Sun™ ONE Application Framework Tutorial

Sun™ ONE Studio 5 update 1

Sun Microsystems, Inc.
www.sun.com

Part No. 817-4358-10
October 2003, Revision A

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Contents

Preface 9

How This Book Is Organized 9

Using UNIX Commands 10

Related Documentation 11

Accessing Sun Documentation 11

Contacting Sun Technical Support 12

Sun Welcomes Your Comments 12

1. Before You Begin 13

Primary Features of the Sun ONE Application Framework 13

QA Certification 14

2. Getting Started 15

Introduction 15

Writing Sun ONE Application Framework Applications 16

J2EE/Sun ONE Application Framework Terminology 17

How Sun ONE Application Framework Applications Are Organized 18

About the Sun ONE Application Framework Tutorial 19

3. Tutorial Sections (Links to) 21

Sections 1.1—1.3 21

Sections 2.1—2.6 22

Sections 3.1—3.3 22

Sections 4.1—4.5 23

4. Tutorial—Section 1.1
Application Infrastructure 25

Task 1: New Sun ONE Web Application 25

Create an Application Wizard 25

Application Servlet 30

Module Servlet 31

Advanced Tip - Modules 31

5. Tutorial—Section 1.2
Create Login Page 33

Task 2: Create the Login Page 33

Add a ViewBean 33

Add Display Fields to the Login Page 37

Add Code to the Login Button 42

6. Tutorial—Section 1.3
Test Run the Login Page 45

Task 3: Test Run the Login Page 45

Compile the Web Application 45

Test Run the Login Page 46

Test a Successful Login 48

Test an Unsuccessful Login 48

Alternative Runtime Environments 49

7. Tutorial—Section 2.1
Prepare Application to Access SQL Database 51

Task 1: Accessing a SQL Database 51

Connect to the Sample Database 51

JDBC Datasources 53

Tomcat (and other non-JNDI containers) SQL Connection Preparation 57

8. Tutorial—Section 2.2
Create the CustomerModel 61

Task 2: Create the CustomerMode 61

Create a JDBC™ SQL Model 61

Mark the Model's Key Field(s) 67

Add Connection Code for Non-JNDI Enabled Containers 68

9. Tutorial—Section 2.3
Create Customer Page 71

Task 3: Create the Customer Page 71

Add a ViewBean 71

Add a Button Component 78

Making a Model Auto Update 82

Add a Hidden Field to the Customer Page 84

Format the JSP 88

10. Tutorial—Section 2.4
Test Run the Customer Page 91

Task 4: Test Run the Customer Page 91

Test a Customer Update 92

11. Tutorial—Section 2.5
Link Login Page to Customer Page 95

Task 5: Link the Login Page to the Customer Page 95

Edit the handleLoginRequest Method in LoginPage 95

12. Tutorial—Section 2.6
Run Application 99

Task 6: Run the Application 99

13. Tutorial—Section 3.1
Create a Command Component 103

Task 1: Create a Command Component 103

Create the UserAccessCommand Component 103

Add Code to the execute Method 107

Configure a Button's Command Descriptor 109

14. Tutorial—Section 3.2
Add a Logout Link to the Customer Page 117

Task 2: Add an HREF to a Customer Page 117

Configure an HREF's Command Descriptor 118

Format the HREF tag in the Customer JSP 123

15. Tutorial—Section 3.3
Test Run the Login/Logout Command Component 125

Task 3: Test Run the Login/Logout Command 125

16. Tutorial—Section 4.1
Prepare to Create a Web Service Model 129

Task 1: Web Service User Registration and Downloading 129

Download the Web Service SDK 129

Register to Use the Web Service 130

Create the Web Service Model 130

17. Tutorial—Section 4.2
Create the Google Search Page 135

Task 2: Create the Google Search Page 135

Add a Page Component 135

Add More Visual Components to the Page 142

Enable the Search Button 147

Manual Code Technique 148

Point & Click Technique (code-free) 148

Format the JSP Content 152

18. Tutorial—Section 4.3
Test Run the Google Search Page 155

Task 3: Test Run the Google Search Page 155

Try a Search 156

19. Tutorial—Section 4.4
Add Results Listing to the Google Search Page 157

Task 4: Create a TiledView Pagelet 157

Add a TiledView 157

Configure the TiledView Pagelet Component 164

Getting the Correct Primary Model Dataset Name 166

Add the Pagelet to a Page 169

Formatting the JSP 172

20. Tutorial—Section 4.5
Test Run the Google Search Page 175

Task 5: Test Run the Google Search Page with Results 175

Try a Search 176

Preface

This Sun™ ONE Application Framework 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 Sun ONE Application Framework.

How This Book Is Organized

In the following chapter, you see an overview of the primary features of the Sun ONE Application Framework and toolset (IDE) for enterprise Web application development.

- Chapter 1, "Before You Begin" on page 13.

In the following chapter, you see an outline of the mechanics of using the Sun™ ONE Application Framework tools to build a J2EE Web application.

- Chapter 2, "Getting Started" on page 15.

In the following chapters, you create the application infrastructure needed for all subsequent chapters, and add your first Sun ONE Application Framework page.

- Chapter 4, "Tutorial—Section 1.1 Application Infrastructure" on page 25.
- Chapter 5, "Tutorial—Section 1.2 Create Login Page" on page 33.
- Chapter 6, "Tutorial—Section 1.3 Test Run the Login Page" on page 45.

In the following chapters, 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.

- Chapter 7, "Tutorial—Section 2.1 Prepare Application to Access SQL Database" on page 51.
- Chapter 8, "Tutorial—Section 2.2 Create the CustomerModel" on page 61.
- Chapter 9, "Tutorial—Section 2.3 Create Customer Page" on page 71.
- Chapter 10, "Tutorial—Section 2.4 Test Run the Customer Page" on page 91.
- Chapter 11, "Tutorial—Section 2.5 Link Login Page to Customer Page" on page 95.
- Chapter 12, "Tutorial—Section 2.6 Run Application" on page 99.

In the following chapters, you create a Command component that can be reused by many buttons and HREFs within the same application. This is the alternative technique to implementing request handling code in the button or HREF's handle request event inside its parent container view class.

- Chapter 13, "Tutorial—Section 3.1 Create a Command Component" on page 103.
- Chapter 14, "Tutorial—Section 3.2 Add a Logout Link to the Customer Page" on page 117.
- Chapter 15, "Tutorial—Section 3.3 Test Run the Login/Logout Command Component" on page 125.

In the following chapters, you expand the existing application by adding a Web service-based model and a page to display that model's data. You need to register for and download the Google developer's SDK to build a model for a Web service.

- Chapter 16, "Tutorial—Section 4.1 Prepare to Create a Web Service Model" on page 129.
- Chapter 17, "Tutorial—Section 4.2 Create the Google Search Page" on page 135.
- Chapter 18, "Tutorial—Section 4.3 Test Run the Google Search Page" on page 155.
- Chapter 19, "Tutorial—Section 4.4 Add Results Listing to the Google Search Page" on page 157.
- Chapter 20, "Tutorial—Section 4.5 Test Run the Google Search Page" on page 175.

Using UNIX Commands

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

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

http://docs.sun.com

Related Documentation

Application	Title	Part Number
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Overview, Sun™ ONE Studio 5 update 1*	817-4360-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework IDE Guide, Sun ONE Studio 5 update 1*	817-4104-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Developer's Guide, Sun ONE Studio 5 update 1*	817-4359-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Component Author's Guide, Sun ONE Studio 5 update 1*	817-4362-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Component Reference Guide, Sun ONE Studio 5 update 1*	817-4661-10
Sun ONE Application Framework 2.1	*Sun ONE Application Framework Tag Library Reference, Sun ONE Studio 5 update 1*	817-4361-10

Accessing Sun Documentation

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

http://www.sun.com/documentation

Contacting Sun Technical Support

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

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

Sun Welcomes Your Comments

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

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

Please include the title and part number of your document with your feedback:

Sun ONE Application Framework Tutorial, part number 817-4358-10

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 chapter 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 as follows:

- Turnkey J2EE™ application development
- High performance, proven J2EE framework runtime
- Full component-based development
- Graphical application builder toolset as follows:
  - 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 Sun ONE Application Framework is used by the following:

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

The Sun ONE Application Framework is a valuable tool that does the following:

- Guides naive 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
  - Provides the ability for inexperienced developers to learn J2EE as they build high-performance enterprise applications
- Complements advanced Java/J2EE developers and architects
  - Provides the ability for advanced developers to gain higher productivity by avoiding tedious low-level J2EE development
  - Offers architects well-defined points from which to extend the application architecture
- Accelerates Web Application development and skill/component reuse by providing easy entree into the J2EE API world

This document shows you how to use the Sun ONE Application Framework features to do the following:

- Create a Sun ONE Web Application
- Create a page (ViewBean and TiledViews) and an associated JSP
- Create and use a Mode (JDBC™ SQL and WebService-based models)
- Link pages together

QA Certification

- Solaris 8™ Operating System
- Solaris™ 9 Operating System
- Windows 2000 Operating System
- JavaSoft™ RI and Apache 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 chapter contains the following topics:

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

Introduction

This document 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 Sun ONE Application Framework.

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

Because the Sun ONE Application Framework is foremost a design pattern and a set of interfaces, the examples in this document show only the most basic way of creating a Sun ONE Application Framework application, by extending existing Sun ONE Application Framework implementation base classes and manually constructing certain application objects. This is only one possible way to create a Sun ONE 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 Sun ONE 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 Sun ONE Application Framework.

Second, building an application using these fundamental techniques is a prerequisite to fully understanding the many possible ways to build Sun ONE Application Framework applications. Features that extend the Sun ONE 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 Sun ONE Application Framework core, and you are able to optionally decide not to use them and instead construct your own Sun ONE 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 Sun ONE Application Framework applications, so they become familiar with Sun ONE Application Framework's interactions with applications built on top of the framework, and more fluent in the Sun ONE Application Framework itself.

Writing Sun ONE Application Framework Applications

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

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

J2EE/Sun ONE Application Framework Terminology

There are terms in this document 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 found in this tutorial.

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 the Sun ONE Application Server 7 and Apache Tomcat that 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 Sun ONE Application Framework provides formal application and module entities. A Sun ONE Application Framework application is a base Java package that contains one or more sub-packages (Sun ONE 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 might also customize this behavior separately from other modules.

In J2EE Web application container terms, a Sun ONE 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 Sun ONE Application Framework application.

Avoid over-categorizing your application into several modules simply because the Sun ONE 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 `$instance_dir/applications/j2ee-modules` directory would bear this name. The deployed application name is the same as the name WAR file name.

For example, if you have two Application Framework modules (named *module1* and *module2*) that comprise a Sun ONE Application Framework application, you would call this application *myapp*. The full application package name would be `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`

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 and 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 Framework Tutorial

You will now develop a simple application so you can experience using the Sun ONE Application Framework and its 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.
- Using a WebServiceModel to perform a Google Internet search.
- Displaying the multiple search results of a WebServiceModel.

This tutorial is divided into sections and tasks the steps required to develop the application. Each section 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 chapter outlines the sections contained in this Sun™ ONE Application Framework Tutorial.

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

- Sections 1.1—1.3
- Sections 2.1—2.6
- Sections 3.1—3.3
- Sections 4.1—4.5

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 CustomerMode
- 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

Sections 3.1—3.3

In Sections 3.1 through 3.3, you create a Command component that can be reused by many buttons and HREFs within the same application. This is the alternative technique to implementing request handling code in the button or HREF's handle request event inside its parent container view class.

- Section 3.1

  Task 1: Create a Command Component
- Section 3.2

  Task 2: Add an HREF to a Customer Page
- Section 3.3

  Task 3: Test Run the Login/Logout Command

Sections 4.1—4.5

In Sections 4.1 through 4.5, you expand the existing application by adding a Web service-based model and a page to display that model's data. You need to register for and download the Google developer's SDK to build a model for a Web service.

- Section 4.1

  Task 1: Web Service User Registration and Downloading

- Section 4.2

  Task 2: Create the Google Search Page

- Section 4.3

  Task 3: Test Run the Google Search Page

- Section 4.4

  Task 4: Create a TiledView Pagelet

- Section 4.5

  Task 5: Test Run the Google Search Page with Results

CHAPTER 4

Tutorial—Section 1.1 Application Infrastructure

This chapter describes how to create the Sun™ ONE Application Framework (also known as Application Framework, App Framework, S1AF, and JATO) application infrastructure needed for all subsequent tasks.

Task 1: New Sun ONE Web Application

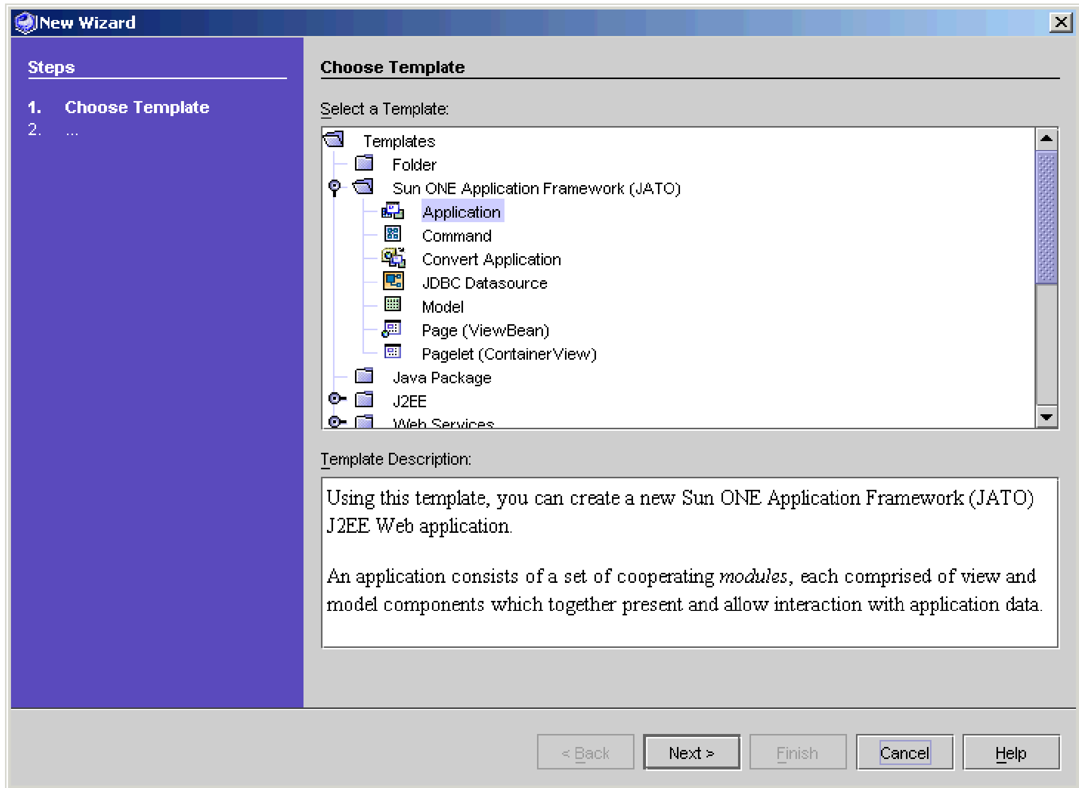
Before developing any pages, you need to create the Sun ONE Application Framework application infrastructure (the WAR directory structure and supporting files). This is a onetime requirement for each Sun ONE Application Framework application.

