained topic and contains several more detailed steps.

Chapter 3

Tutorial Sections (Links to)

This section outlines the sections contained in this *Sun™ ONE Application Framework Getting Started* document.

This section lists the links to the various tasks as follows:

- Sections 1.1—1.3
- Sections 2.1—2.6

Sections 1.1—1.3

In Sections 1.1 through 1.3, you create the application infrastructure needed for all subsequent chapters, and add your first Sun™ ONE Application Framework page.

- Section 1.1

  Task 1: New Sun ONE Web Application
- Section 1.2

  Task 2: Create the Login Page
- Section 1.3

  Task 3: Test Run the Login Page

Sections 2.1—2.6

In Sections 2.1 through 2.6, you expand the existing application by adding a SQL-based model, and a page to display that model's data. You then link the two application pages together so they show coordinated data.

- Section 2.1

  Task 1: Accessing a SQL Database
- Section 2.2

  Task 2: Create the CustomerModel
- Section 2.3

  Task 3: Create the Customer Page
- Section 2.4

  Task 4: Test Run the Customer Page
- Section 2.5

  Task 5: Link the Login Page to the Customer Page
- Section 2.6

  Task 6: Run the Application

Chapter 4

Tutorial—Section 1.1 Application Infrastructure

This section describes how to create the Sun™ ONE Application Framework application infrastructure needed for all subsequent tasks.

Task 1: New Sun ONE Web Application

Before developing any pages, you need to create the Application Framework application infrastructure (the WAR directory structure and supporting files). This is a onetime requirement for each Application Framework application.

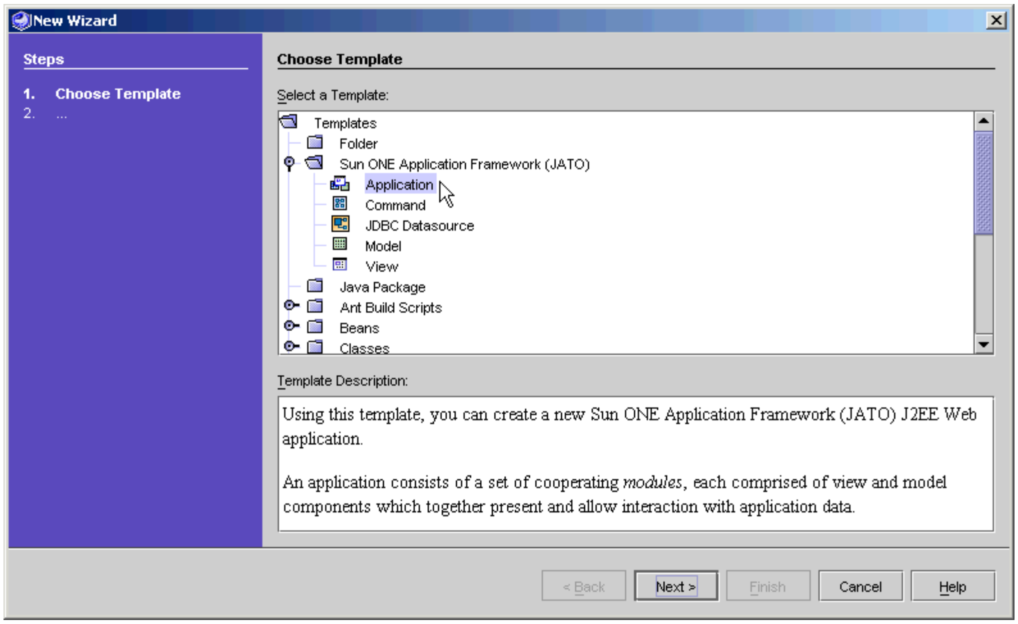
Create an Application Wizard

Before you create the application, you need to decide where the application should be located. Typically, developers develop the application directly in the `webapps` directory of a servlet container so the application can be tested without deploying it to the target runtime environment. Since you are already using the Sun ONE Studio, you can locate the application anywhere and use the built-in Tomcat module to test it in place without the deployment step.

1. Select the menu option File-->New from the Sun ONE Studio.

   The Choose Template page appears.

2. Expand the Sun ONE Application Framework folder.
3. Select Application.



Select a Template:

- Templates
  - Folder
  - Sun ONE Application Framework (JATO)
    - Application
    - Command
    - JDBC Datasource
    - Model
    - View
  - Java Package
  - Ant Build Scripts
  - Beans
  - Classes

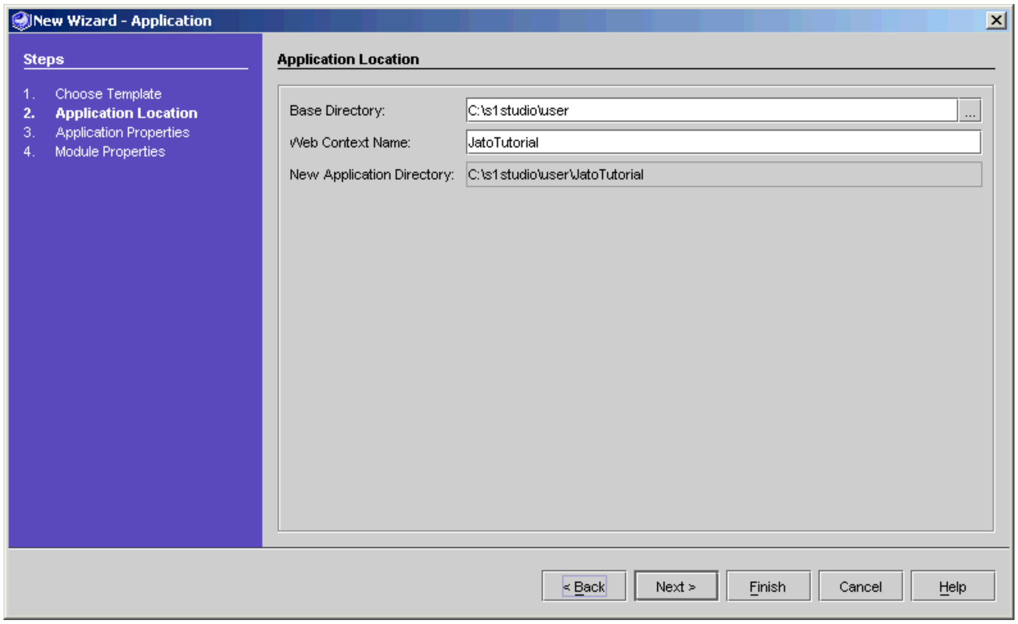
Template Description:

Using this template, you can create a new Sun ONE Application Framework (JATO) J2EE Web application.

An application consists of a set of cooperating *modules*, each comprised of view and model components which together present and allow interaction with application data.

4. Click Next.

The Application Location page appears.



Base Directory: C:\s1studio\user

Web Context Name: JatoTutorial

New Application Directory: C:\s1studio\user\JatoTutorial

The default base directory is your Sun ONE Studio `user-dir`, which may be different than the one shown in this example. You can choose any existing directory to be your base directory for your Sun ONE Application Framework applications.

NOTE Many developers use the `webapps` directory of the servlet container in which the application is deployed.

Later in this tutorial, you will see how to run your Sun ONE Application Framework Web application using the Sun ONE Studio, so you can put your Web application anywhere you want it to be.

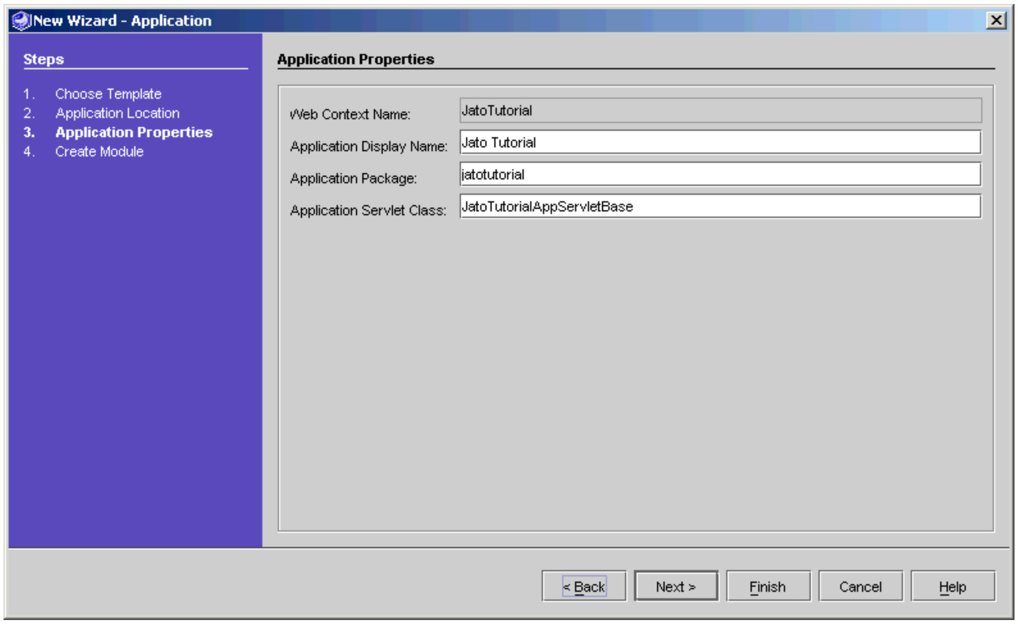
---

5. Enter `JatoTutorial` in the Web Context Name field.

   The New App Directory field is populated after you make entries in the Base Directory and Context Name fields.

6. Click Next.

   The Application Properties page appears.



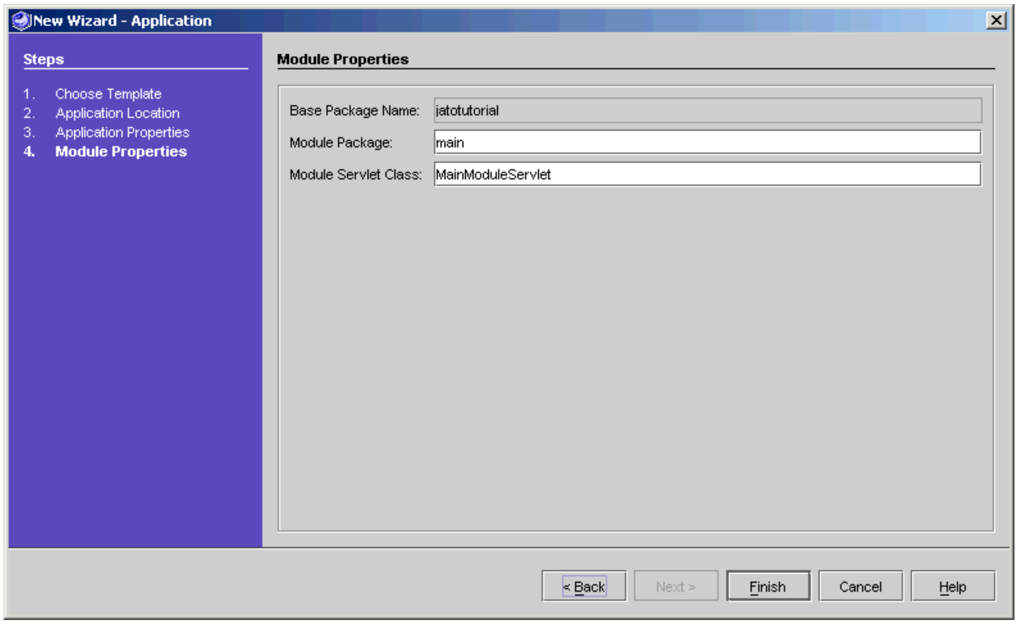
Web Context Name:	JatoTutorial
Application Display Name:	Jato Tutorial
Application Package:	jatotutorial
Application Servlet Class:	JatoTutorialAppServletBase

The fields on this page are populated using the value of the *Web Context Name* field from the previous page.

7. For this tutorial, accept the default values.

8. Click Next.

   The Module Properties page appears.