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 (Studio), you can locate the application anywhere and use the built-in Sun™ ONE Application Server 7 module to test it in place.

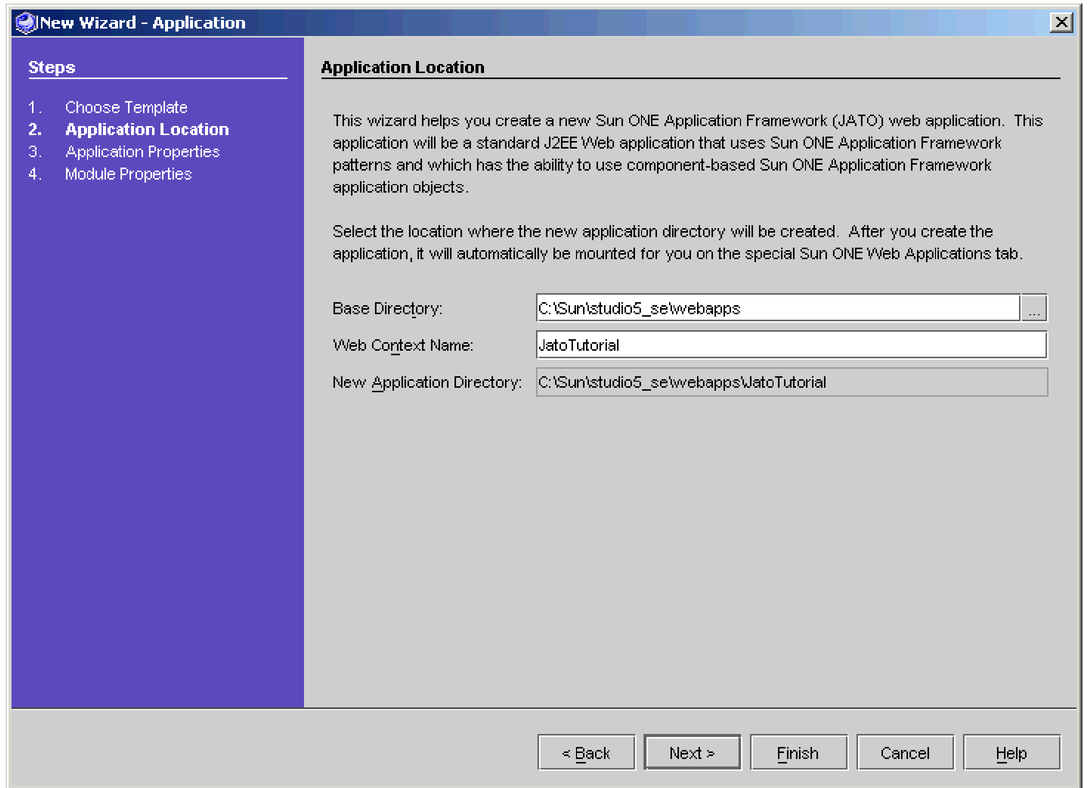
1. **Select the Sun ONE Studio menu option File -> New.**

   The Choose Template panel displays.



2. **Expand the Sun ONE Application Framework folder.**
3. **Select Application.**
4. **Click Next.**

   The Application Location panel displays.



The default base directory is your Sun ONE Studio `user-dir`, which might 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.

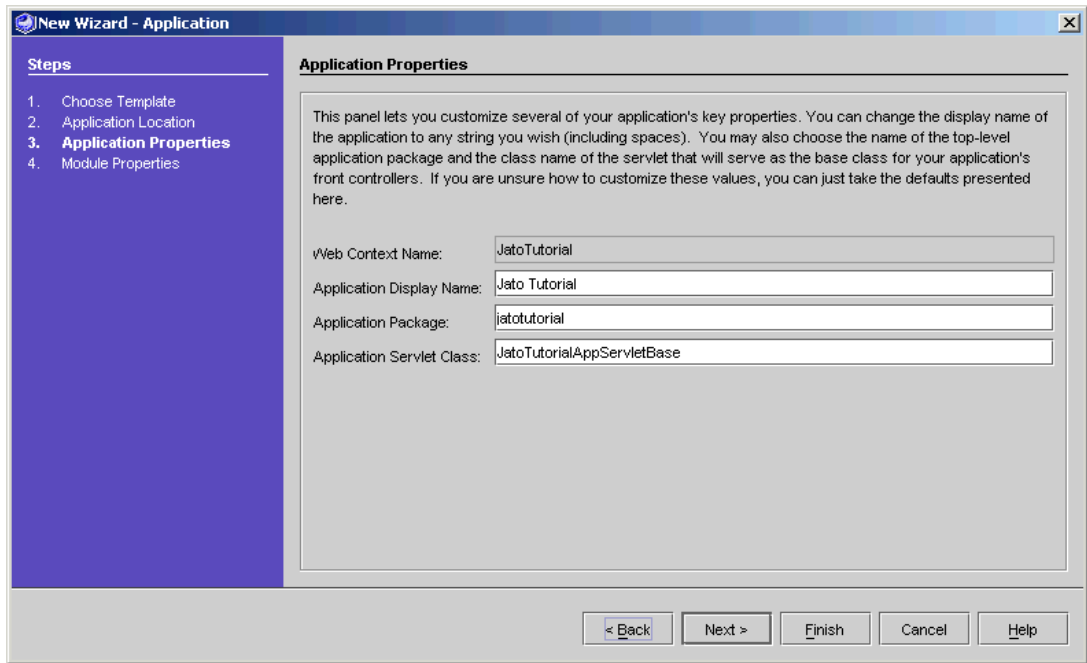
---

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 panel displays.

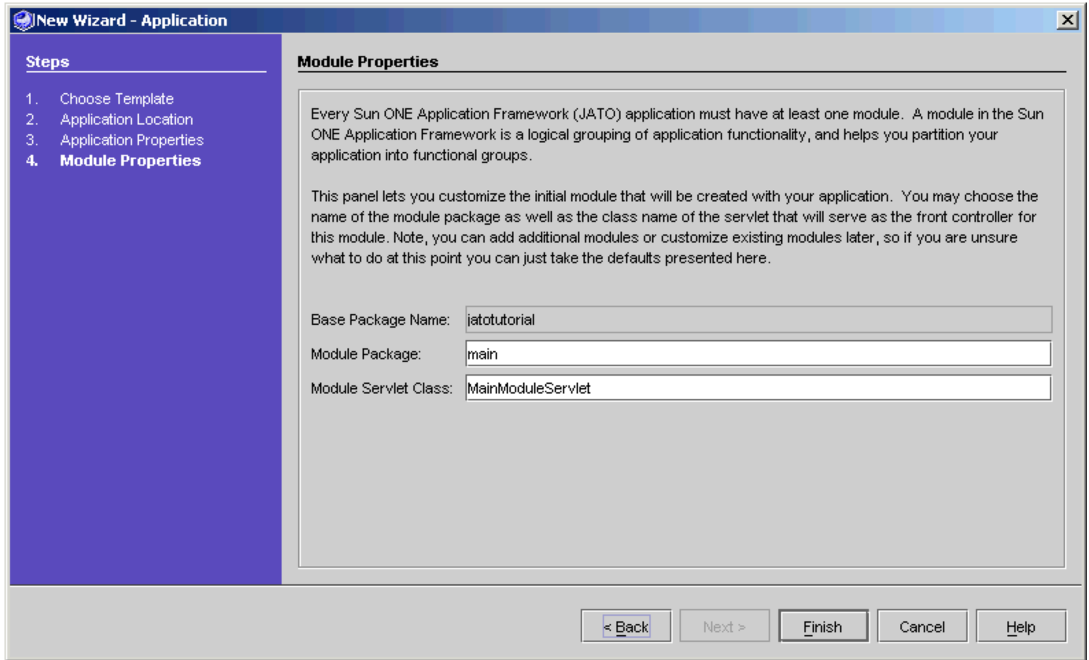


The fields on this panel are populated using the value of the *Web Context Name* field from the previous panel.

For this tutorial, accept the default values.

7. **Click Next.**

The Module Properties panel displays.



For this tutorial, accept the default values.

8. **Click Finish.**

9. **Click OK.**

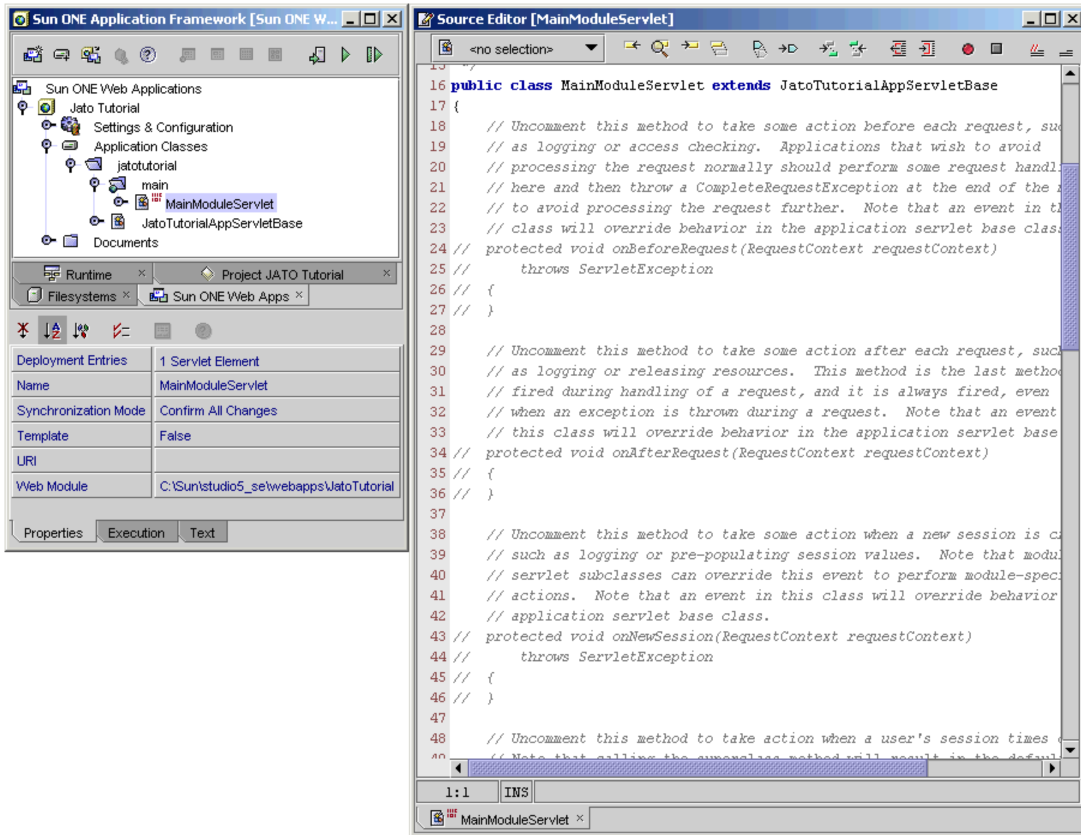
   The application is created.

Note – The processing time depends upon your machine.

The new application displays in the Sun ONE Application tree in the Sun ONE Studio Explorer which is labeled Sun ONE Web Apps.

10. **Expand the modules node in the Sun ONE Web Apps Explorer to see the application layout and observe the code in the two servlet classes that were created.**

    - JatoTutorialAppServletBase
    - MainModuleServlet



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 Sun ONE 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 Sun ONE Application Framework's `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 Sun ONE Application Framework's `com.iplanet.jato.ApplicationServletBase` file. Advanced developers can gain some insight on how requests are handled by reviewing the source code in the `com.iplanet.jato.ApplicationServletBase` class.

Advanced Tip - Modules

Notice that if the `main` module folder is selected, its properties are reflected in the property sheet at the bottom of the Studio Explorer window. Notice that its Module property is *True*. By changing it to *False*, this module becomes an ordinary folder/package, and the entries in the `web.xml` file (a standard Web application configuration file) for the `MainModuleServlet` are removed.

You can make any ordinary folder a Sun ONE Application Framework module by right-clicking the folder and selecting the *Convert to Module* action. You are then prompted to select a Java servlet 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

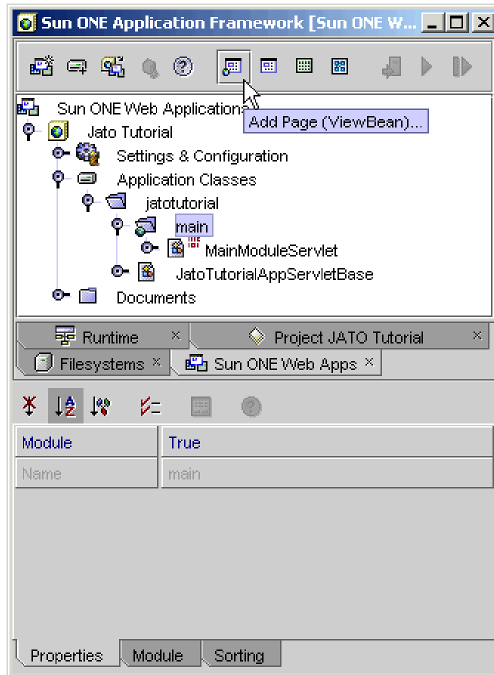
This chapter describes how to add your first Sun™ ONE Application Framework page to the application infrastructure you created.

Task 2: Create the Login Page

Create the first page of the application.

Add a ViewBean

1. **Select the main module folder from the Sun ONE Web Apps Explorer.**



2. **Click the Add Page button on the Sun ONE Application Framework toolbar**

   Or:

   a. **Select the Sun ONE Studio menu option File -> New.**

   b. **Expand Sun ONE Application Framework (JATO) node**

   c. **Select Page (ViewBean)**

   d. **Click Next.**

   Or:

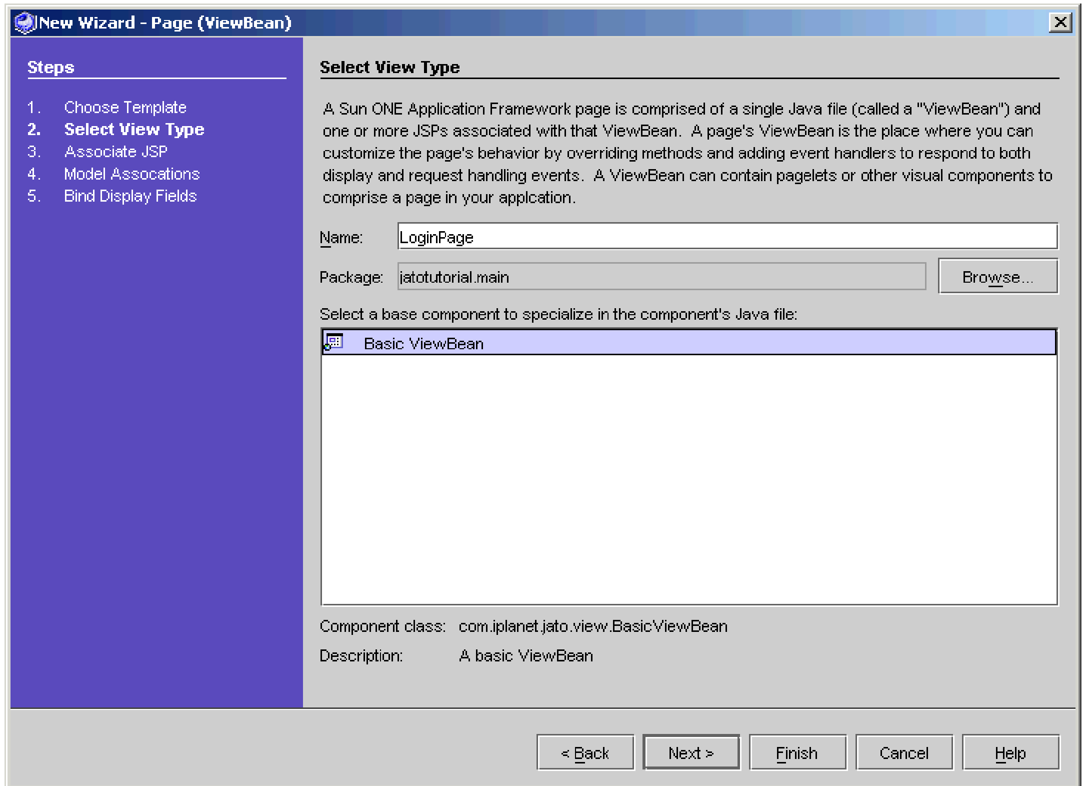
   a. **Right-click the main module folder.**

   b. **Select Add.**

   c. **Select Page (ViewBean).**

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

   The View Location panel displays.

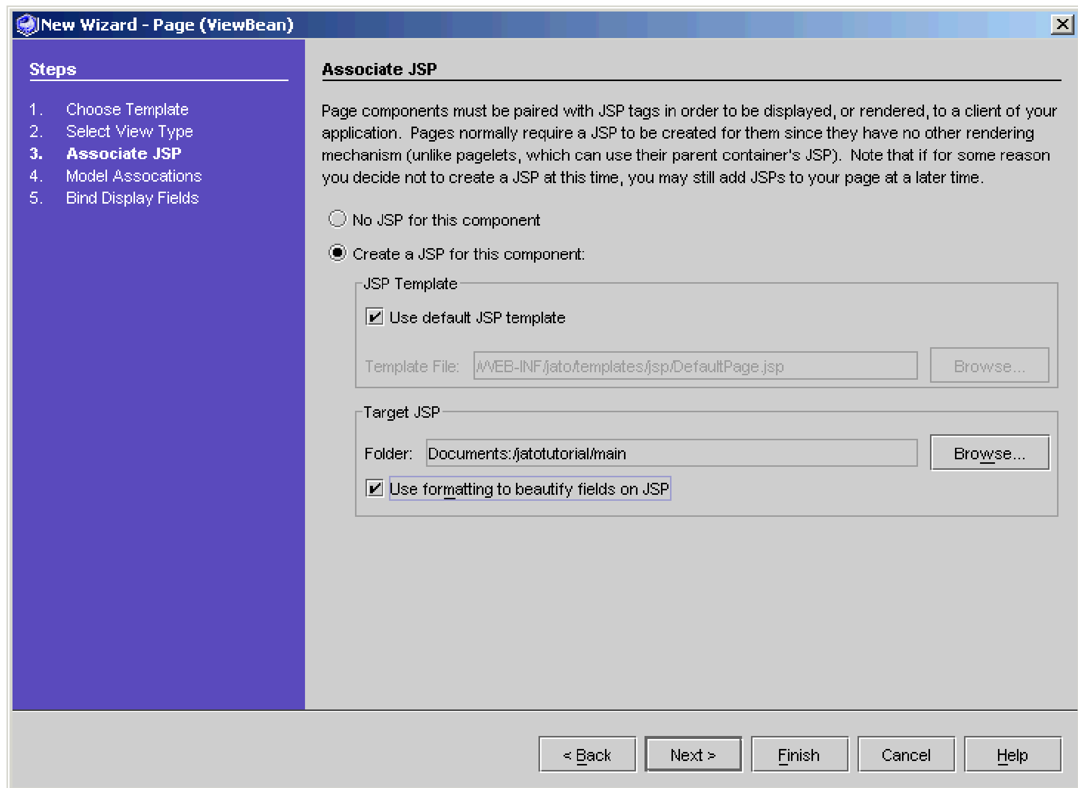


6. **Enter LoginPage in the Name field (to replace <*default*>).**

7. **In the View beans tab, select *Basic ViewBean*.**

8. **Click Next.**

   The Associate JSP panel displays.



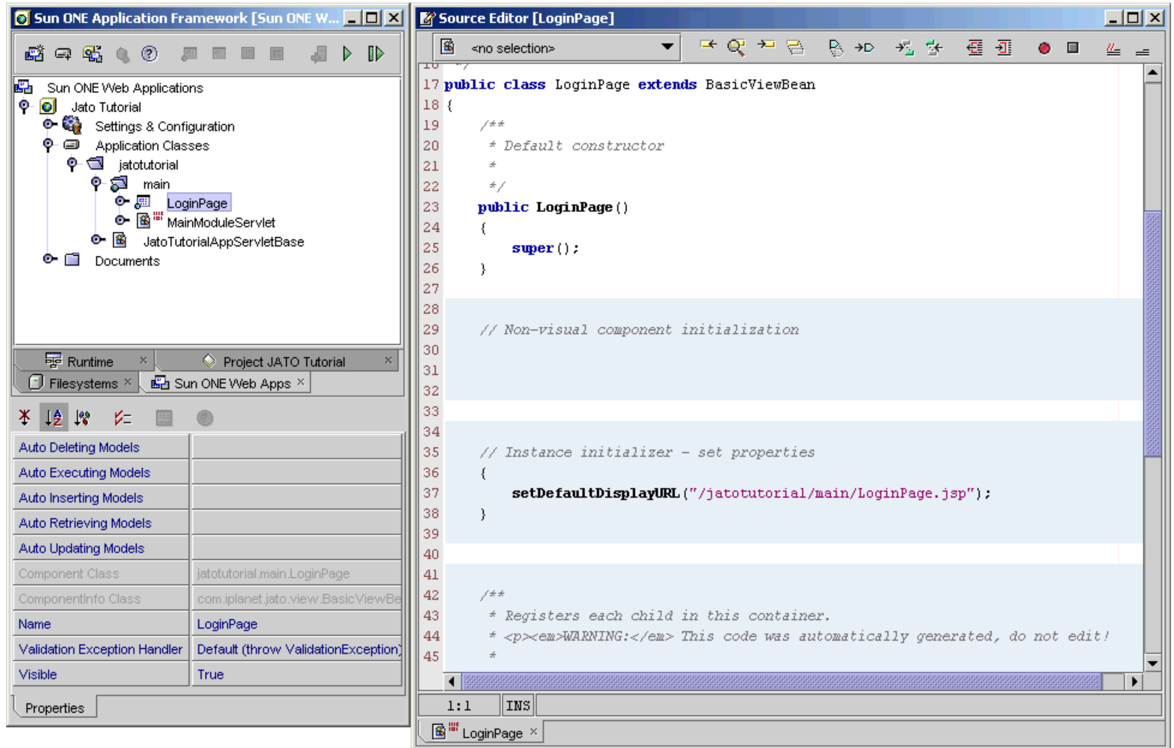
Accept all defaults.

9. **Check the *Use formatting to beautify fields on JSP* option.**

10. **Click Finish.**

    The ViewBean is created.

Note – There are additional steps in the Page wizard. However, those steps involve model field binding which is not required for the LoginPage. In a later task, you will use these additional steps.



11. **Double-click `LoginPage`.**

    The generated source code displays in the Sun ONE Studio editor.

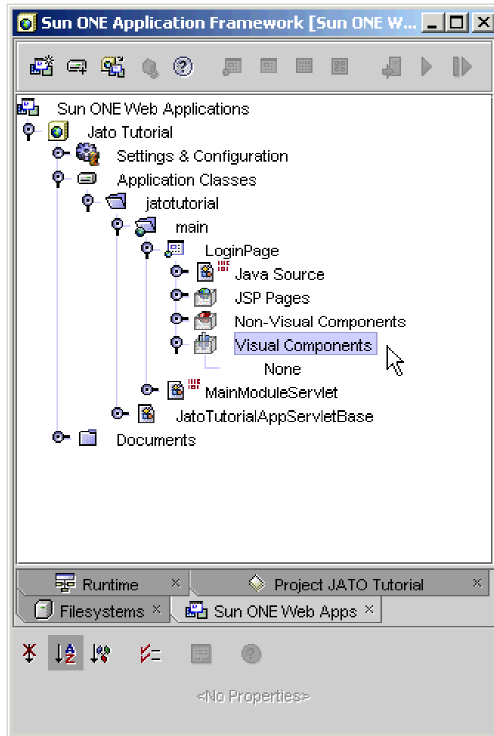
Note – Because you elected to created a JSP when you created the LoginPage, a JSP was added to the Documents folder in a directory structure that mirrors the ViewBean's package structure (`/jatotutorial/main`).

For convenience, a link to the JSPs that use the LoginPage are placed in the node of the JSP, which is under the `LoginPage` node.

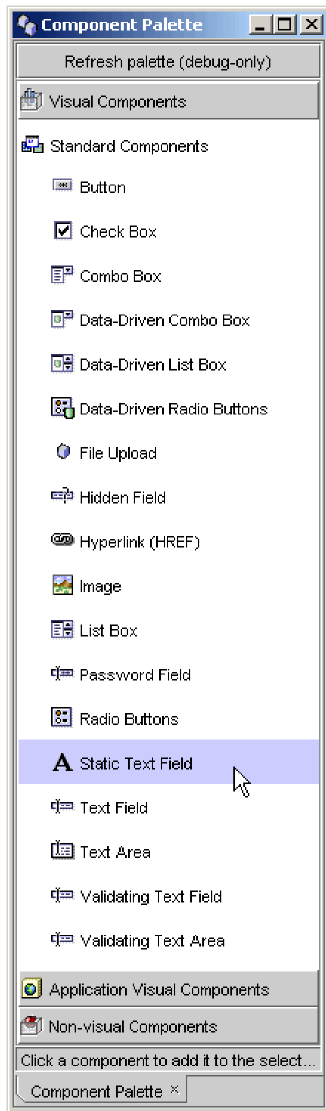
Add Display Fields to the Login Page

12. **Expand the `LoginPage` node.**

13. **Select the Visual Components node under the LoginPage node.**

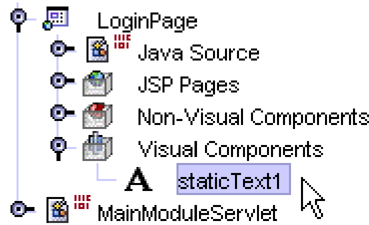


14. **In the Sun ONE Application Framework Component Palette, click the Static Text Field option.**



A static text visual component is added to the Visual Components node.

The default name is *staticText1*.



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

16. **Select Rename.**

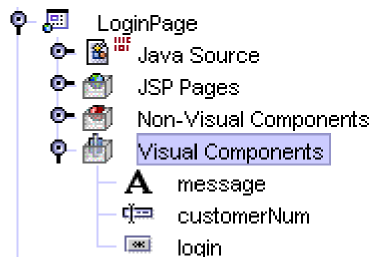
17. **Rename the field to *message*.**

18. **Add two more display fields.**

    The following table contains a list of the two visual component 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

    The three display fields display under the Visual Components node of the LoginPage.



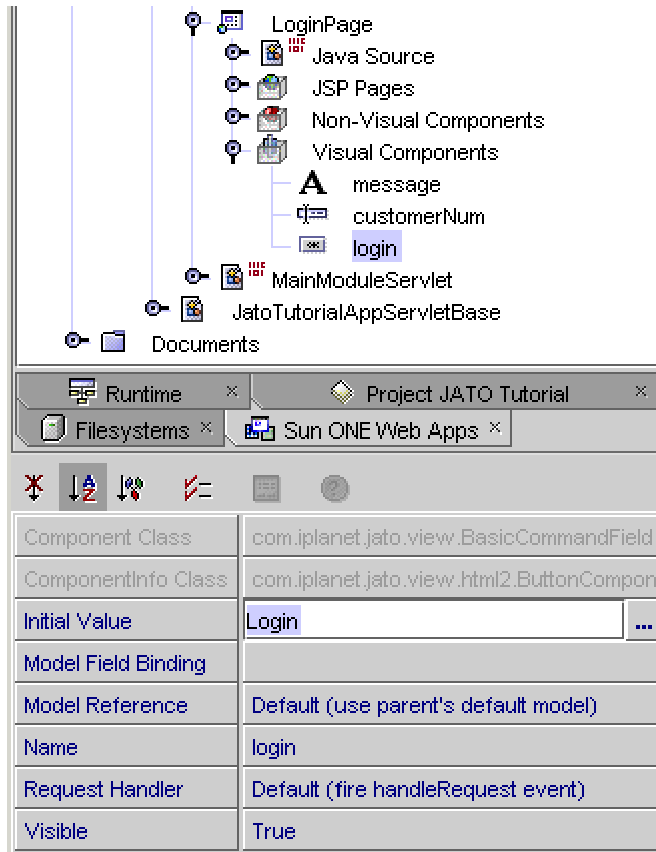
    Adding display fields to the Page also adds the appropriate JSP tags for the display fields to the JSPs that are using this Page.