Module Properties

Base Package Name: jatotutorial
Module Package: main
Module Servlet Class: MainModuleServlet

9. For this tutorial, accept the default values.

10. Click Finish.

11. Click OK.

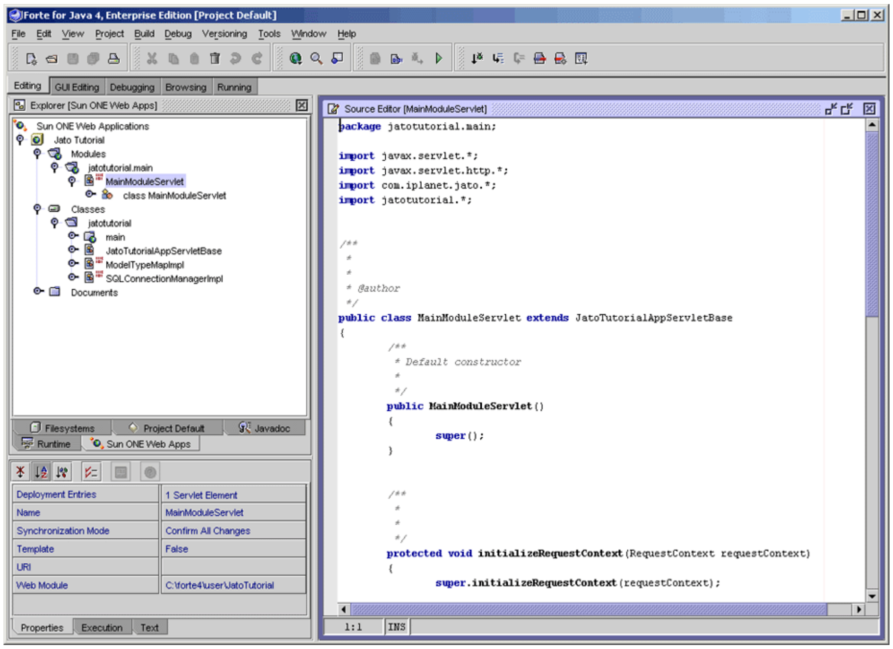
    The application is created.

NOTE The processing time depends upon your machine.

The new application appears in the Sun ONE Application tree in the Sun ONE Studio Explorer (Sun ONE Web Apps).

12. Expand the modules node in the Sun ONE Studio Explorer on the Sun ONE Web Apps tab to see the application layout and observe the code in the two servlet classes that were created:

    - `JatoTutorialAppServletBase`
    - `MainModuleServlet`



```
Sun ONE Web Applications
  Jato Tutorial
    Modules
      jatotutorial.main
        MainModuleServlet
          class MainModuleServlet
    Classes
      jatotutorial
        main
        JatoTutorialAppServletBase
        ModelTypeMapImpl
        SQLConnectionManagerImpl
    Documents
```

Deployment Entries	1 Servlet Element
Name	MainModuleServlet
Synchronization Mode	Confirm All Changes
Template	False
URI	
Web Module	C:\forte4\user\JatoTutorial

```
package jatotutorial.main;

import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.*;
import jatotutorial.*;


/**
 *
 *
 * @author
 */
public class MainModuleServlet extends JatoTutorialAppServletBase
{
        /**
         * Default constructor
         *
         */
        public MainModuleServlet()
        {
                super();
        }


        /**
         *
         *
         */
        protected void initializeRequestContext(RequestContext requestContext)
        {
                super.initializeRequestContext(requestContext);
```

Module Node

The file structure under the `Classes` node appears to be duplicated under the `Modules` node.

In reality, the `Modules` node is a *flattened* version of the `Classes` node, but only folders that are marked as Application Framework modules appear in the `Modules` node.

The `Classes` node reflects the complete and true layout of the package structure of your application. The `Modules` node makes it easier to navigate your application, since the most commonly visited classes are located in a module folder.

Application Servlet

The application servlet, `JatoTutorialAppServletBase`, has no special meaning to the application, except that it is meant to be a super class for all module servlets in the application.

The Application Framework module servlets have events that can be implemented to customize and control the session and request life cycle.

For example:

- `onNewSession`
- `onSessionTimeout`
- `onBeforeRequest`
- `onAfterRequest`

It is common that all module servlets within the same application require the same behavior for all of these events. Therefore, it is a good idea to implement such behavior for these events in a class that all module servlets can extend.

However, technically speaking, the application servlet is not required. You can customize the hierarchy of the module servlet as long as that hierarchy derives from the Application Framework `com.iplanet.jato.ApplicationServletBase` file.

This application has only one module (and by definition, one module servlet), so the role of the application servlet is not as beneficial as it would be in multi-module applications.

Module Servlet

The module servlet, `MainModuleServlet`, is the actual servlet that is invoked for every request. All access to the application goes through this front controller servlet before control is handed to the appropriate *request handler* class (implemented later in this tutorial).

Not much code is required in this class. All of the necessary request handling code is located in the Application Framework's `com.iplanet.jato.ApplicationServletBase` file.

To gain some insight on how requests are handled, see the source code for this servlet class.

Advanced Tip - Modules

Notice above that the `main` module is selected, and its properties are reflected in the property sheet at the bottom of the Sun ONE Studio Explorer window.

Notice also that its Module property is True. By changing it to False, this module becomes an ordinary folder, and the entries in the `web.xml` file for the `MainModuleServlet` are removed.

You can make any ordinary folder an Application Framework module by right-clicking the folder and selecting the *Convert to Module* action. You are then prompted to select a Java class from that folder to be the module servlet, or you can provide a name to create a new one.

Chapter 5

Tutorial—Section 1.2 Create Login Page

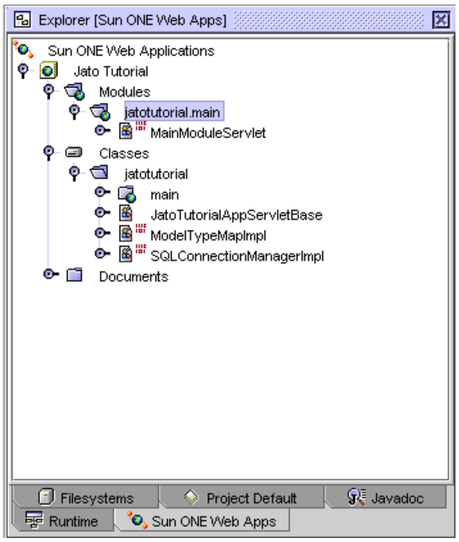
This section describes how to add your first Sun™ ONE Application Framework page to the application infrastructure you created.

Task 2: Create the Login Page

You will now create the first page of the application.

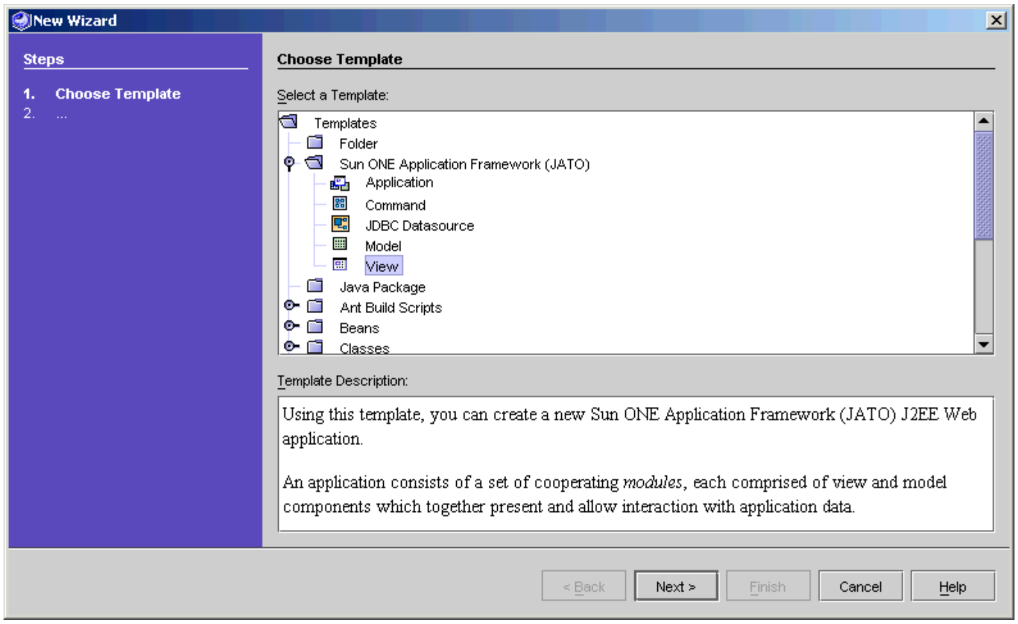
1.2.1 Add a ViewBean

1. Select the *main* module from the Sun ONE Studio Explorer window.



2. Select the menu option File-->New from the Sun ONE Studio.

   The Choose Template page appears.

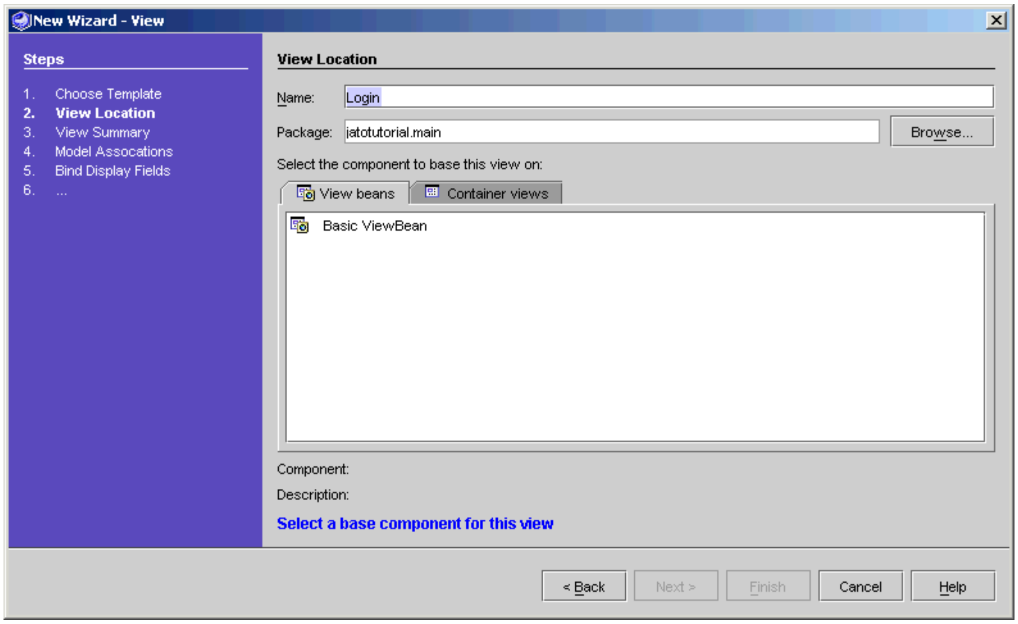


Using this template, you can create a new Sun ONE Application Framework (JATO) J2EE Web application.

An application consists of a set of cooperating *modules*, each comprised of view and model components which together present and allow interaction with application data.

3. Expand the Sun ONE Application Framework folder.
4. Select View.
5. Click Next.

   The View Location page appears.


New Wizard - View

Steps

1. Choose Template
2. **View Location**
3. View Summary
4. Model Assocations
5. Bind Display Fields
6. ...

View Location

Name: Login

Package: jatotutorial.main

Select the component to base this view on:

Basic ViewBean

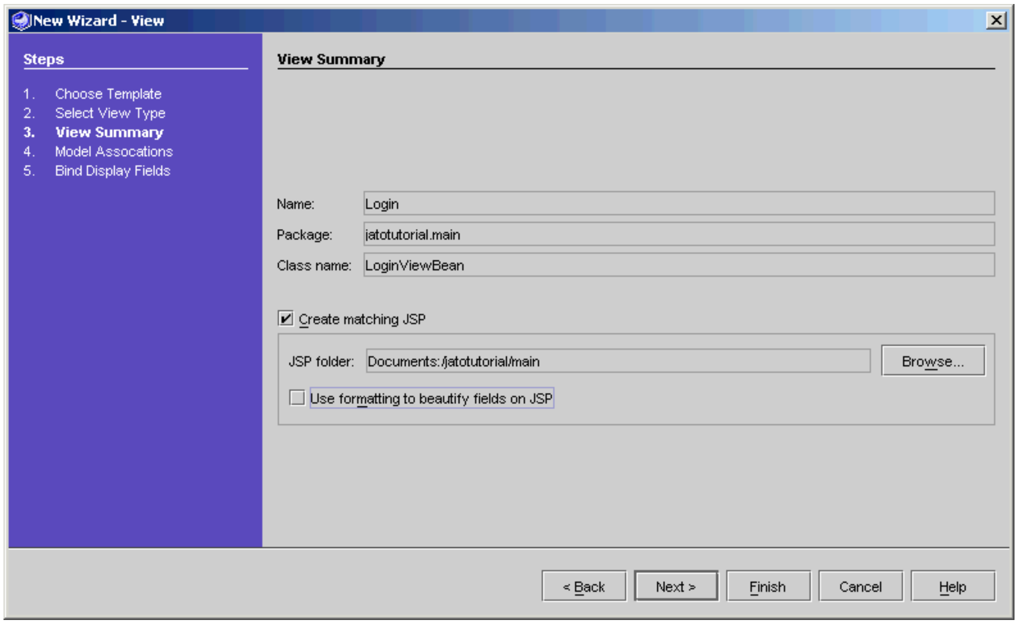
Component:

Description:

Select a base component for this view

6. Enter Login in the Name field (to replace <*default*>).
7. In the View beans tab, select *Basic ViewBean*.
8. Click Next.

   The View Summary page appears.



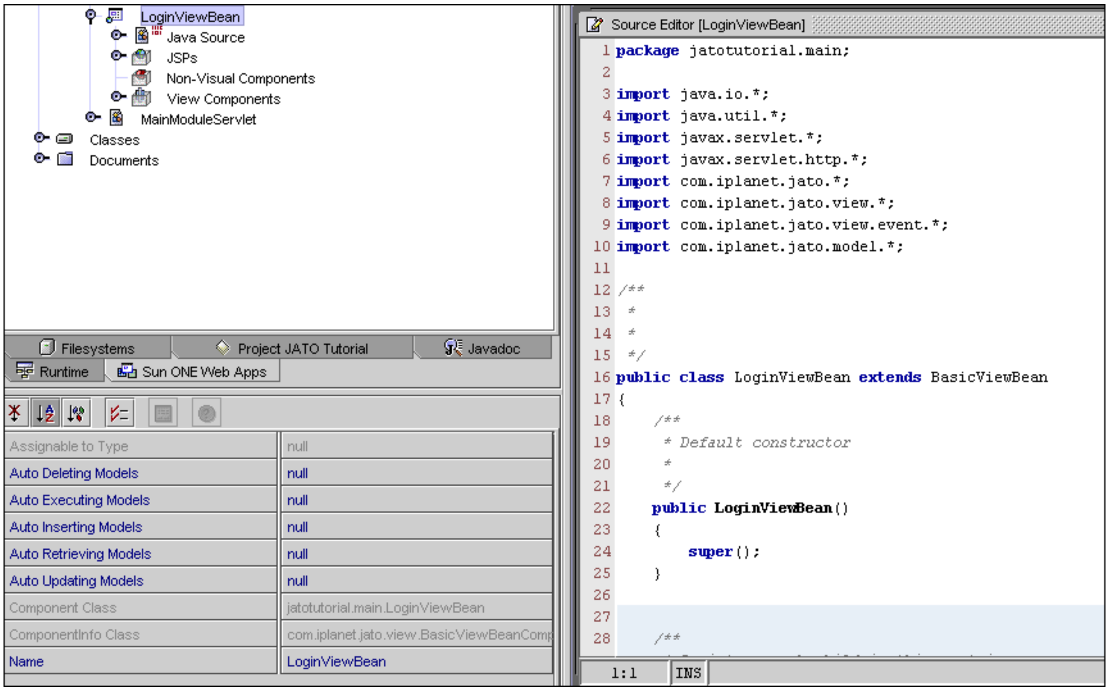
Name: Login
Package: jatotutorial.main
Class name: LoginViewBean
[✓] Create matching JSP
JSP folder: Documents:/jatotutorial/main
[ ] Use formatting to beautify fields on JSP

The Name, Package, and Class name fields are populated as a result of previous entries.

9. Click Finish to create the ViewBean.

The Sun ONE Studio Explorer page appears.

NOTE There are additional steps in the View Wizard. However, those steps involve model field binding which is not required for this Login ViewBean. In a later task, you will use these additional steps.



Assignable to Type	null
Auto Deleting Models	null
Auto Executing Models	null
Auto Inserting Models	null
Auto Retrieving Models	null
Auto Updating Models	null
Component Class	jatotutorial.main.LoginViewBean
ComponentInfo Class	com.iplanet.jato.view.BasicViewBeanComp
Name	LoginViewBean

```
package jatotutorial.main;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;
import com.iplanet.jato.model.*;

/**
 *
 *
 */
public class LoginViewBean extends BasicViewBean
{
    /**
     * Default constructor
     *
     */
    public LoginViewBean()
    {
        super();
    }


    /**
```

10. Double-click `LoginViewBean`.

The added code appears in the right-hand panel of the Sun ONE Studio Explorer page.

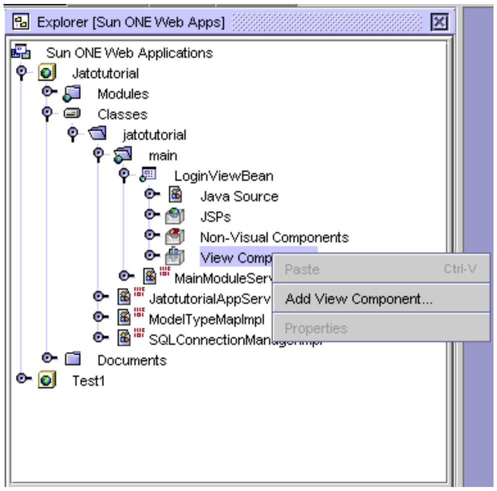
NOTE

- Because you created a JSP when you created the ViewBean, a JSP was added to the Documents folder in a directory structure that mirrors the ViewBean's package structure (`/jatotutorial/main`).
- For convenience, a link to the JSPs that use this ViewBean (just one for `LoginViewBean`) are placed in the JSPs' node, which is under the `LoginViewBean` node.

1.2.2 Add Display Fields to the ViewBean

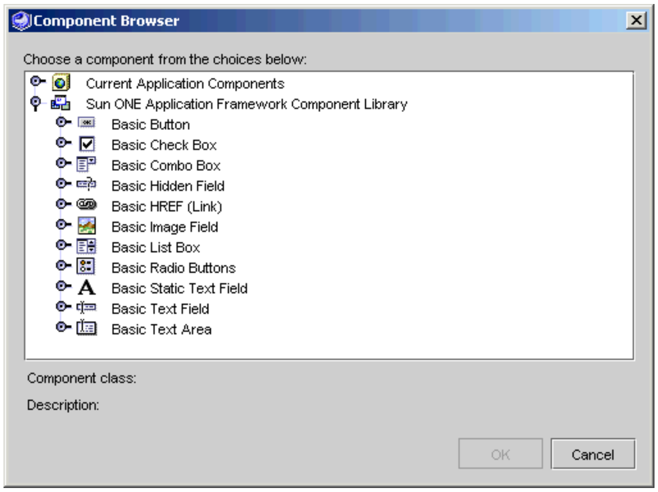
11. Expand the `LoginViewBean` from the Sun ONE Studio Explorer page.

12. Right-click View Components.



13. Select Add View Component... from the pop-up menu.

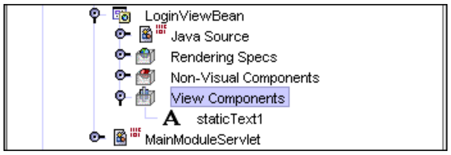
The Component Browser dialog appears.



14. Select Static Text Field.

15. Click OK.

This adds a text field to the View Bean.



The default name is *staticText1*.

16. Right-click the *staticText1* field name.

17. Select Rename.

18. Rename the field *message.*

19. Add two more display fields.

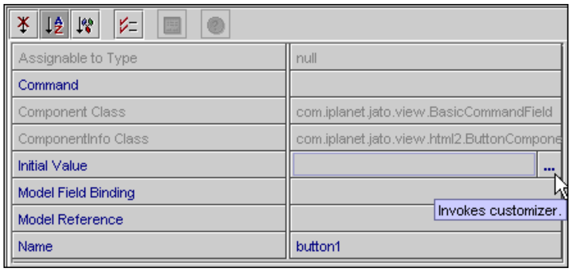
The following table contains a list of the two display field types with each of their names and the initial value for the *Button* type.

Type	Name	Initial Value
Text Field	customerNum	
Button	login	Object Type: String Object Value: Login

Adding display fields to the ViewBean also adds the fields to the JSPs using this ViewBean.

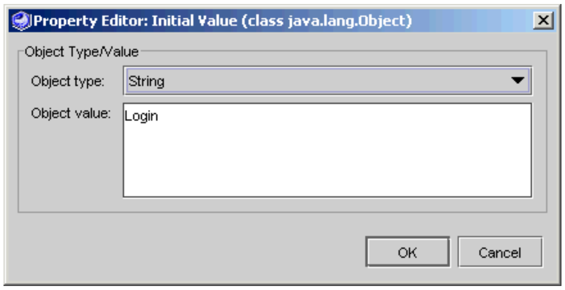
20. Set the button’s Initial Value property:

Click the “...” button in the property’s value cell.



Assignable to Type	null
Command	
Component Class	com.iplanet.jato.view.BasicCommandField
ComponentInfo Class	com.iplanet.jato.view.html2.ButtonCompone
Initial Value	...
Model Field Binding	
Model Reference	
Name	button1

The Initial Value property editor appears.



Property Editor: Initial Value (class java.lang.Object)

Object Type/Value

Object type: String

Object value: Login

21. Set Object type to String.

22. Set Object value to Login.

The button’s value appears on the button in the browser.

23. Open the Login JSP to see the tags for the three display fields.

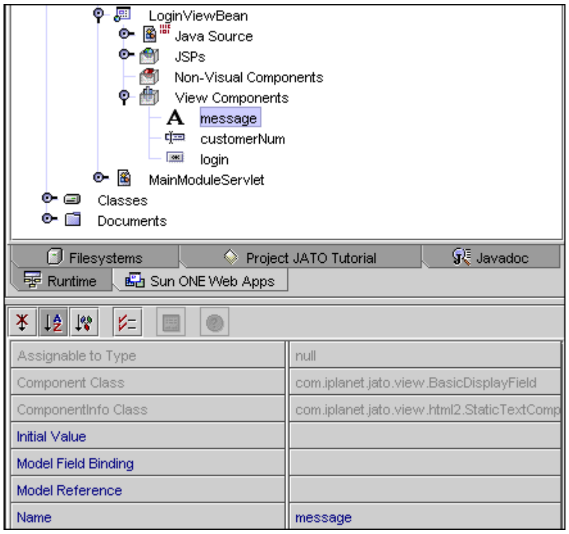
24. Format your JSP layout any way you want to, however:

a. Give customerNum a label.

b. Make certain the message is on its own line (at the minimum, place a *<br>* after the tag).

NOTE Currently, the JSPs that are generated are left in a *vanilla* state. This allows you to format the JSP the way you want to without having to undo a whole lot of unwanted prescribed layout. (Customizable JSP templates are planned for an upcoming release of the tools.)

You can edit it directly in the Sun ONE Studio editor, or you can use your favorite WYSWIG HTML editor.

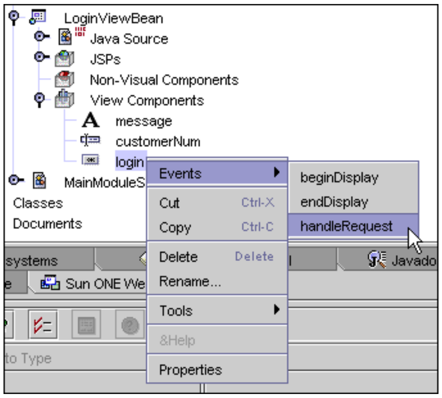


Assignable to Type	null
Component Class	com.iplanet.jato.view.BasicDisplayField
ComponentInfo Class	com.iplanet.jato.view.html2.StaticTextComp
Initial Value	
Model Field Binding	
Model Reference	
Name	message

1.2.3 Add Code to the Login Button

25. Right-click login.

26. Select Events-->handleRequest.



The `LoginViewBean.java` file opens and the handleLoginRequest event stub is inserted.

27. Add the code that will be run when a user clicks *login* in the browser.

The following table contains the code that you need to add.

```
public void handleLoginRequest(RequestInvocationEvent event)
{

     // Retrieve the customer number
     String custNum = getDisplayFieldStringValue(CHILD_CUSTOMER_NUM);

     String theMessage = "";

     // Check the customer number
     if (custNum.equals("1") ||
         custNum.equals("777") ||
         custNum.equals("410"))
     {
         theMessage = "Congratulations, " + custNum +
              ", you are now logged in!";
     }

     else
     {
          theMessage = "Sorry, " + custNum +
               ", your customer number was incorrect!";
     }

     // Set the ouput status message
     getDisplayField(CHILD_MESSAGE).setValue(theMessage);

     // Redisplay the current page
     forwardTo();
}
```

Chapter 6

Tutorial—Section 1.3 Test Run the Login Page

This section describes how to run your Sun™ ONE Application Framework application.