19. **Set the button's Initial Value property by selecting *login*.**

20. **Click in the Initial Value property value entry area.**

21. **Enter the string *Login*.**

    The button's value is the string that displays on the button in the browser.



22. **Open the LoginPage's JSP to see the tags for the three display fields.**

    a. **Expand the JSP Pages node under the LoginPage node.**

    b. **Double-click the LoginPage JSP to open it in the Sun ONE Studio editor.**

23. **Format your JSP layout however you want.**

Note – Because you checked the option in the page wizard to *beautify* the JSP page contents, some basic formatting was applied to get you started. However, you will probably want to modify things a bit more.

For example, adjust the `customerNum` label so that it is proper case, and remove the unnecessary label for the button and the static text message field.

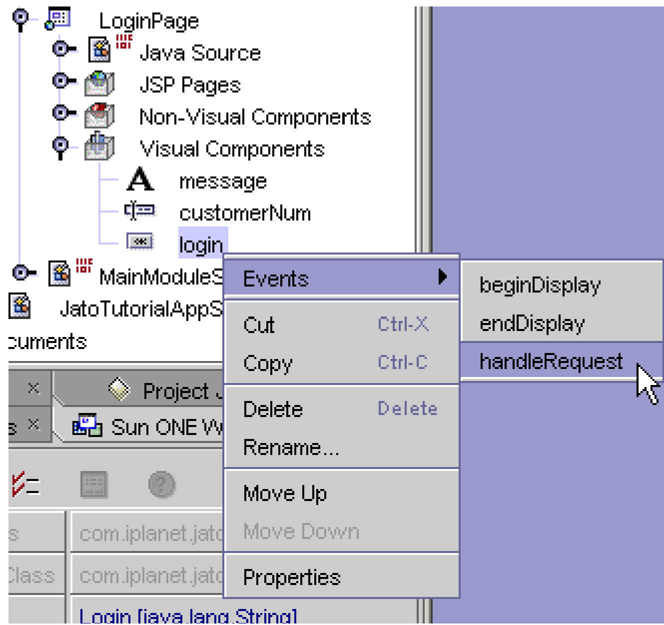
You can edit it directly in the Sun ONE Studio editor, or you can use your favorite WYSWIG HTML editor.

Here is an example of some minimal JSP changes (only pertinent code is shown here). Some HTML source code appears in **bold** type below for emphasis.

```
<jato:form name="LoginPage" method="post">
<table border=0 cellspacing=2 cellpadding=2 width="100%">
<tr>
  <td align=right valign=middle width="20%"></td>
  <td align=left valign=middle><jato:text name="message"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"><b>Customer Num:</b></td>
  <td align=left valign=middle><jato:textField name="customerNum"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"></td>
  <td align=left valign=middle><jato:button name="login"/></td>
</tr>
</table>
</jato:form>
```

Add Code to the Login Button

24. **Right-click the login button.**
25. **Select Events -> handleRequest**



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

26. Implement the login button handle request code.

Replace the following default code:

```
getParentViewBean().forwardTo(getRequestContext());
```

with the code shown in **bold** below:

```
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 output status message
    getDisplayField(CHILD_MESSAGE).setValue(theMessage);

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

CHAPTER 6

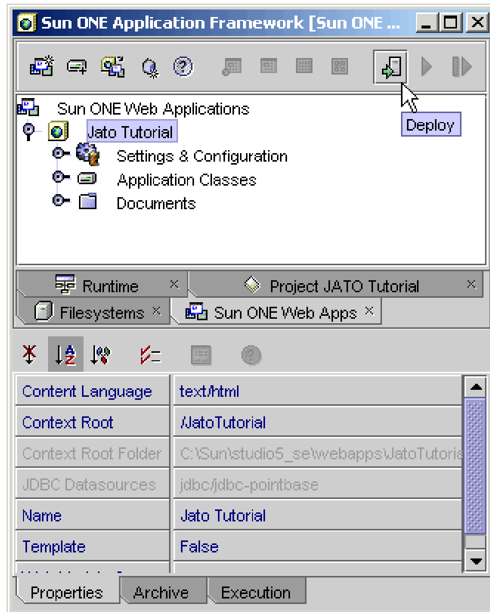
Tutorial—Section 1.3 Test Run the Login Page

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

Task 3: Test Run the Login Page

Compile the Web Application

1. **Select the Application Name folder.**



2. **Click the Deploy button on the Sun ONE Application Framework toolbar at the top of the Explorer window.**

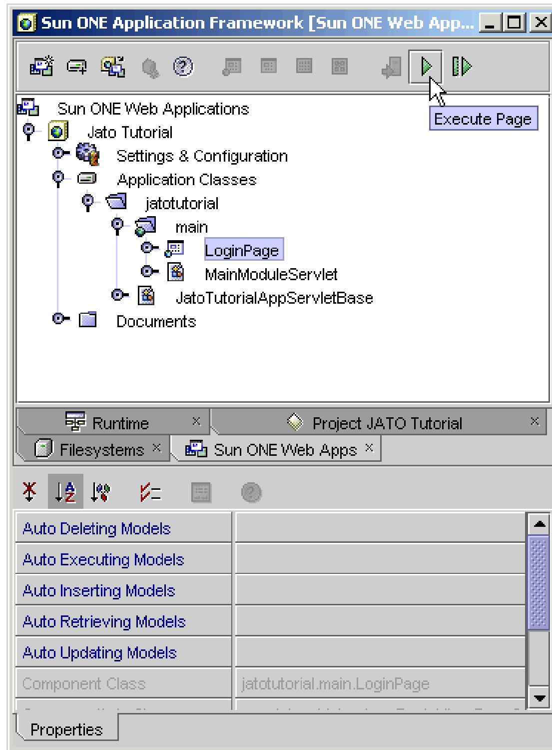
   This compiles the entire Web application (those classes that need to be compiled) and deploys it to the Sun™ ONE Application Server in one step.

   If you followed all of the tutorial instructions, the Web application compiles and deploys without error. See the Sun™ ONE Studio Output window for error messages.

   This deployment step is required for any change you make to any of the resources in your Web application when running it in the Sun™ ONE Application Server 7 (Application Server).

Test Run the Login Page

1. **Select LoginPage.**



2. **Click the Execute Page button located on the Sun ONE Application Framework toolbar at the top of the Explorer window.**

Note – The Execute Page (Redeploy) button (just to the right of the Execute button on the Sun ONE Application Framework toolbar) forces the Sun ONE Application Server to reload all resources (for example, JSPs, classes, and so on). It actually restarts the Sun ONE Application Server. This is necessary if the Sun ONE Application Server must be restarted to pick up the new changes so that it does not use objects in memory.

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

A default browser starts the application.

Test a Successful Login

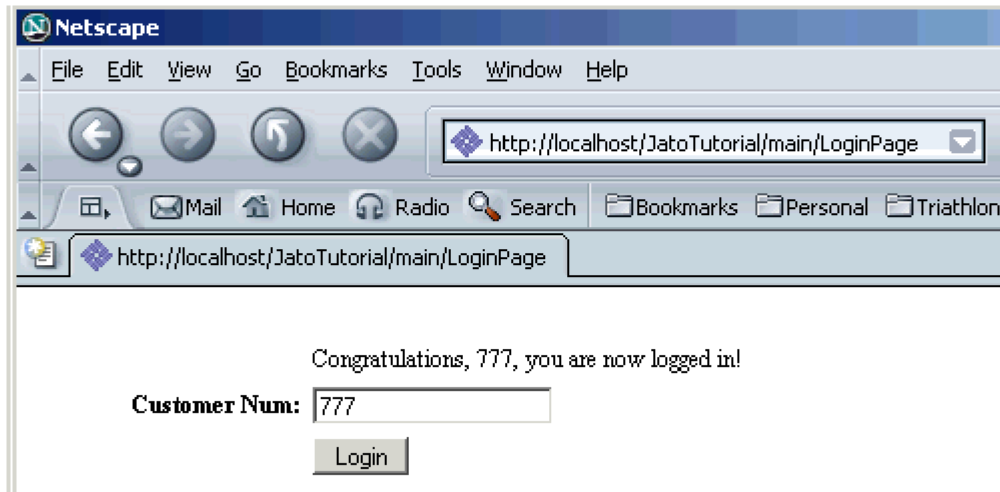
1. **Enter a valid login (for example, 1, 777, or 410 are valid (hard-coded) customer numbers).**
2. **Click Login.**

Caution – If you press the enter key while in the text field, the form is submitted for you. However, the server does not know which button to address from this submit action. 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.

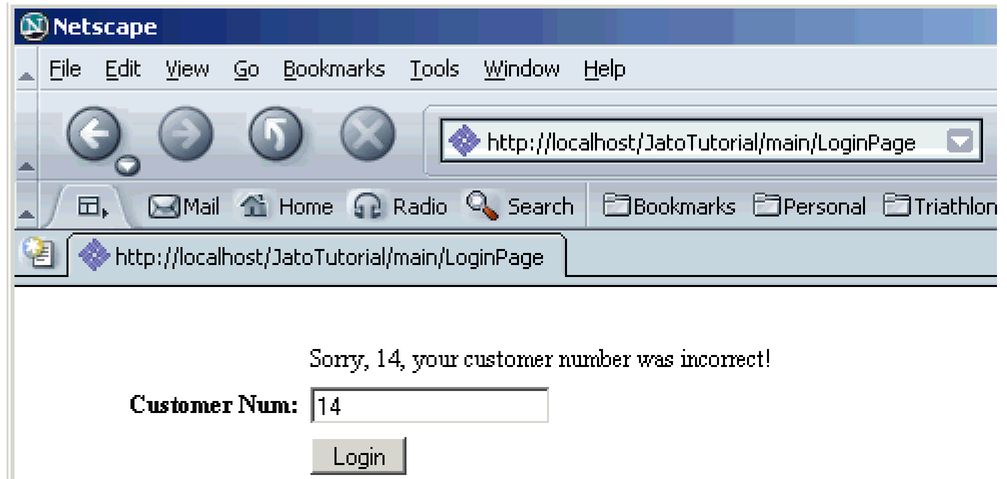
The login page should refresh displaying the success message.



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



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. **You need to add the PointBase driver file to the servlet container's classpath. The driver can be found in the installation directory of the Sun ONE Studio, as follows:**

   ```
   <studio-install-dir>/appserver7/pointbase/server/lib/pbserver42RE.jar
   ```

   The easiest way to accomplish this is to copy this driver to your application's `web-inf/lib` directory.

3. **Open a browser and run it with the URL appropriate to the servlet container.**

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

   Apache Tomcat or Caucho Resin servlet containers:
   `http://localhost:8080/JatoTutorial/main/LoginPage`