Task 3: Test Run the Login Page

1.3.1 Set the Servlet Properties for Execution

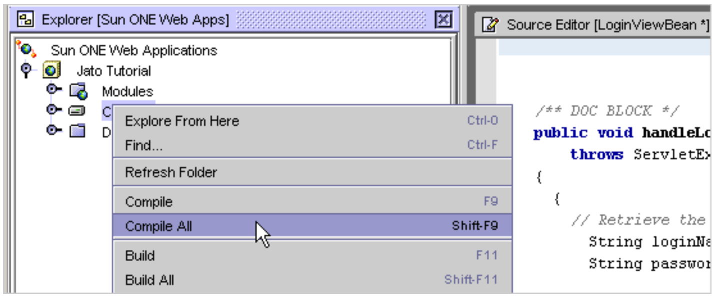
As mentioned previously, the Sun ONE Studio has a method for executing Web applications in place using the built-in Tomcat servlet container module. In an ordinary (non Sun ONE Application Framework) Web module, to take advantage of this facility, you would have to first prepare the servlet for execution (in this case, the module servlet `MainModuleServlet`) by manipulating execution and URL properties.

Fortunately, the Sun ONE Application Framework tools module has automated everything. All Module Servlets and ViewBeans in the application are already prepared to be run as is.

However, first you need to compile the Web application.

1.3.2 Compile the Web Application

1. Right-click the Classes folder.



```
/** DOC BLOCK */
public void handleLc
    throws ServletE
{
  {
    // Retrieve the
      String loginNa
      String passwo
```

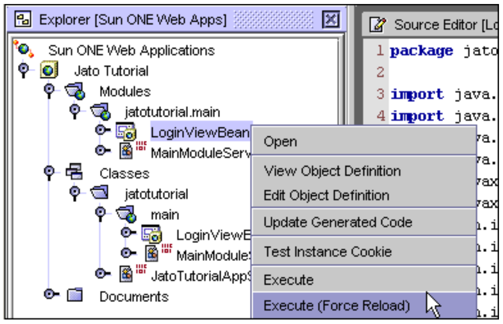
2. Select Compile All from the pop-up menu.

   The Output Window (compiler) at the bottom of the page displays the message *Finished*.

   If you followed all of the tutorial instructions, the Web application compiles without error.

1.3.3 Test Run the Login Page

1. Right-click LoginViewBean.



```
package jato

import java.
import java.
```

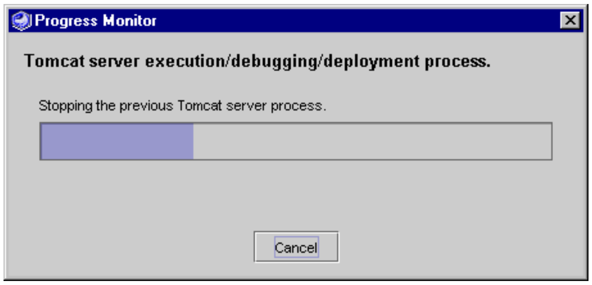
2. Select Execute.

   This deploys and executes the application.

NOTE The Execute (Force Reload) option forces Tomcat to reload all resources (JSPs, classes, and so on). It is just like restarting Tomcat.

For some browsers, you might have to close all instances of that browser before you can rerun any page in your application.

The status page appears.



Tomcat server execution/debugging/deployment process.

Stopping the previous Tomcat server process.

A default browser starts the application.

Test a Successful Login

1. Enter a valid login (for example, `1`, `777` or `410`).

NOTE Use `1`, `777`, and `410`, and hard-coded valid customer numbers. Later you login with a customer number that exists in the database so that you can retrieve and update the customer's record.

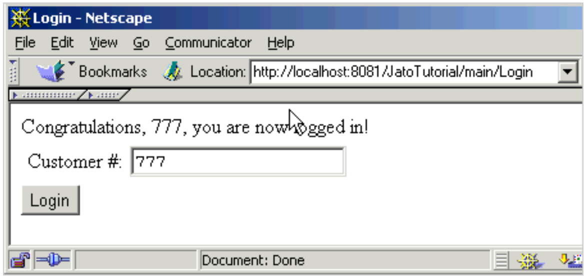
2. Click Login.

CAUTION If you press the enter key while in the text field, the form will be submitted for you. However, the server won't know which button such a submit should address. The *<jato:form>* tag provides an attribute *defaultCommandChild* that can be used to tell the server which button should be activated in the default case.

Refer to the tag library documentation for more information on this feature.

However, for now, just click the button directly.

The login page should refresh displaying the success message.


Login - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8081/JatoTutorial/main/Login

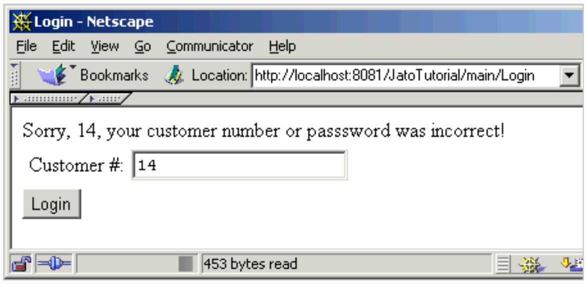
Congratulations, 777, you are now logged in!

Customer #: 777

Test an Unsuccessful Login

1. Enter an invalid login name (for example, `foo`, `8` or `14` - anything other than the valid, hard-coded customer numbers described above).
2. Click Login.

   The login page should refresh displaying the failure message.


Login - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://localhost:8081/JatoTutorial/main/Login
453 bytes read

Sorry, 14, your customer number or passsword was incorrect!

Customer #: 14

Login

Alternative Runtime Environments

1. If you prefer to test run you application outside of the Sun ONE Studio, compile and package your application into a WAR file and place the WAR file in the `webapps` directory (this varies from container to container, but most call it `webapps`).
2. Open a browser and run it with the URL appropriate to the servlet container.

   The only possible variation is the page name (Login) at the end of the URL.

   - Apache Tomcat or Caucho Resin

     `http://localhost:8080/JatoTutorial/module1/Login`

   - Sun ONE Application Server 6.x

     `http://localhost/NASApp/JatoTutorial/module1/Login`

NOTE You may find it necessary to refer to this task again during this tutorial.

Chapter 7

Tutorial—Section 2.1 Prepare Application to Access SQL Database

This section describes how to expand the application and prepare the Sun™ ONE Application Framework application to access a SQL Database.

You expand the existing application by adding a SQL-based model and a page to display that model's data. You then link the two application pages together so they show coordinated data.

Task 1: Accessing a SQL Database

2.1.1 Connect to the Sample Database

Connect to the Sample Database Before Proceeding

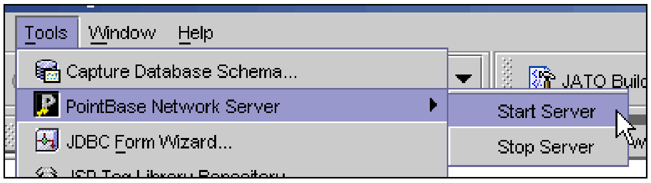
NOTE

- The remainder of the tutorial assumes the presence of an RDBMS database which is used as a prerequisite for introducing you to some additional Application Framework features.

  There is no requirement for an Application Framework application to access an RDBMS. Therefore, your actual applications may not access an RDBMS, but rather some other enterprise system that requires another form of preparation, setup, and connection.

- The step that follows (starting the PointBase Network Server) uses a Sun ONE Studio tool that is not actually a part of the Sun ONE Application Framework toolset module. However, the sample database, the PointBase Network Server, and the tools to connect to it are included with all of the various versions of the Sun ONE Studio.

Select the menu option Tools-->PointBase Network Server-->Start Server from the Sun ONE Studio to start the PointBase Network Server (database server).



2.1.2 Create a Datasource

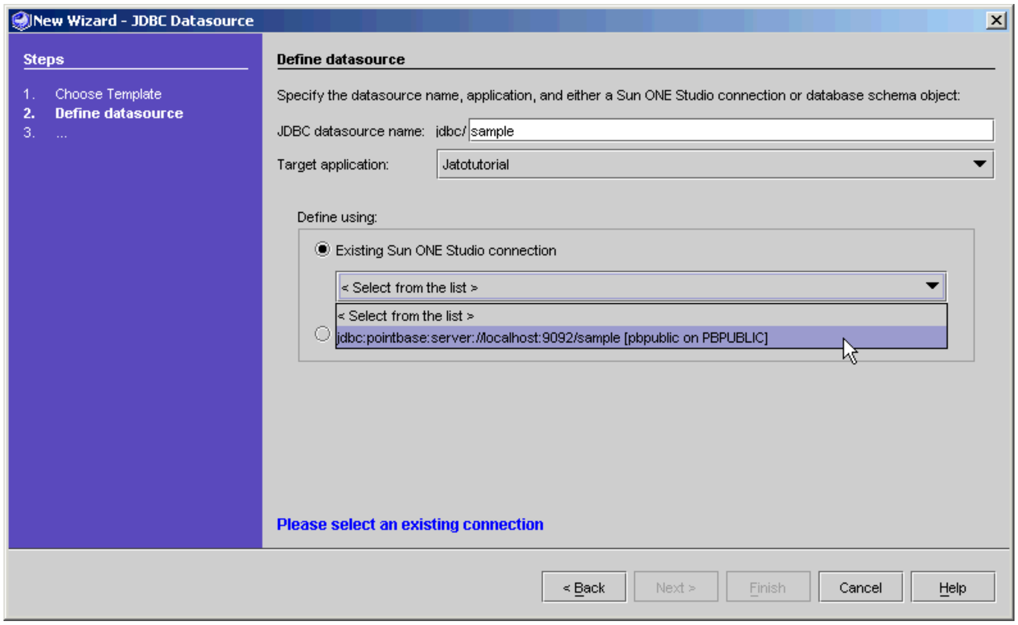
Create a JDBC Datasource using the Sun ONE Application Framework JDBC Datasource wizard.

1. Select File-->New-->Sun ONE Application Framework-->JDBC Datasource.

   The Choose Template page appears.

2. Click Next.

   The Define datasource page appears.



Define datasource

Specify the datasource name, application, and either a Sun ONE Studio connection or database schema object:

JDBC datasource name: jdbc/ sample

Target application: Jatotutorial

Define using:

Existing Sun ONE Studio connection

< Select from the list >

< Select from the list >

jdbc:pointbase:server://localhost:9092/sample [pbpublic on PBPUBLIC]

Please select an existing connection

3. Enter *sample* in the New datasource name textbox.

4. In the Select connection combo box, select the PointBase sample database jdbc connection, which is the only connection option available.

5. Click Finish.

   A dialog box appears confirming that the jdbc/sample datasource has been properly saved.

   You do not find a datasource file, node, or property anywhere. It is only needed when creating JDBC SQL Models.

   The JDBC SQL wizard presents you with a selection of the datasources that you have created.

2.1.3 Tomcat SQL Connection Preparation

NOTE

- If you are using the Sun ONE Application Server to run your tutorial application, this step is optional.
- If you are using the built-in Tomcat engine, or running the tutorial application in another servlet container that does not support JNDI, then you need to make a few minor modifications to the `SQLConnectionManagerImpl` class in your application.

1. Expand the Classes folder.
2. Expand the `jatotutorial` package folder.
3. Double-click the `SQLConnectionManagerImpl` class to open it.

   There is not much code in here.

   You need to make three modifications.

   At the beginning of the static initializer code, you see a line of code that looks like the following:

   ```
   setUsingJNDI(true);
   ```
4. Change *true* to *false*.

   The container will then use JDBC URLs instead of JNDI lookups.

   Below that code, a block of disabled code appears.