Note – You might 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 chapter describes how to expand the application and prepare the Sun™ ONE Application Framework application to access a SQL Database.

Expand the existing application by adding a SQL-based model and a page to display that model's data.

Link the two application pages together so they show coordinated data.

Task 1: Accessing a SQL Database

Connect to the Sample Database

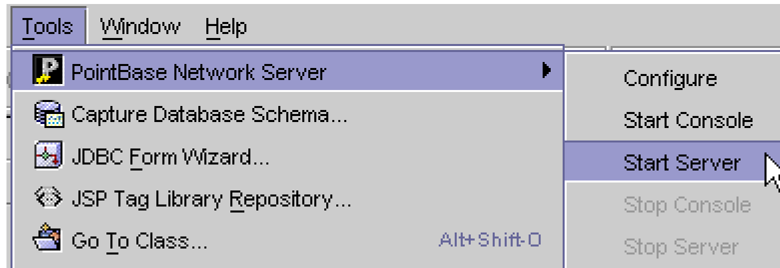
Note – The remainder of the tutorial assumes the presence of an RDBMS database which is used as a prerequisite for introducing you to some additional Sun ONE Application Framework features.

There is no requirement for an Sun ONE Application Framework application to access an RDBMS. Therefore, your actual applications might 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.

---

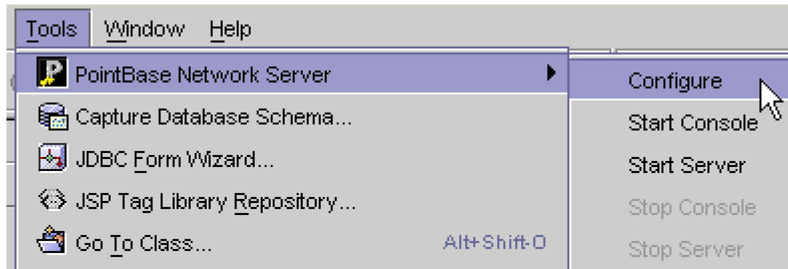
1. **Select the menu option Tools -> PointBase Network Server -> Start Server from the Sun ONE Studio to start the PointBase Network Server (database server).**



Caution – Depending upon how you installed the Sun ONE Studio and the Sun ONE Application Server, you might get an error message that prohibits you from starting the PointBase Network Server. If this happens, you just need to configure it first. If you were able to start the server, skip to the JDBC Datasources section.

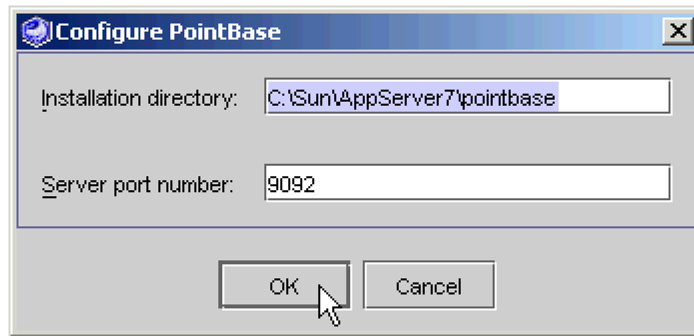
---

2. **Select Tools -> PointBase Network Server -> Configure.**



A dialog displays that prompts for a file storage location and a port number.

3. **Accept the defaults by clicking OK.**



You receive two dialogs warning that a file already exists, and asking if it is OK to overwrite them.

4. **Click OK to overwrite both of them.**

Your PointBase Network Server is ready to be started.

JDBC Datasources

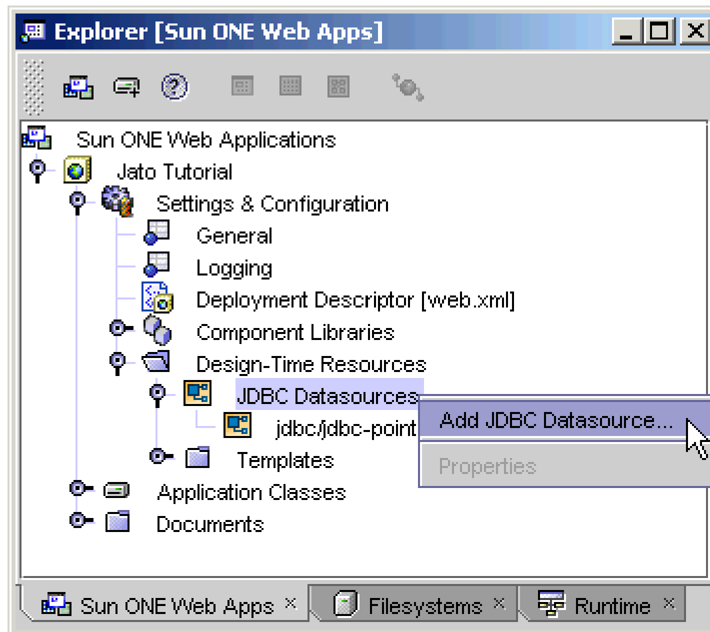
You can create a JDBC Datasource using the Sun ONE Application Framework JDBC Datasource wizard.

However, by default, one was created for you that points to the PointBase sample database that ships with the Sun ONE Studio.

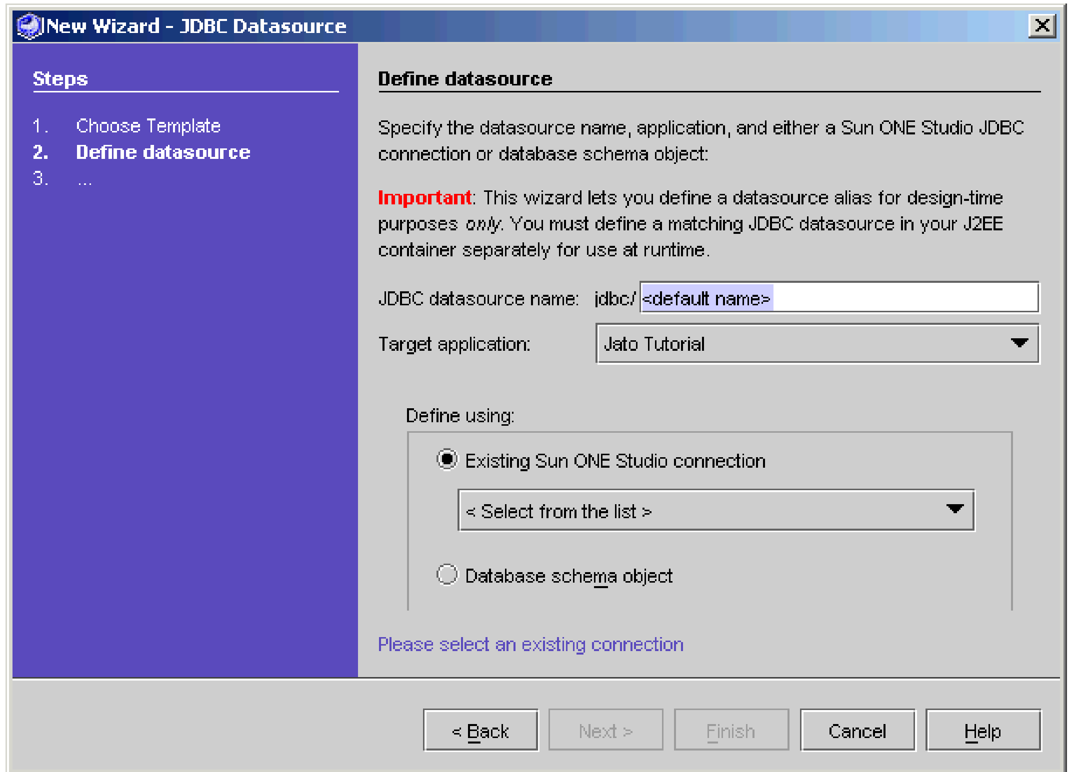
If you need to create additional JDBC Datasources for a different database other than the one used in the tutorial, use the following steps.

(Otherwise, read over to become familiar with this topic, or skip to the Tomcat (and other non-JNDI containers) SQL Connection Preparation section.)

1. **Under the Sun ONE Application Framework Web application node (Jato Tutorial), expand the Settings & Configuration folder.**
2. **Expand the Design-Time Resources folder.**
3. **Right-click the JDBC Datasources node.**
4. **Select Add JDBC Dataource.**



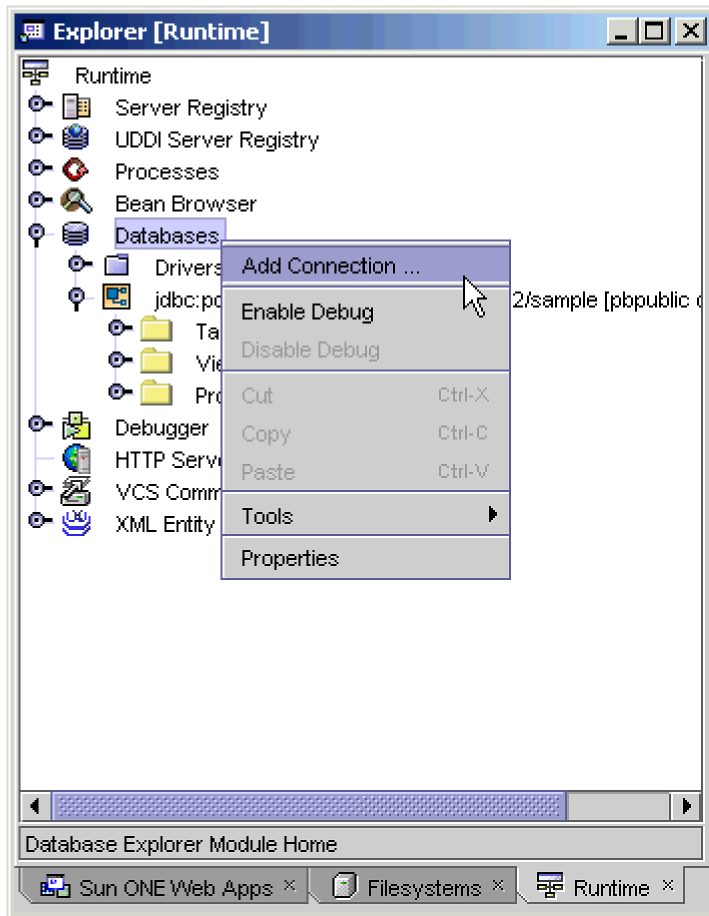
The Define JDBC datasource panel displays.



5. **Enter the preferred datasource name in the New datasource name textbox.**

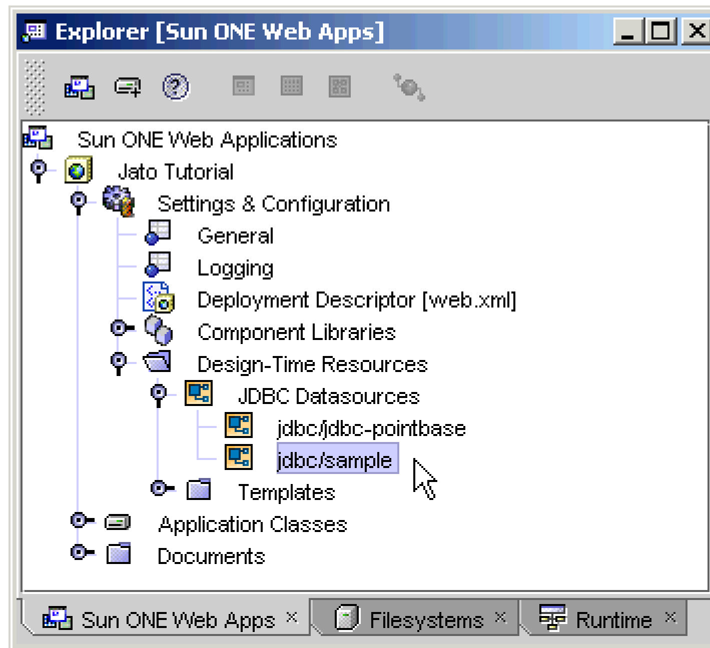
6. **In the Select connection combo box, select the appropriate JDBC connection.**

   If the connection you need does not exist, you must create one. This is performed by a tool that is outside the scope of the Sun ONE Application Framework tools. You need to select the Runtime tab in the Explorer window, expand the Databases node, right-click it, and select Add Connection. You might need to add a driver for your database before you can add a connection. See the Sun ONE Studio online help for more details.



7. **Click Finish.**

   A new JDBC Datasource node is created.



Note – JDBC Datasources are only needed at design-time when creating JDBC SQL Models (tables and stored procedures). The JDBC SQL Model wizard presents a selection of the datasources that have been created.

The JDBC Datasources are not involved in the runtime environment. You must configure your runtime container with the proper JNDI settings, unless you are using direct JDBC URLs to connect to databases.

Tomcat (and other non-JNDI containers) SQL Connection Preparation

---

Note – If you are using the Sun ONE Application Server to run your tutorial application, you can skip this step, because JNDI is supported.

If you are using the built-in Tomcat engine, or running the tutorial application in another servlet container that does not support JNDI, you need to make a few minor modifications to the application servlet base class (JatoTutorialAppServletBase) in your application.

---

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

   There is a lot of commented-out event code in here with comments describing what you can do with the events. Ignore them as you do not need any of them for this tutorial application.

   You need to add a static initializer to perform the following:

   a. **Instruct the Sun ONE Application Framework not to use JNDI lookups**

   b. **Load the PointBase JDBC driver**

   c. **Map the JDBC Datasource (jdbc/jdbc-pointbase) to the PointBase sample database's JDBC connection URL**

   The following code sample shows the code that needs to be added to the JatoTutorialAppServletBase class. Only the **bold** code needs to be added. Much of the code/comments from the JatoTutorialAppServletBase class has been omitted here.

```
public class JatoTutorialAppServletBase extends ApplicationServletBase
{

    static
    {
    // Turn off JNDI lookup (turn on DriverManager use)
    SQLConnectionManagerBase.setUsingJNDI(false);

    try
    {
        // load the PointBase JDBC driver
        Class.forName("com.pointbase.jdbc.jdbcUniversalDriver");
    }

    catch (ClassNotFoundException e)
    {
        // if the driver is unavailable, an exception will be thrown
        e.printStackTrace();
    }

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

    } // static init

}
```

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

Important: If you value performance in your Web application, use JNDI for production.

- If you are not going to be testing the tutorial in the Sun ONE Application Server, you need to copy the PointBase client library JAR file (pbserver42RE.jar) into your WEB-INF/lib directory.

  You can get the PointBase client library from the following directory:

  <studio-install-dir>/appserver7/pointbase/server/lib

- If you are using a different database, you might need to place that database vendor's client library in your WEB-INF/lib directory or in your servlet container's lib/ext directory (somewhere in the classpath). Place it in the Web application’s WEB-INF/lib directory.
- This is not necessary for the Sun ONE Application Server because the PointBase libraries are already included in its classpath.

CHAPTER 8

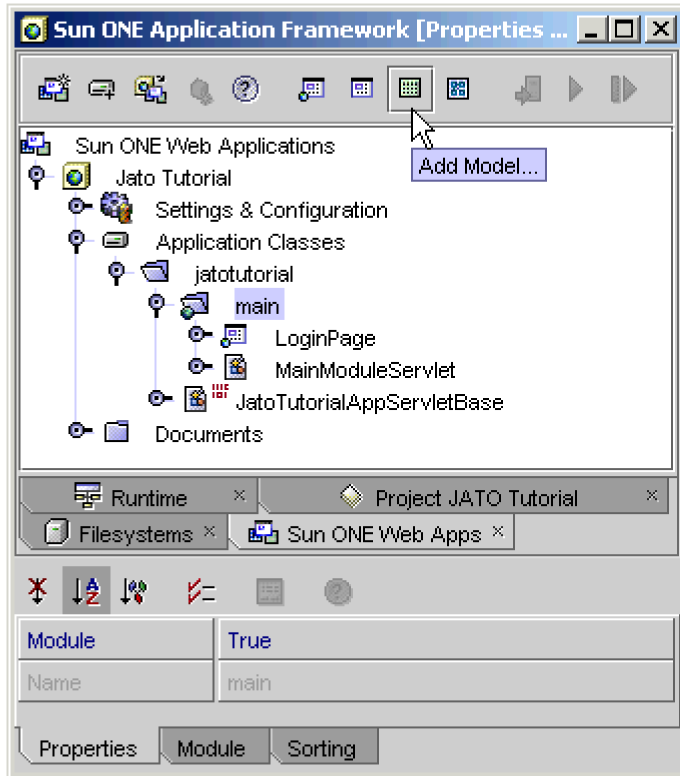
Tutorial—Section 2.2 Create the CustomerModel

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

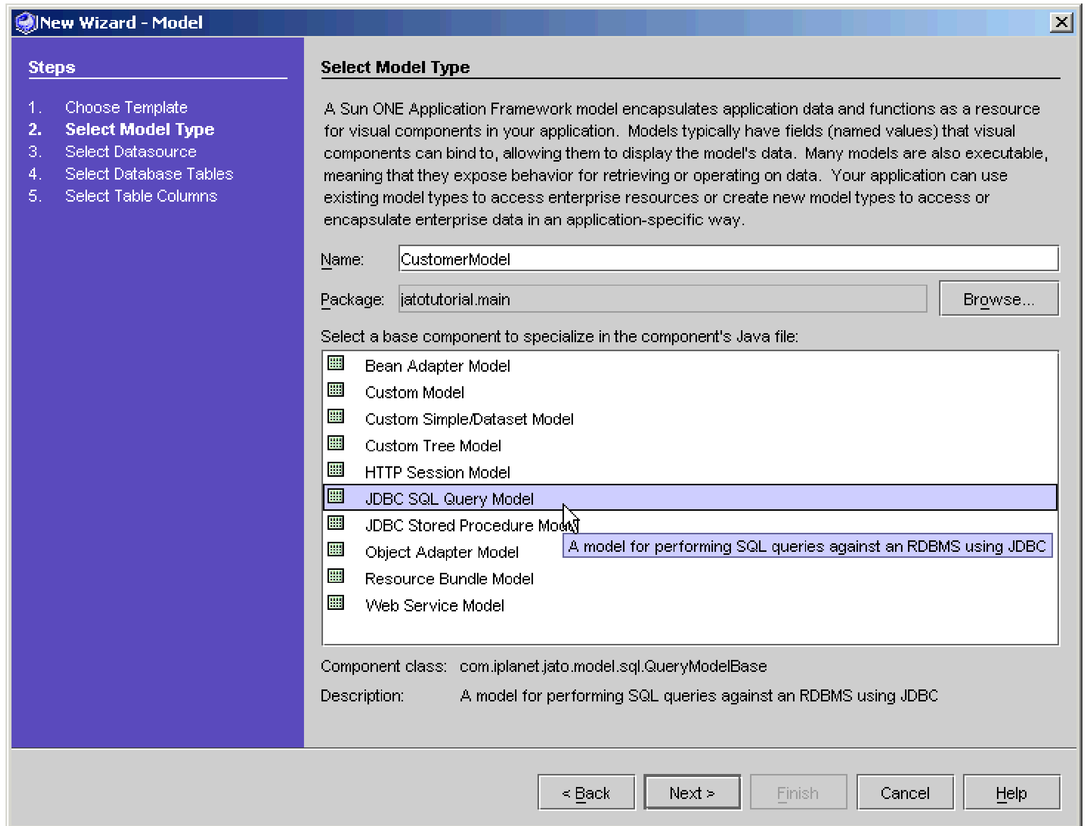
Task 2: Create the CustomerMode

Create a JDBC™ SQL Model

1. **Select the main module folder.**
2. **Click the Add Model button on the Sun ONE Application Framework toolbar.**



The Select Model Type panel displays.

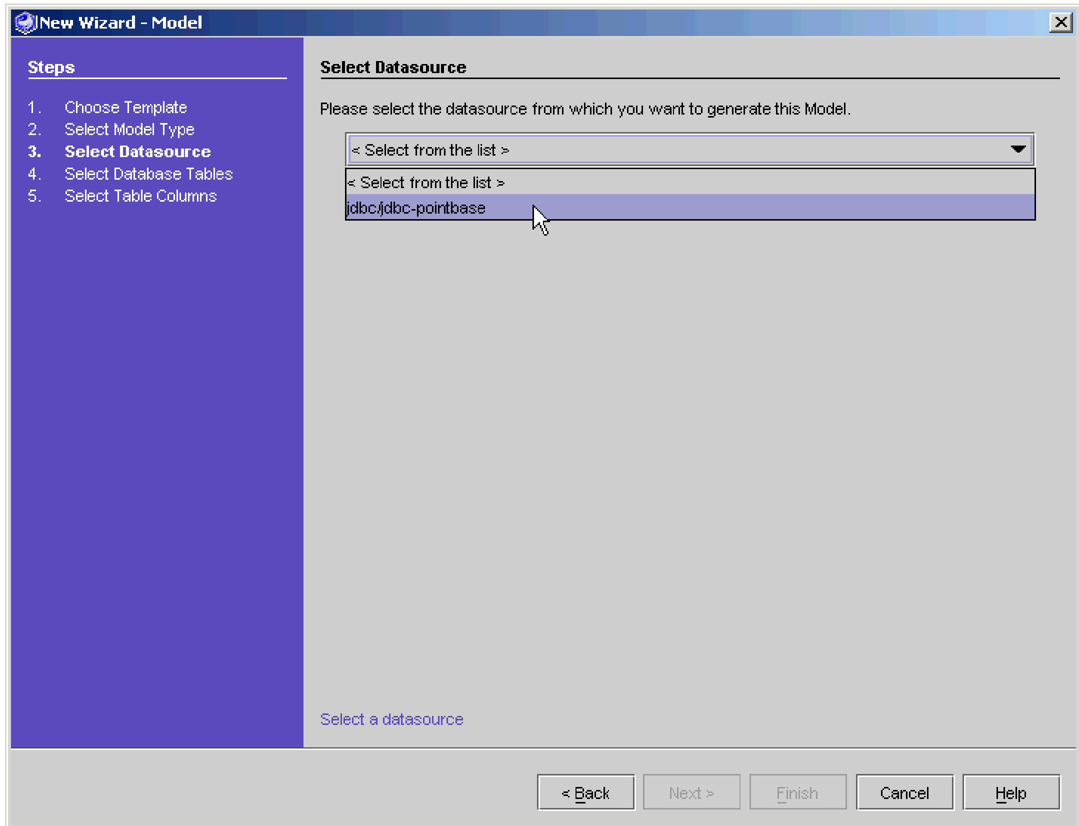


3. **Enter *CustomerModel* in the Name field.**
4. **Select JDBC SQL Query Model from the model component list.**

   The list you see might vary depending on the Sun ONE Application Framework version and the possible addition of custom or third party component libraries.

5. **Click Next.**

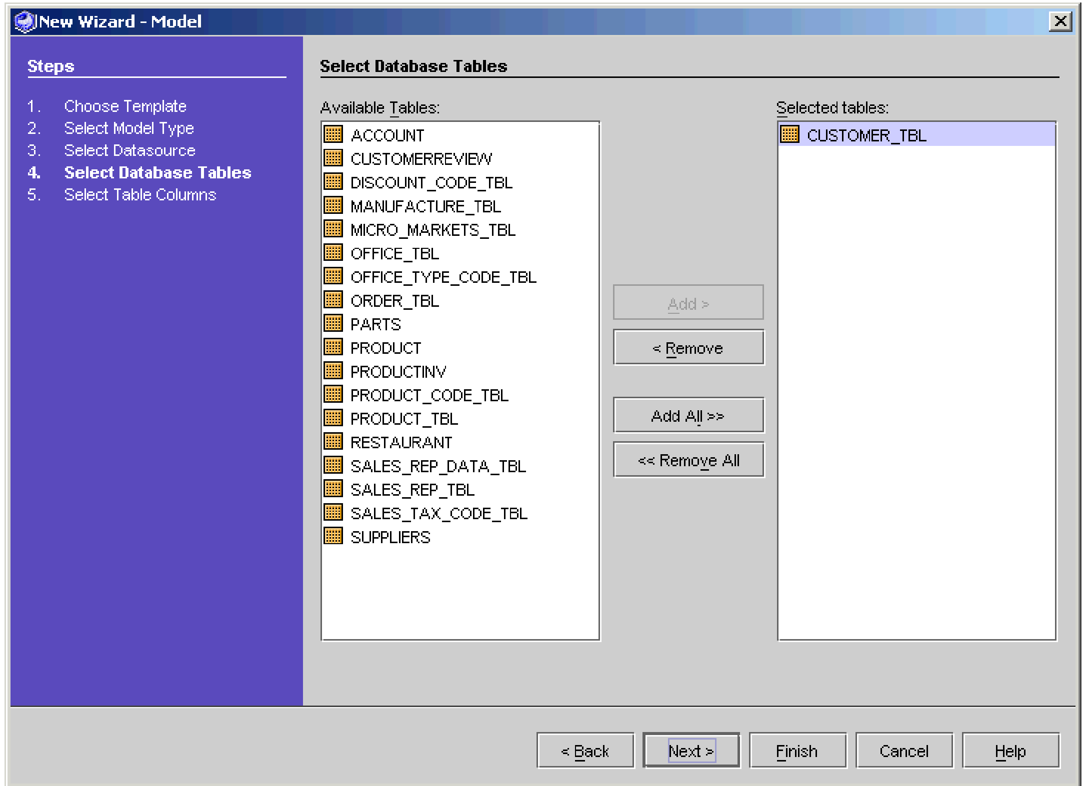
   The Select Datasource page displays.



6. **Select *jdbc/jdbc-pointbase* from the combo box.**

7. **Click Next.**

   The Select Database Tables page displays.

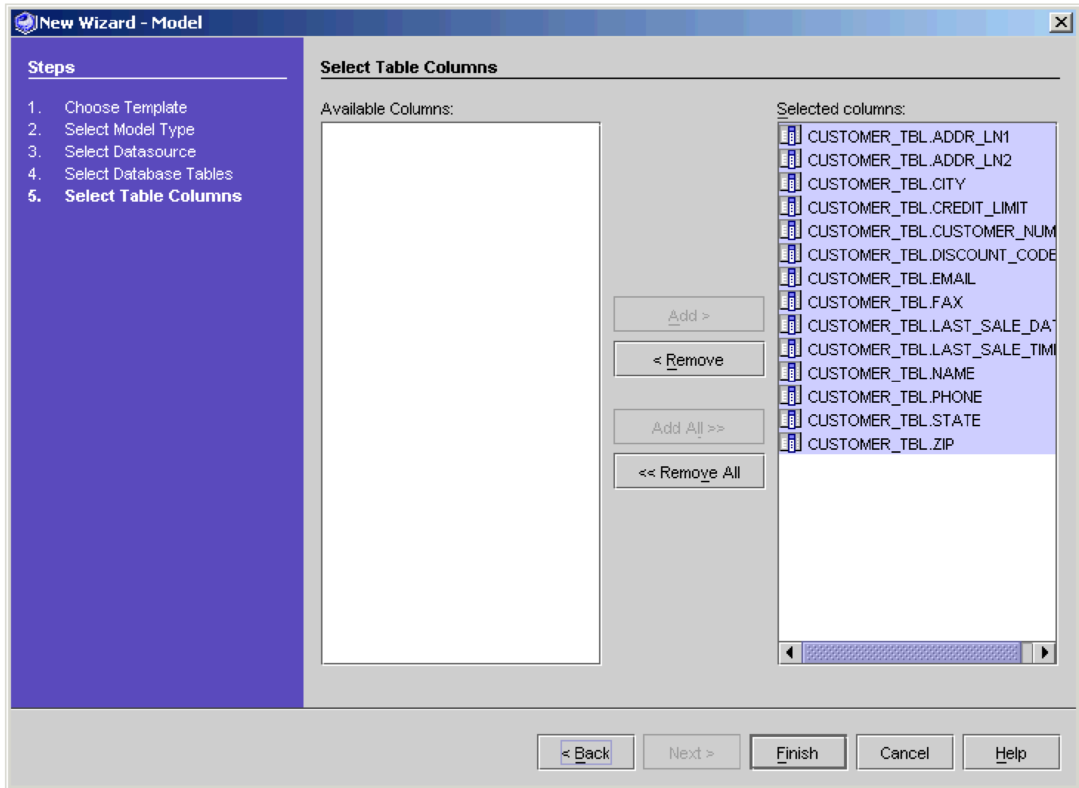


8. **Select** CUSTOMER_TBL**.**

9. **Click Add.**

10. **Click Next.**

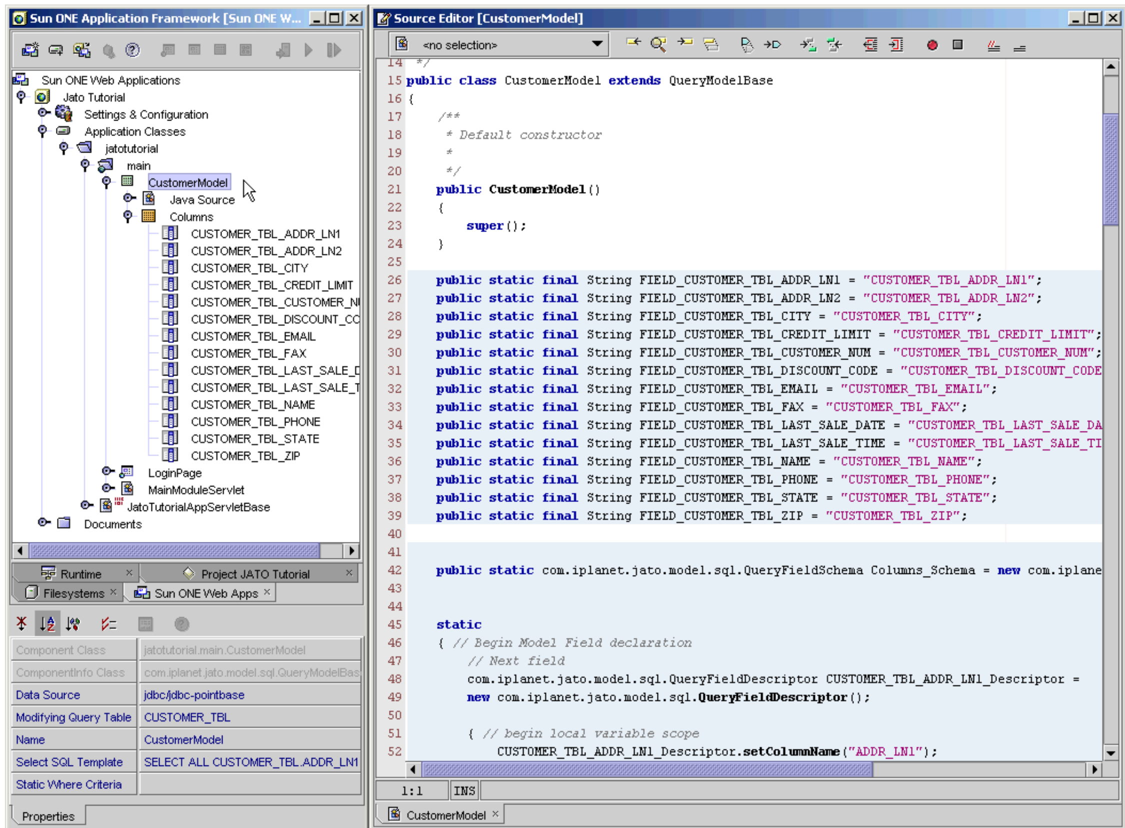
    The Select Table Columns page displays.



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

12. **Click Finish to create the Model.**

    The CustomerModel object is created in the main module.



13. **Expand the CustomerModel to see all of the columns.**

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

Mark the Model's Key Field(s)

Note – Due to a special type of key field indicator in the PointBase database schema metadata, the Model wizard does not properly detect the key field `CUSTOMER_TBL_CUSTOMER_NUM`. Therefore, you must set the key field manually.

This is not a problem if you create the datasource from a database schema object, and is also not a problem for non-PointBase databases, such as Oracle.

---

1. **Under the Columns node of the CustomerModel, select the** `CUSTOMER_TBL_CUSTOMER_NUM` **model field.**