   The following table contains that block of disabled code:

```
/*
 * enable this code block and add the necessary jdbc drivers
entries in the try block
    try
    {
    // load the PointBase JDBC driver
        Class.forName("com.pointbase.jdbc.jdbcUniversalDriver");
    }
    catch(ClassNotFoundException e)
    {
        e.printStackTrace();
    }
*/
```

5. Delete the enclosing comment markers (`/*`, `*` and `*/`) so that the JDBC driver will be loaded.

   Just before the end of the static initializer code block, there is a line of code as follows:

   ```
   // addDataSourceMapping(“jdbc/sample”,
   “jdbc:PointBase://localhost:9092/sample”);
   ```

6. Remove the comment marker (`//`) to enable this line of code.

   Your application will now use a JDBC URL directly to make a connection to the database, instead of using the connection pooling via JNDI.

Chapter 8

Tutorial—Section 2.2 Create the CustomerModel

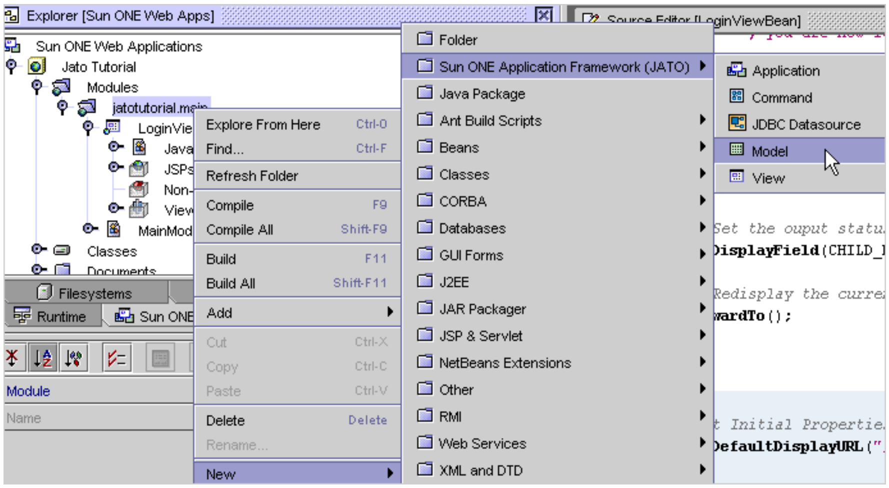
This section describes how to create a model to access the RDBMS in the Sun™ ONE Application Framework application.

Task 2: Create the CustomerModel

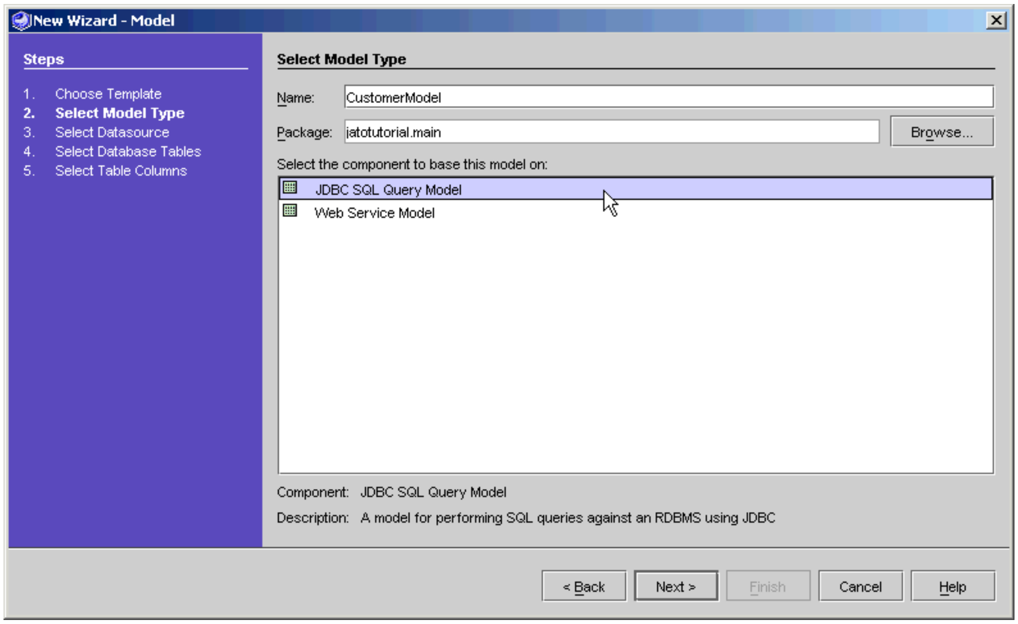
2.2.1 Create a JDBC SQL Model

1. Right-click the *main* module folder.

2. Select the menu option New-->Sun ONE Application Framework-->Model.



The Choose Model Type page appears.



Name: CustomerModel

Package: jatotutorial.main

Select the component to base this model on:

JDBC SQL Query Model

Web Service Model

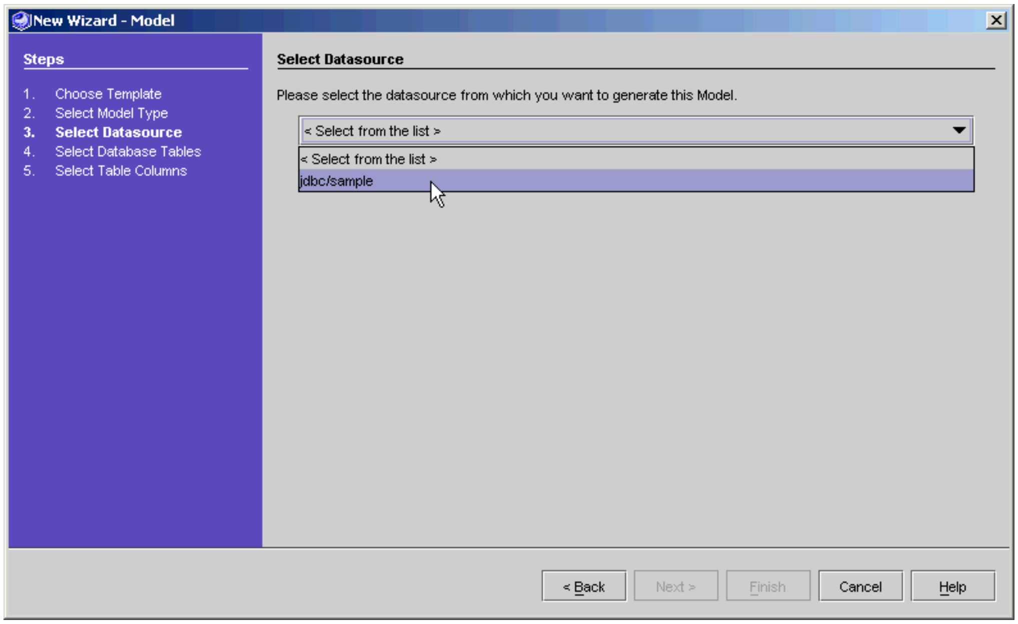
Component: JDBC SQL Query Model

Description: A model for performing SQL queries against an RDBMS using JDBC

3. Enter *CustomerModel* in the Name field.

4. Select JDBC SQL Query Model from the model component list.
5. Click Next.

   The Select Datasource page appears.



Select Datasource

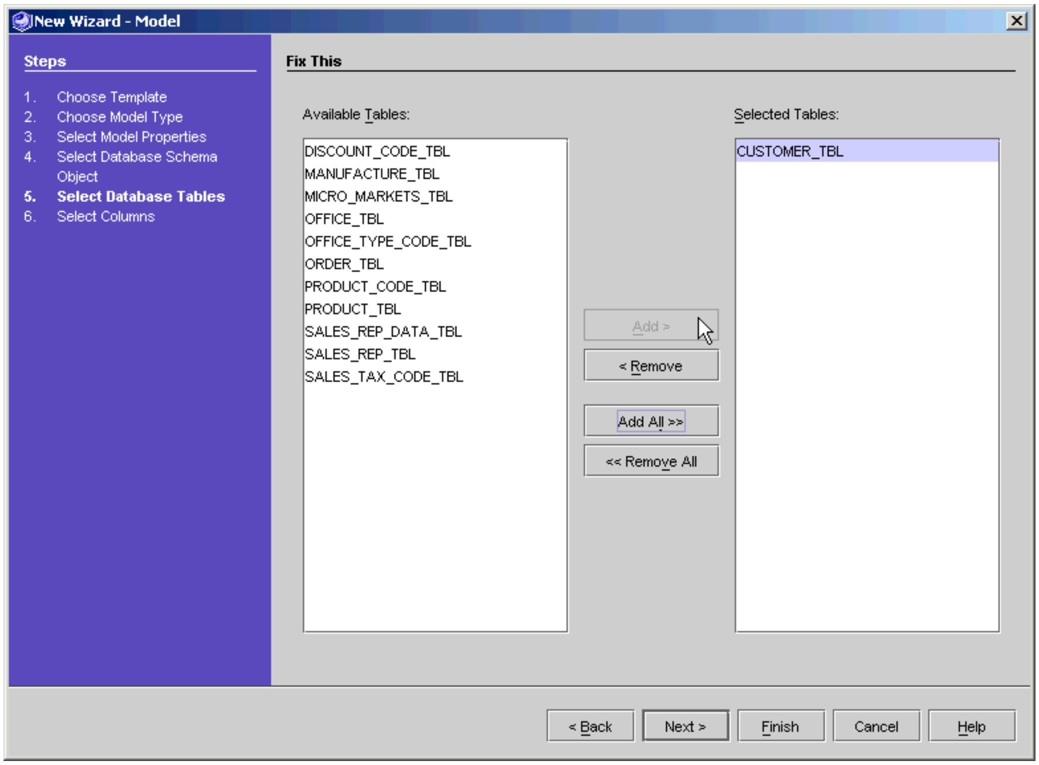
Please select the datasource from which you want to generate this Model.

< Select from the list >

< Select from the list >
jdbc/sample

6. Select *jdbc/sample* from the Existing Datasource combo box.
7. Click Next.

   The Select Database Tables page appears.



Available Tables:

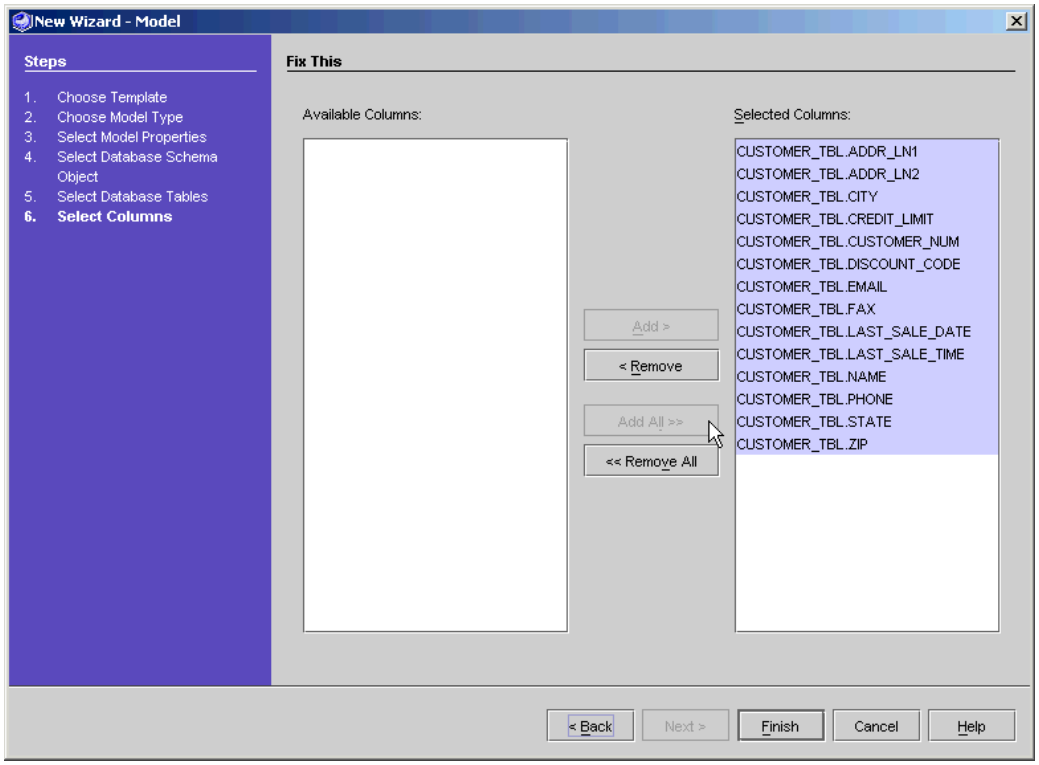
DISCOUNT_CODE_TBL
MANUFACTURE_TBL
MICRO_MARKETS_TBL
OFFICE_TBL
OFFICE_TYPE_CODE_TBL
ORDER_TBL
PRODUCT_CODE_TBL
PRODUCT_TBL
SALES_REP_DATA_TBL
SALES_REP_TBL
SALES_TAX_CODE_TBL

Selected Tables:

CUSTOMER_TBL

8. Select `CUSTOMER_TBL`.
9. Click Add.
10. Click Next.

    The Select Columns page appears.