2. **In the property sheet, select the Model Field Properties tab.**

   If the Properties tab is not visible, click the View -> Properties menu option, or right-click the key field column and select the Properties action.
3. **Change the value of the Key Field property from *false* to *true*.**

Column Name	CUSTOMER_NUM
Computed Field	False
Empty Formula	null
Empty Value Policy	Exclude
Field Type	java.lang.Integer
Insert Formula	null
Insert Value Source	Application
Key Field	False
Qualified Column Name	True
Supported Operations	False

Add Connection Code for Non-JNDI Enabled Containers

For servlet containers that do not support JNDI data sources, you can rely on explicit use of a JDBC driver.

Note – In section 2.1 of this tutorial (Task 1: Accessing a SQL Database), if you are testing your this application in a servlet container that does not support JNDI, you disabled the use of JNDI and declared the explicit use of the PointBase JDBC driver in the SQLConnectionManagerImpl class.

If you are testing in a servlet container that does not support JNDI, you must set the connection username and password explicitly in the model so that a proper database connection can be created before the model is executed.

Note – For production environments, you should use JNDI connections.

Add the **bold** code below to the CustomerModel's constructor.

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

While providing the datasource username and password as demonstrated above is not a good practice for a real world application, it is practical for this tutorial. Take extra care to obtain and provide the username and password in a more secure and robust implementation. When using the JNDI method, this code is unnecessary and this login information is provided by the configured JNDI connections in the application server.

CHAPTER 9

Tutorial—Section 2.3 Create Customer Page

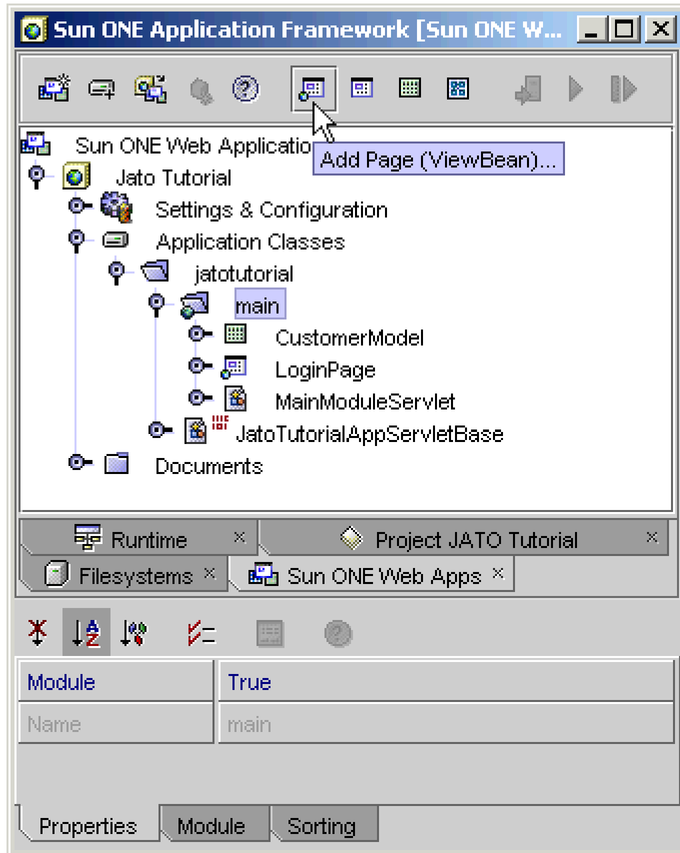
This chapter describes how to create a page in the Sun™ ONE Application Framework that displays data from a model that accesses a relational database.

Task 3: Create the Customer Page

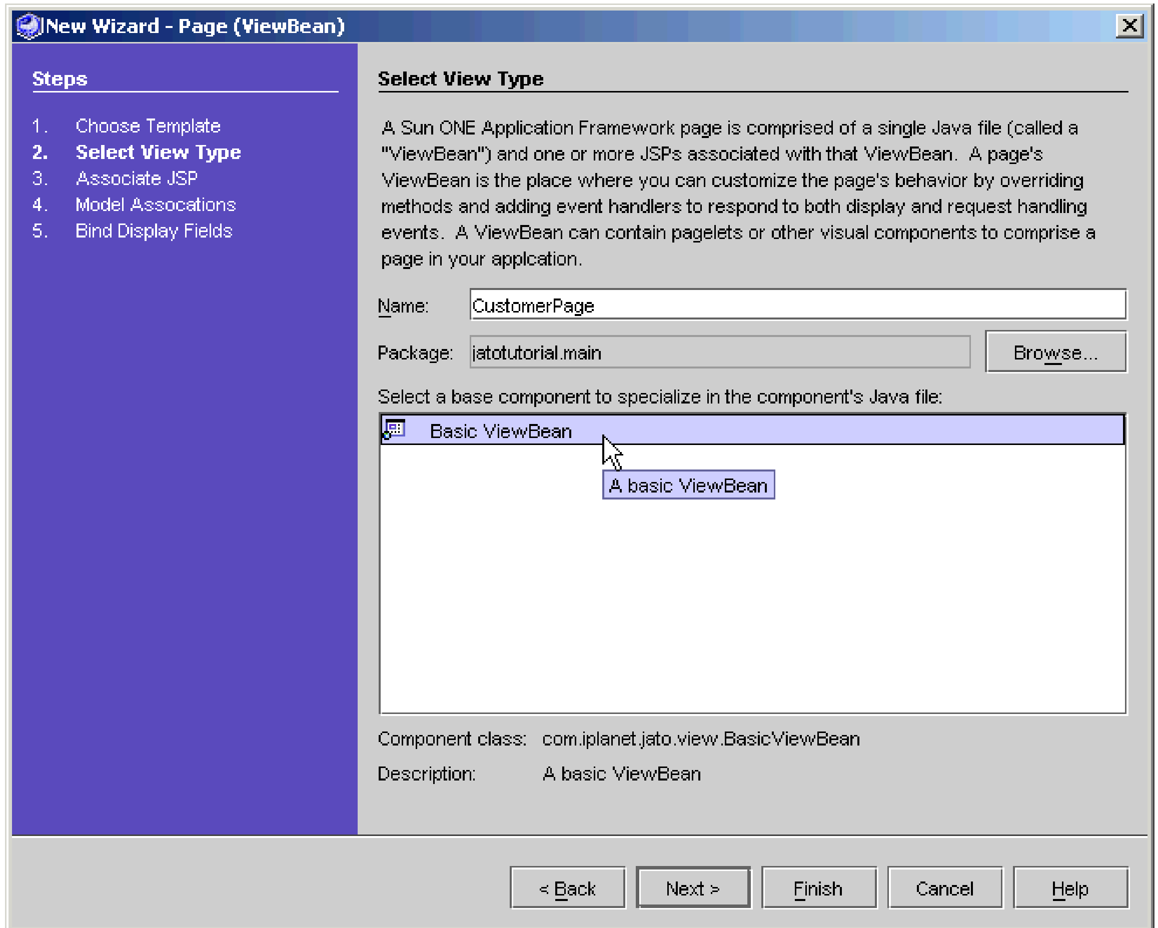
You will now create the second page of the application. However, this page will be bound to a model. This binding process automatically creates display fields on the page that display the data that is stored in the model's fields.

Add a ViewBean

1. **Select the main module.**
2. **Click the Add Page button on the Sun ONE Application Framework toolbar.**

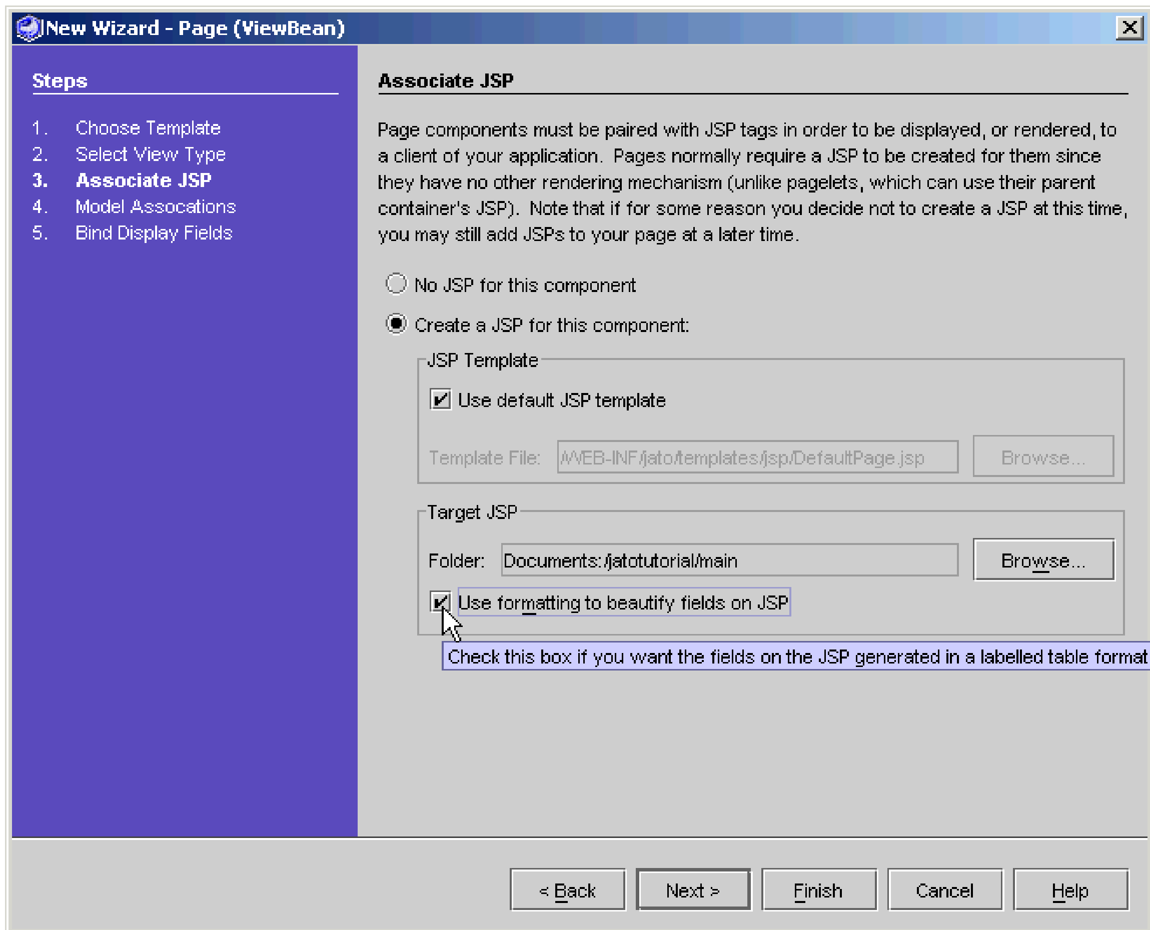


The Select View Type panel displays.



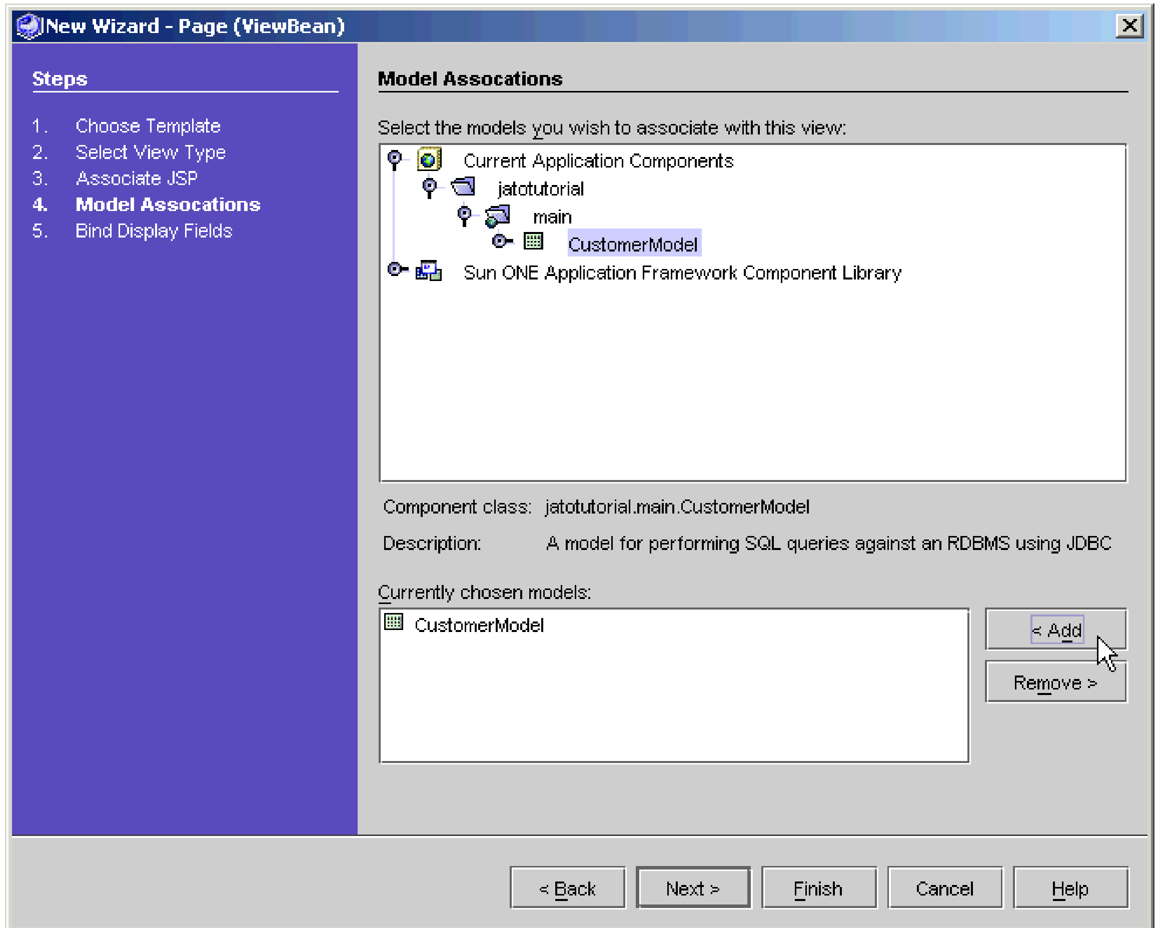
3. **Enter CustomerPage in the Name field (to replace <*default*>).**
4. **Select Basic ViewBean to create a ViewBean type Page component.**
5. **Click Next.**

   The Associate JSP panel displays.



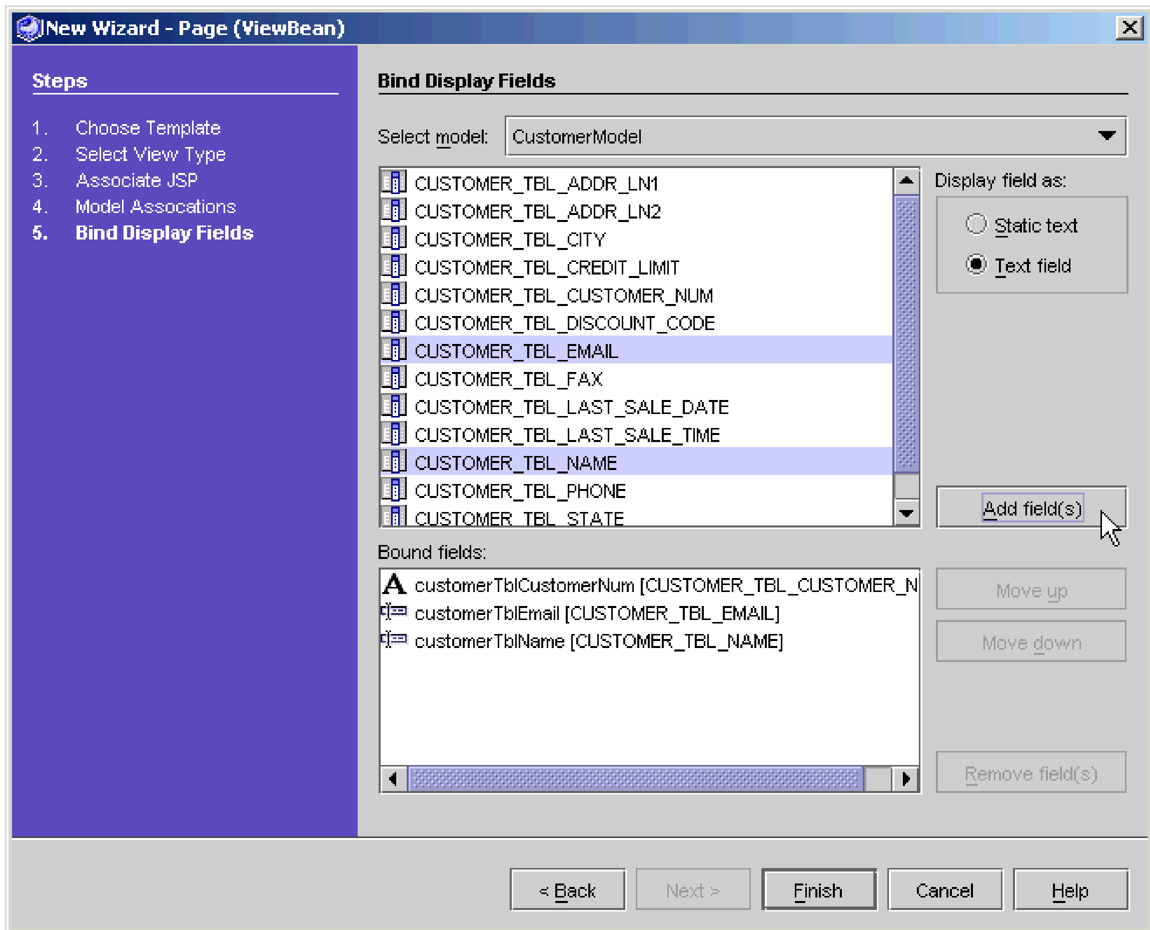
6. **Click the *Use formatting to beautify fields on JSP* check box to apply some basic formatting.**
7. **Click Next.**

   The Model Associations panel displays.



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

    The Bind Display Fields panel displays.



You only need to add three fields.

12. **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 Bound fields list box.

13. **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).**