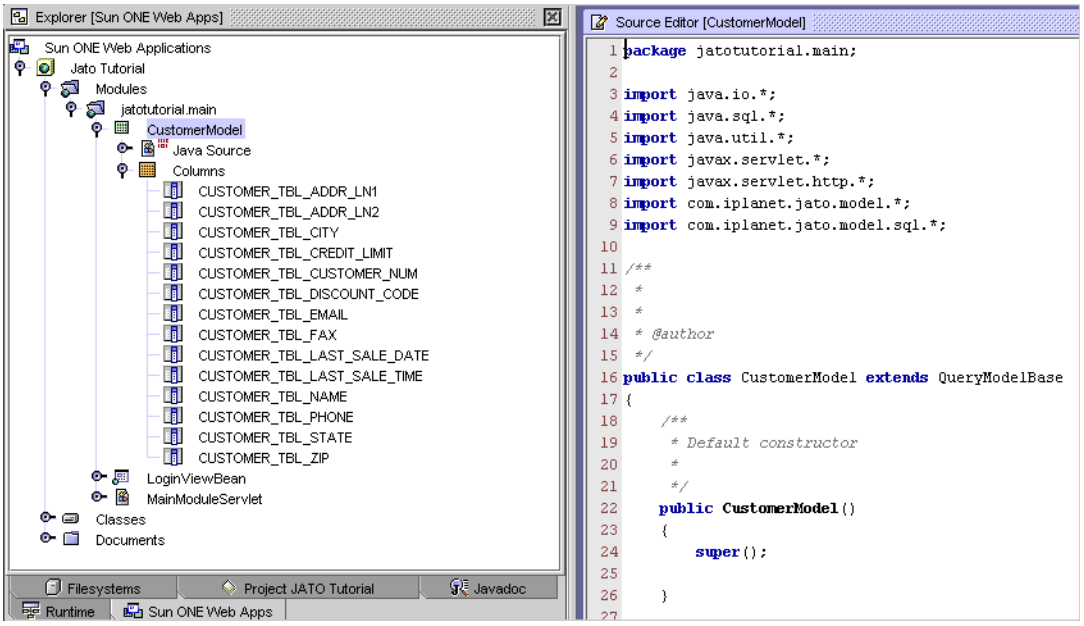


CUSTOMER_TBL.ADDR_LN1
CUSTOMER_TBL.ADDR_LN2
CUSTOMER_TBL.CITY
CUSTOMER_TBL.CREDIT_LIMIT
CUSTOMER_TBL.CUSTOMER_NUM
CUSTOMER_TBL.DISCOUNT_CODE
CUSTOMER_TBL.EMAIL
CUSTOMER_TBL.FAX
CUSTOMER_TBL.LAST_SALE_DATE
CUSTOMER_TBL.LAST_SALE_TIME
CUSTOMER_TBL.NAME
CUSTOMER_TBL.PHONE
CUSTOMER_TBL.STATE
CUSTOMER_TBL.ZIP

11. Click Add All to include all of the columns in your Model.

12. Click Finish to create the Model.

The Customer model object is created in the main module.



```
package jatotutorial.main;

import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.model.*;
import com.iplanet.jato.model.sql.*;

/**
 *
 *
 * @author
 */
public class CustomerModel extends QueryModelBase
{
    /**
     * Default constructor
     *
     */
    public CustomerModel()
    {
        super();

    }
```

13. Expand the CustomerModel so that you can see all of the columns.

14. Double-click the CustomerModel folder to view the code in the CustomerModel Java class.

2.2.2 Mark the Model’s Key Field(s)

NOTE Due to a special key field indicator in the PointBase database, the Model wizard does not properly detect the key field CUSTOMER_TBL_CUSTOMER_NUM. Therefore, you must set the key field manually.

This will be corrected in the next update of the Sun ONE Application Framework tools module.

1. Select the CUSTOMER_TBL_CUSTOMER_NUM model field.

2. In the property sheet, select the Model Field Properties tab.

3. Change the value of the Key Field property from *false* to *true.*

2.2.3 Add Connection Code for Non-JNDI Enabled Containers

For servlet containers that do not support JNDI data sources, or for rapid prototyping work, as in this tutorial, you can rely on explicit use of a JDBC driver.

Note that in section 2.1 of this tutorial, you disabled the use of JNDI and declared the explicit use of the PointBase JDBC driver in the SQLConnectionManagerImpl class.

Now, you will set the connection username and password explicitly in the model so that a proper database connection can be opened during model execution.

NOTE For production environments, you should use JNDI connections.

The following table contains two lines of code (shown in bold) that you need to add to the CustomerModel's constructor:

```
public CustomerModel()
{
    super();
    setDefaultConnectionUser("pbpublic");
    setDefaultConnectionPassword("pbpublic");
}
```

Chapter 9us

Tutorial—Section 2.3 Create Customer Page

This section describes how to create a page in the Sun™ ONE Application Framework that displays data it gets from a model that accesses data from an RDBMS.

Task 3: Create the Customer Page

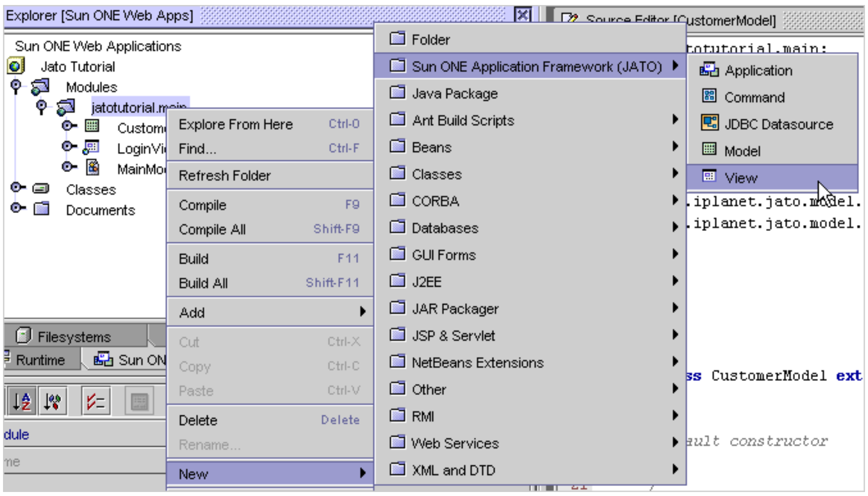
You will now create the second page of the application.

However, this page will *bind* to a model (to model fields). This binding process automatically creates on the page display fields that will display the data from the model.

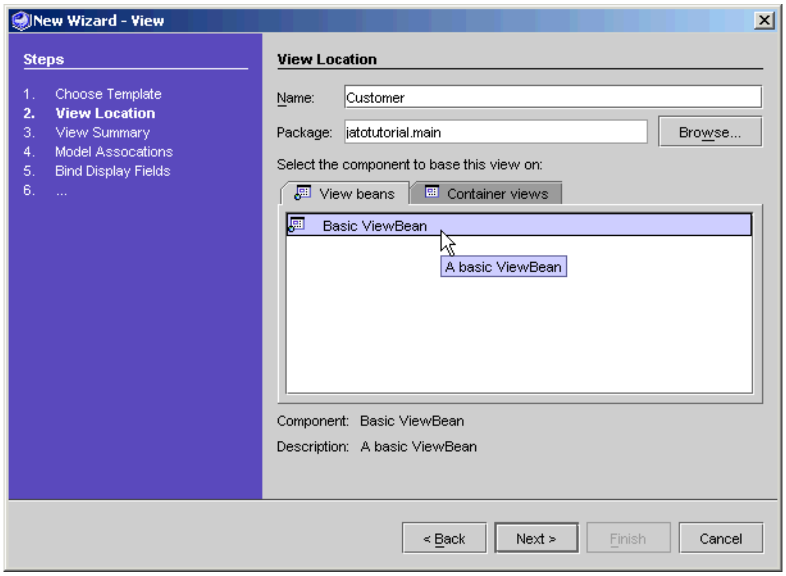
2.3.1 Add a ViewBean

1. Right-click the *main* module from the Sun ONE Studio Explorer page.

2. Select New-->Sun ONE Application Framework-->View.



The View Location page appears.



Steps

1. Choose Template
2. **View Location**
3. View Summary
4. Model Assocations
5. Bind Display Fields
6. ...

View Location

Name: Customer

Package: jatotutorial.main

Select the component to base this view on:

View beans | Container views

Basic ViewBean

A basic ViewBean

Component: Basic ViewBean

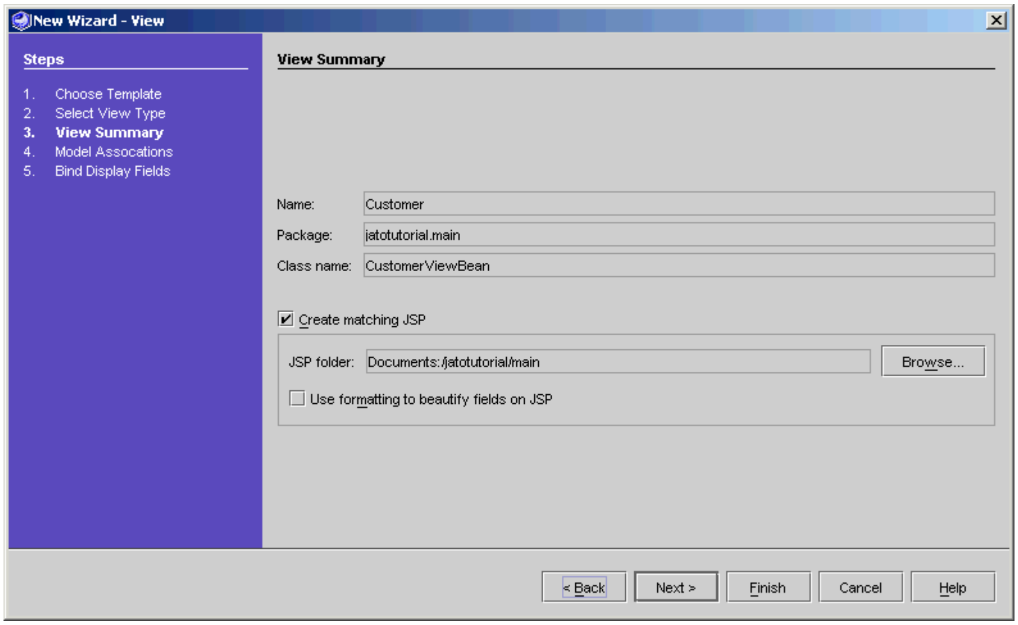
Description: A basic ViewBean

3. Enter Customer in the Name field (to replace <*default*>).

4. Select Basic ViewBean in the View beans tab to create a ViewBean.

5. Click Next.

The View Summary page appears.



Name: Customer

Package: jatotutorial.main

Class name: CustomerViewBean

[✓] Create matching JSP

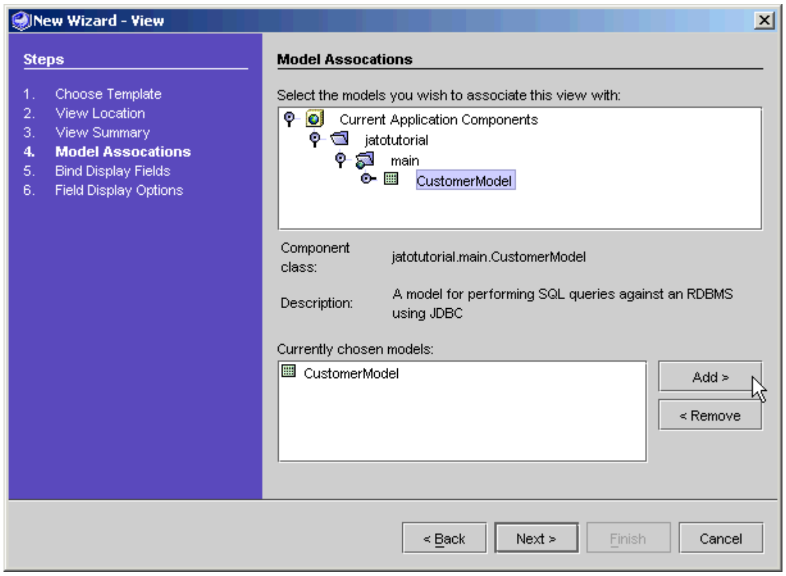
JSP folder: Documents:/jatotutorial/main

[ ] Use formatting to beautify fields on JSP

The Name, Package, and Class name fields are populated as a result of previous entries.

6. Click Next.

The Model Associations page appears.



Select the models you wish to associate this view with:

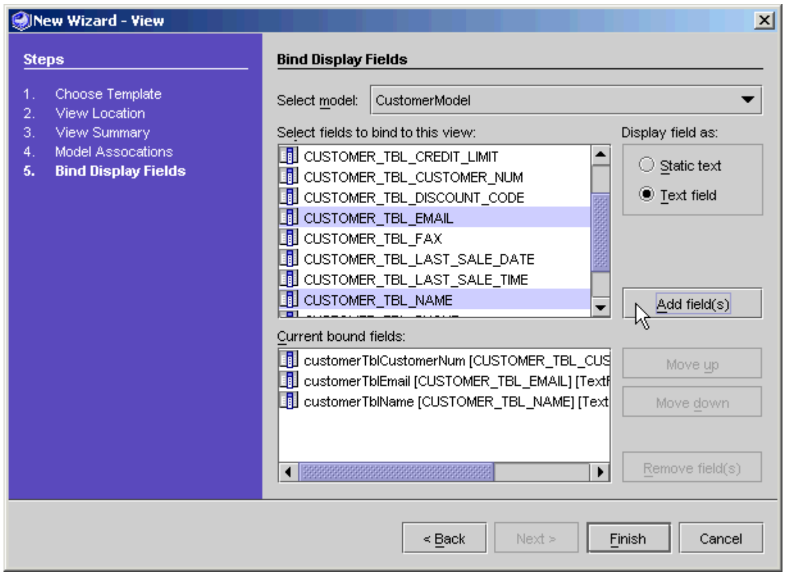
Component class: jatotutorial.main.CustomerModel

Description: A model for performing SQL queries against an RDBMS using JDBC

Currently chosen models: CustomerModel

7. Expand Current Application Components to expose jatotutorial-->main.
8. Select Customer model.
9. Click Add.
10. Click Next.

    The Bind Display Fields page appears.


New Wizard - View
Steps
1. Choose Template
2. View Location
3. View Summary
4. Model Assocations
5. Bind Display Fields
Bind Display Fields
Select model: CustomerModel
Select fields to bind to this view:
CUSTOMER_TBL_CREDIT_LIMIT
CUSTOMER_TBL_CUSTOMER_NUM
CUSTOMER_TBL_DISCOUNT_CODE
CUSTOMER_TBL_EMAIL
CUSTOMER_TBL_FAX
CUSTOMER_TBL_LAST_SALE_DATE
CUSTOMER_TBL_LAST_SALE_TIME
CUSTOMER_TBL_NAME
Display field as:
Static text
Text field
Add field(s)
Current bound fields:
customerTblCustomerNum [CUSTOMER_TBL_CUS
customerTblEmail [CUSTOMER_TBL_EMAIL] [TextF
customerTblName [CUSTOMER_TBL_NAME] [Text
Move up
Move down
Remove field(s)
< Back
Next >
Finish
Cancel

You only want to use three fields.

11. Add the first field:

a. Select the CUSTOMER_TBL_CUSTOMER_NUM field.

Accept the Static text default.

b. Click *Add field(s)*.

The CUSTOMER_TBL_CUSTOMER_NUM field is added to the *Current bound fields* list box.

12. Add the second and third fields simultaneously:

a. Select the CUSTOMER_TBL_EMAIL and CUSTOMER_TBL_NAME fields (hold down the Ctrl key to select multiple non-sequential fields).

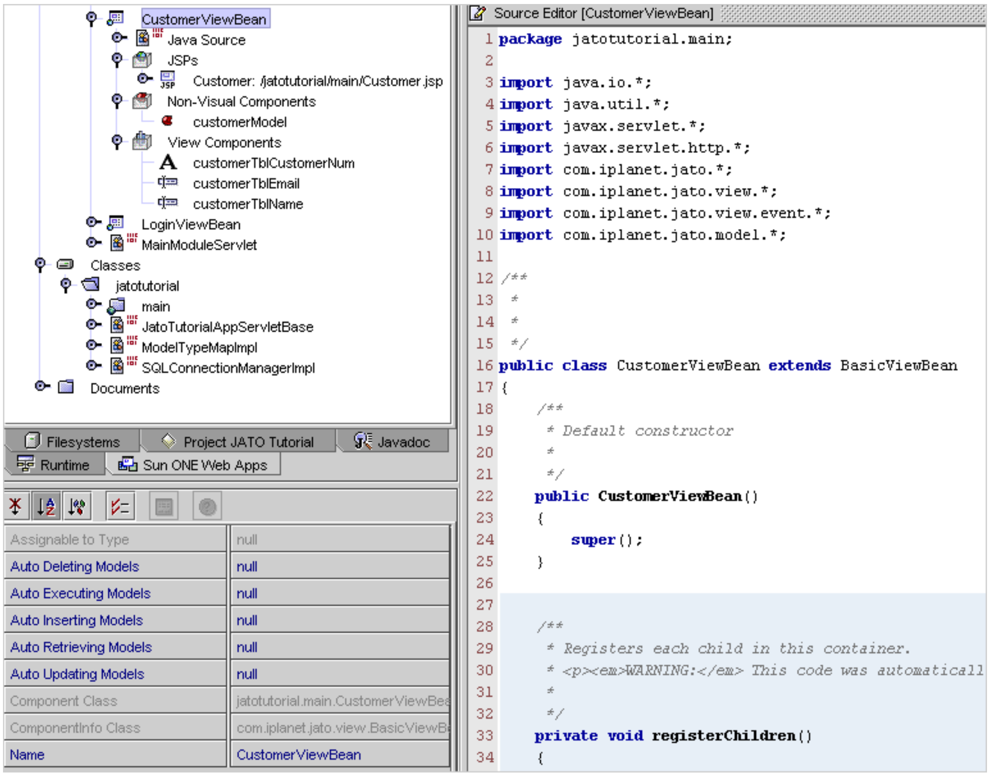
b. Select *Text field*.

c. Click *Add field(s)*.

The CUSTOMER_TBL_EMAIL and CUSTOMER_TBL_NAME fields are added to the *Current bound fields* list box.

13. Click Finish.

You have created the ViewBean.



Assignable to Type	null
Auto Deleting Models	null
Auto Executing Models	null
Auto Inserting Models	null
Auto Retrieving Models	null
Auto Updating Models	null
Component Class	jatotutorial.main.CustomerViewBea
ComponentInfo Class	com.iplanet.jato.view.BasicViewB
Name	CustomerViewBean

```
package jatotutorial.main;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.iplanet.jato.*;
import com.iplanet.jato.view.*;
import com.iplanet.jato.view.event.*;
import com.iplanet.jato.model.*;

/**
 *
 *
 */
public class CustomerViewBean extends BasicViewBean
{
    /**
     * Default constructor
     *
     */
    public CustomerViewBean()
    {
        super();
    }


    /**
     * Registers each child in this container.
     * <p><em>WARNING:</em> This code was automaticall
     *
     */
    private void registerChildren()
    {
```

14. Double-click CustomerViewBean.

The code appears in the right-hand panel.

NOTE Like the LoginViewBean, a JSP for the CustomerViewBean was added to the Documents folder (`/jatotutorial/main`), and there is a link to that JSP under this ViewBean's JSPs folder.

You see three display fields that were created because you indicated that you wanted to bind to the CustomerModel's fields. This allows data to automatically be displayed on the Customer page.

You also see an entry under the Non-Visual Components node which is a reference to the CustomerModel class.

2.3.2 Making a Model Auto Retrieve

Finally, make the CustomerModel *auto retrieve* when the Customer page is requested for display.

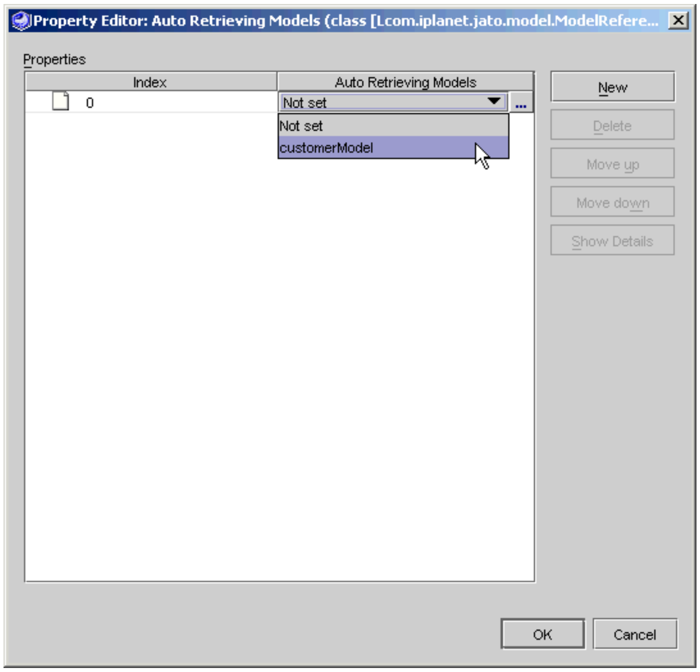
You accomplish this by populating the ViewBean's Auto Retrieving Models property with the appropriate model reference—in this case, the customerModel reference.

1. Select the CustomerViewBean node.
2. Click the value area for the Auto Retrieving Models property.

   The "..." button appears.
3. Click "...".

   The Auto Retrieving Models custom editor launches.

   Note that the Properties area is blank when this editor first appears.



Property Editor: Auto Retrieving Models (class [Lcom.iplanet.jato.model.ModelRefere...

Properties

Index	Auto Retrieving Models
0	Not set

Not set
customerModel

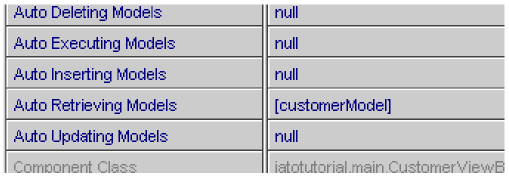
4. Click New.

   This adds an entry.

5. Select customerModel from the Auto Retrieving Models combo box.

6. Click OK.

   This sets the property.



Auto Deleting Models	null
Auto Executing Models	null
Auto Inserting Models	null
Auto Retrieving Models	[customerModel]
Auto Updating Models	null
Component Class	iatotutorial.main.CustomerViewB

NOTE You could have written a few lines of code to accomplish the same goal.

Commonly, this code would be implemented in the CustomerViewBean's beginComponentDisplay event.

2.3.3 Add a Button View Component

1. Add a button to the CustomerViewBean.

   The following table contains the specifications for adding a button to the CustomerViewBean.

Type	Name	Initial Value
Button	update	Object Type: String Object Value: Update

2. Enable the button to update the customer record.

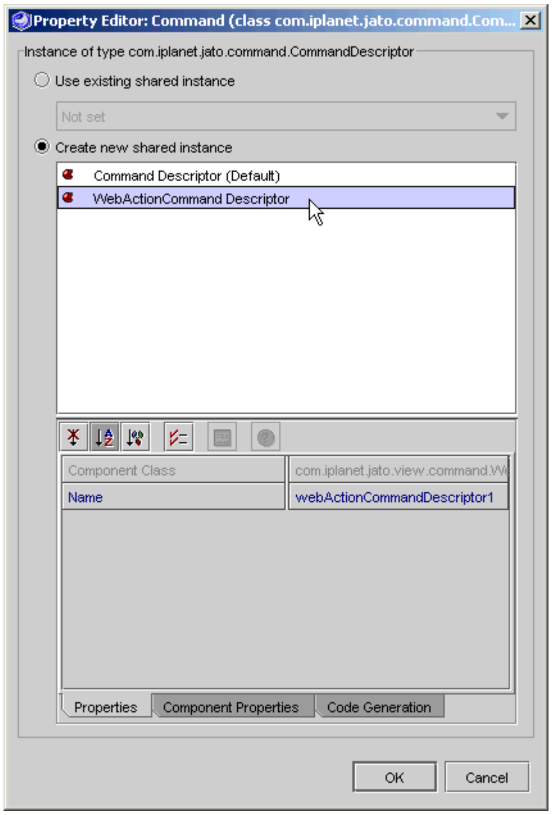
   a. Select the *update* button field.

   b. Click in the value area of the Command Descriptor property.

      The "..." button appears.

c. Click “...”.

The Command Descriptor editor launches.



Property Editor: Command (class com.iplanet.jato.command.Com...

Instance of type com.iplanet.jato.command.CommandDescriptor

( ) Use existing shared instance

Not set

(●) Create new shared instance

Command Descriptor (Default)

WebActionCommand Descriptor

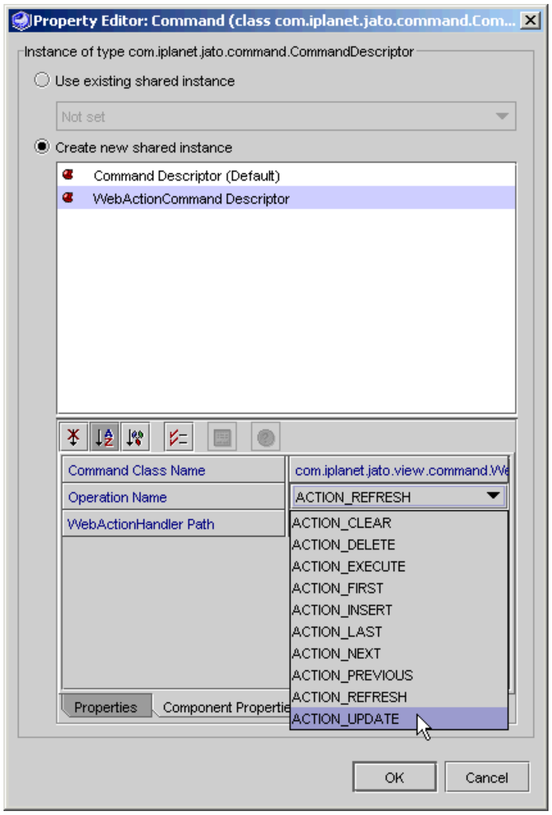
Component Class	com.iplanet.jato.view.command.We
Name	webActionCommandDescriptor1

The *Use existing shared instance* option is the default selection.

3. Select *Create new shared instance.*

4. Select WebActionCommand Descriptor from the list.

5. Select the Component Properties tab at the bottom of the editor.


Property Editor: Command (class com.iplanet.jato.command.Com...

Instance of type com.iplanet.jato.command.CommandDescriptor

○ Use existing shared instance

Not set

● Create new shared instance

- Command Descriptor (Default)
- WebActionCommand Descriptor

Command Class Name	com.iplanet.jato.view.command.We
Operation Name	ACTION_REFRESH
WebActionHandler Path	

ACTION_CLEAR
ACTION_DELETE
ACTION_EXECUTE
ACTION_FIRST
ACTION_INSERT
ACTION_LAST
ACTION_NEXT
ACTION_PREVIOUS
ACTION_REFRESH
ACTION_UPDATE

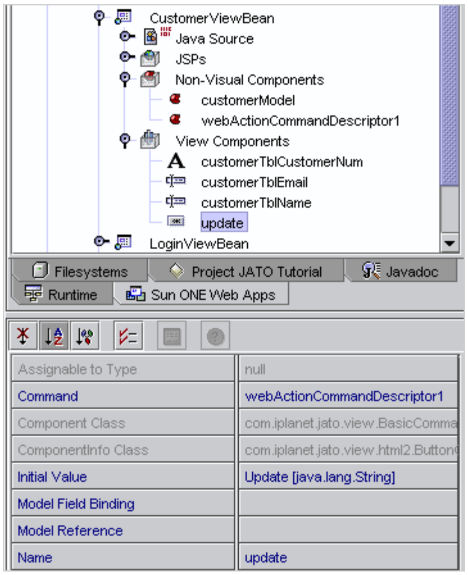
6. Select ACTION_UPDATE for the Operation Name property.

Accept the defaults for the other two properties.

7. Click OK.

You have finished setting this property.

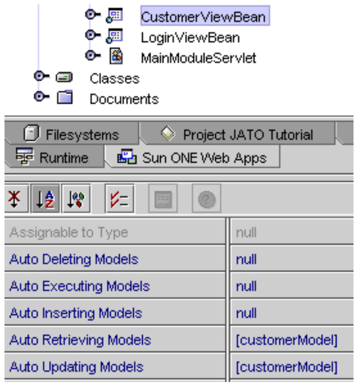
NOTE A new entry is added under the Non-Visual Components node, and the Command Descriptor property is set.



Assignable to Type	null
Command	webActionCommandDescriptor1
Component Class	com.iplanet.jato.view.BasicComma
ComponentInfo Class	com.iplanet.jato.view.html2.Button
Initial Value	Update [java.lang.String]
Model Field Binding	
Model Reference	
Name	update

Now you need to add the customerModel reference as an Auto Updating Model on the CustomerViewBean.

This is exactly the same procedure that you followed to set the Auto Retrieving Models property in an earlier step. See the section 2.3.2 Making a Model Auto Retrieve.



Assignable to Type	null
Auto Deleting Models	null
Auto Executing Models	null
Auto Inserting Models	null
Auto Retrieving Models	[customerModel]
Auto Updating Models	[customerModel]

NOTE You could have implemented some code in the update button's handleUpdateRequest event to accomplish this task.

You might want to do this in a more complex, real-world application.

2.3.4 Add a Hidden Field

2.3.4 Add a Hidden Field Tag to the JSP

Add a hidden field tag in the JSP to preserve the value of the key field customerNum to ensure proper update logic.

In the Customer.jsp file, the customerNum field is associated with a jato:text tag. Jato:text tags render their values as static, non-editable HTML content, not as HTML input fields. Therefore, the customerNum value will be displayed in the rendered document, but it will not be submitted back to the server when the form is posted.

However, the update logic needs the value of the key field in order to limit the update to a single database row. This value must be posted back along with the other input field values so that you can perform an update on the proper customer record. To achieve this, you will preserve the customerNum field value in a hidden field, which will be posted back on form submit.

CAUTION If you neglect this step, no key field value is submitted with the form. The resulting JDBC update statement would lack a WHERE clause and therefore result in the unintentional modification of the entire table.

1. Copy the `jato:text` tag for the customerNum field tag and paste it just below.
2. Change this copy from a `jato:text` to a `jato:hidden`.

   Leave the name attribute the same.

   There is no need to modify anything in the peer ViewBean. Both tags use the same peer display field.

Chapter 10

Tutorial—Section 2.4 Test Run the Customer Page

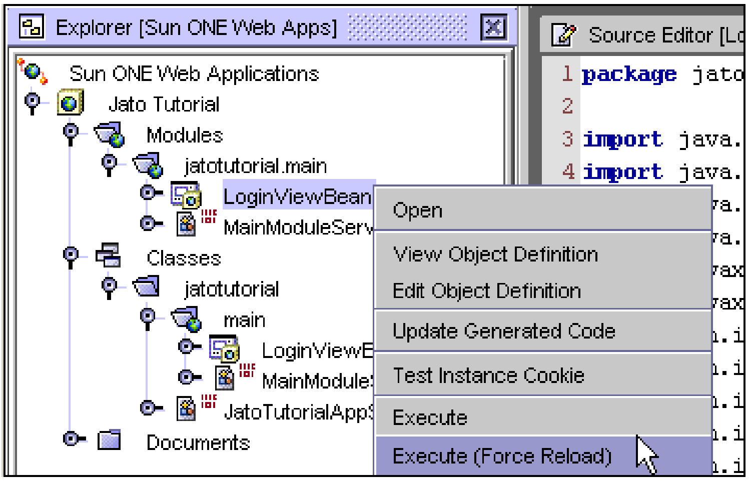
This section describes how to run your Sun™ Application Framework application.

Task 4: Test Run the Customer Page

NOTE Make sure that the PointBase Network Server is running.

---

1. Right-click CustomerViewBean.



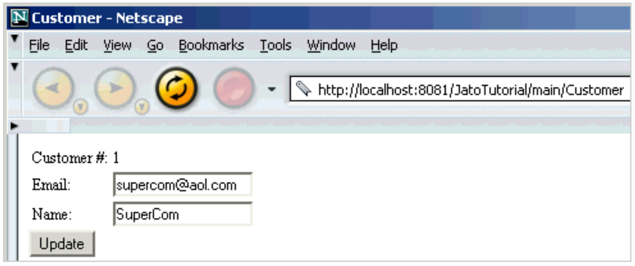
```
package jato

import java.
import java.
```

2. Select Execute (Force Reload).

This ensures that Tomcat restarts and picks up all changes.

A default browser starts the application.


Customer - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost:8081/JatoTutorial/main/Customer
Update

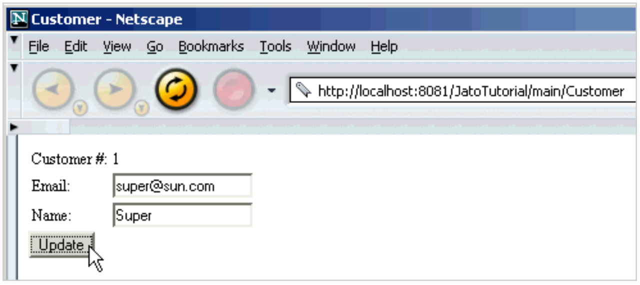
Customer #: 1
Email: supercom@aol.com
Name: SuperCom

Test a Customer Update

1. Make a change to some or all of the fields.

In the screenshot, the email name, domain, and customer name were changed.

2. Click Update.


Customer - Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost:8081/JatoTutorial/main/Customer
Update

Customer #: 1
Email: super@sun.com
Name: Super

Chapter 11

Tutorial—Section 2.5 Link Login Page to Customer Page

This section describes how to link the Login page to the Customer page in the Sun™ ONE Application Framework application, filtering the data the Customer page displays based on the user's login.

Task 5: Link the Login Page to the Customer Page

Edit the handleLoginRequest Method in LoginViewBean

Edit the `LoginViewBean.java` file.

Modify the logic in the `handleLoginRequest()` method as shown in the code example below so that in the event of a successful login, the Customer page displays with the customer data that corresponds to the value entered in the User Name field.

NOTE

- In the code example below, the only legal values for the User Name field are *also* CustomerID values.

  Therefore, you can take the Login ID value and apply it to the `where` clause used by the `CustomerModel`.

  This ensures that the data retrieved by the `CustomerModel` corresponds to the appropriate CustomerID.

- Make code changes cautiously.

  The code that appears below practically replaces all of the code that appeared previously in this event.

  Adding just what appears to be the delta likely leads to errors.

The following table contains the code you need to enter to modify the logic in the `handleLoginRequest()` method.

```
public void handleSubmitRequest(RequestInvocationEvent event)
  throws ServletException, IOException
{
  // Retrieve the customer number
  String custNum = getDisplayFieldStringValue(CHILD_CUSTOMER_NUM);

  String theMessage = "";

  // Check the customer number
  // Note, we don't check the password in this example
  if (custNum.equalsIgnoreCase("1") ||
    custNum.equals("777") ||
    custNum.equals("410"))
  {
    // Instead of returning the login page, display the Customer
    // page for the customer that matches the customer number
    // theMessage = "Congratulations, " + custNum + ...

    // Get a reference to the CustomerModel
    CustomerModel model =
      (CustomerModel)getModel(CustomerModel.class);

    // Modify the where criteria to reflect the customer number used to login
    model.clearUserWhereCriteria();
    model.addUserWhereCriterion(
      "CUSTOMER_TBL_CUSTOMER_NUM", new Integer(custNum));

    // Display the Customer page
    getViewBean(CustomerViewBean.class).forwardTo(event.getRequestContext());
  }
  else
  {
    theMessage = "Sorry, "+ custNum +
           “,your customer number was incorrect!”;

          // Set the output status message
          getDisplayField(CHILD_MESSAGE).setValue(theMessage);
    forwardTo();
  }
  }
```

Chapter 12

Tutorial—Section 2.6 Run Application

This section describes how to run the Sun™ ONE Application Framework application now that you have added an additional page to your application and have linked it to the first page.

Task 6: Run the Application

1. Compile the application.
2. Execute (force reload) on the LoginViewBean.
3. Enter a valid customer number (1, 777, or 410).
4. Click Login.

   You should see the Customer page with the customer record that corresponds to the customer number that you used to login.

Index