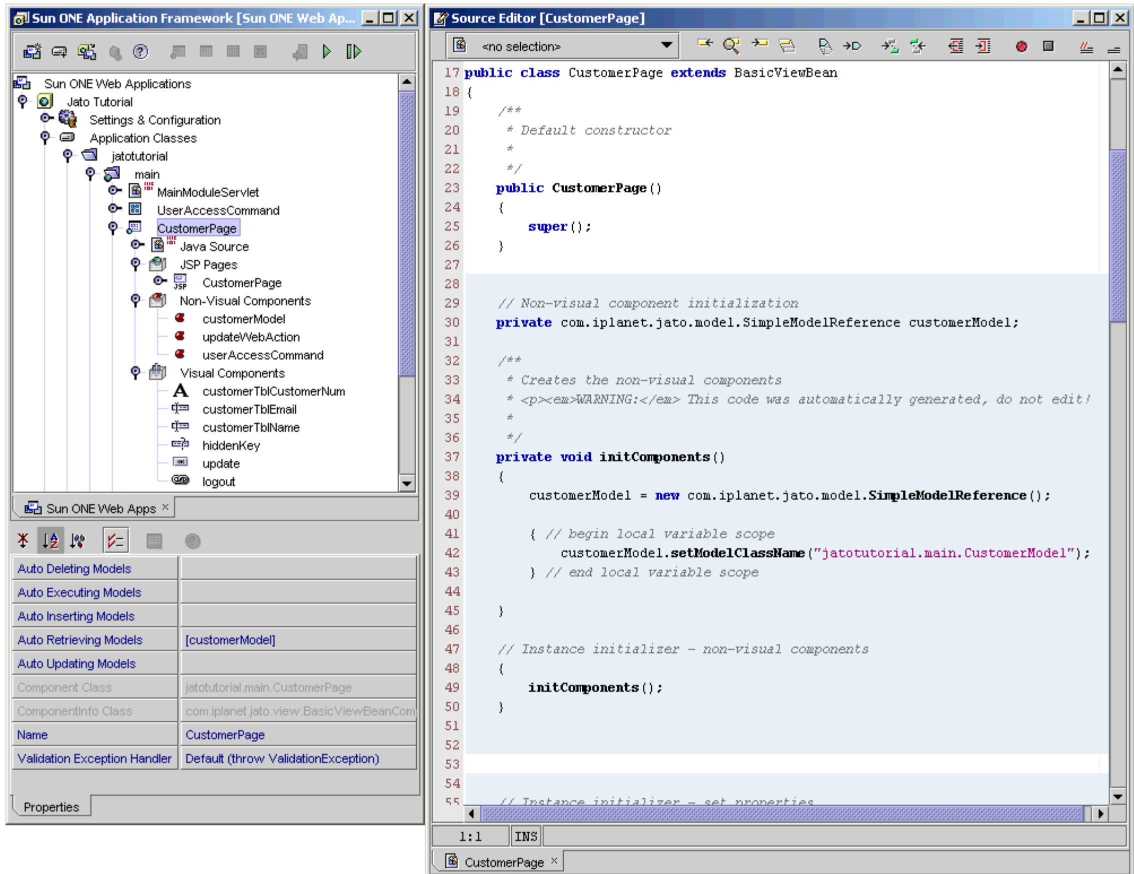
    b. **Select Text field.**

c. **Click Add field(s).**

The `CUSTOMER_TBL_EMAIL` and `CUSTOMER_TBL_NAME` fields are added to the Bound fields list box.

14. **Click Finish.**

You have created the ViewBean.



15. **Double-click CustomerPage.**

The code displays in the right-hand panel.

Expand all the subnodes of the CustomerPage to see the JSP Page, Visual Components, and Non-Visual Components that were automatically generated by the wizard.

Note – Like the LoginPage, a JSP for the CustomerPage 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 visual components 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 and changes to those fields to be automatically mapped back into the model, at which point you can execute the model to perform an update to the database. All of the SQL generation and connection creation are handled for you by the Sun ONE Application Framework.

If you really want or need to work with the JDBC API directly, the Application Framework does not require that you use all of the features it provides, and you are free to handle all of the JDBC responsibilities on your own. In other words, you can pick and choose what you need to use in the Sun ONE Application Framework, but it is most likely that the Sun ONE Application Framework's implementation is exactly what you need.

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

Add a Button Component

1. **Add a button to the CustomerPage.**

   The following table contains the specifications for adding a button to the CustomerPage. Use the Component Palette to add the button, just like you added the fields for the Login Page.

2. **If the Component Palette is not visible, select menu option:**

   View -> Sun ONE Application Framework -> Component Palette.

   The table shown below lists the specifications for adding a button to the CustomerPage. The left column shows the button type, the middle column shows the name, and the right column shows the initial value.

Type	Name	Initial Value
Button	update	Update

3. **Enable the button to update the customer record.**

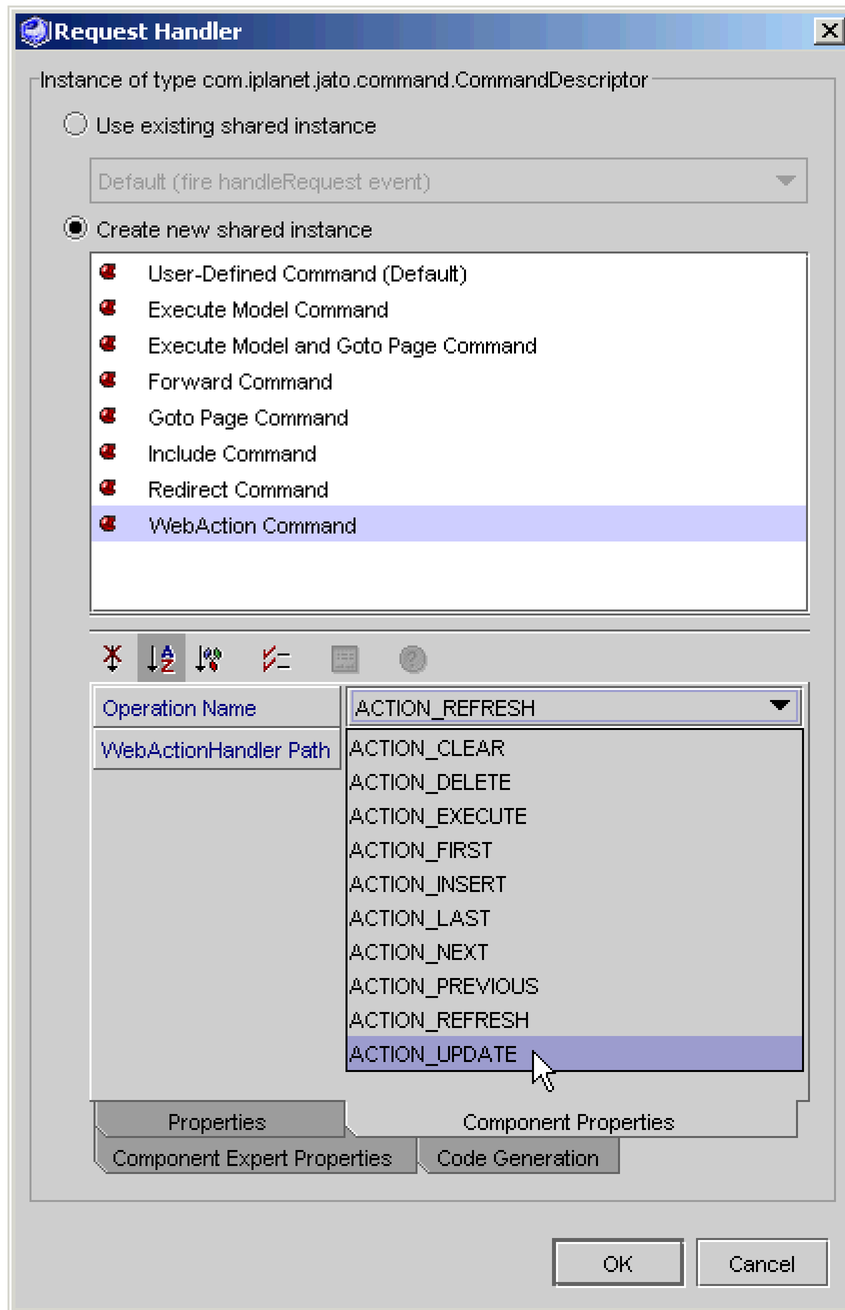
   a. **Select the update button field.**

b. **In the property sheet of the button, click the value area of the *RequestHandler* property.**

   The ellipsis button displays.

c. **Click the ellipsis button.**

   The Command Descriptor editor launches.



4. **Select Create new shared instance.**
5. **Select *WebAction Command* from the list.**

6. **In the Properties tab, change the name to *updateWebAction*.**

7. **Select the Component Properties tab at the bottom of the editor.**

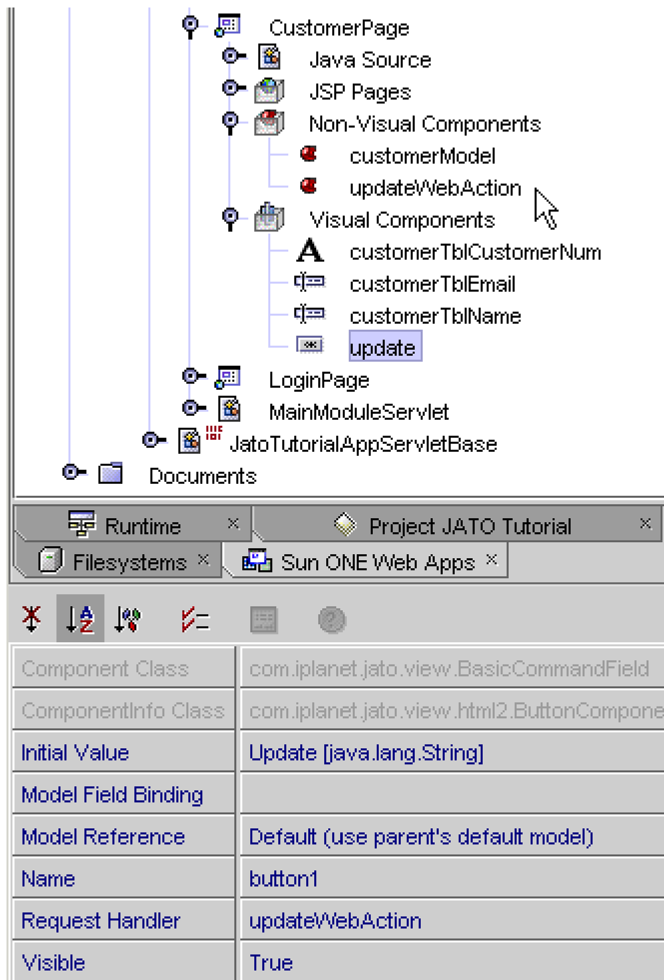
8. **Select ACTION_UPDATE for the Operation Name property.**

   Accept the defaults for the other two properties.

9. **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.



Making a Model Auto Update

You now need to add the customerModel reference as an Auto Updating Model on the CustomerPage.

You accomplish this by populating the Page's Auto Updating Models property with the appropriate model reference—in this case, the customerModel reference that was created for you by the wizard as a result of the model association or field binding you specified.

1. **Select the CustomerPage node.**
2. **Click the value area for the Auto Updating Models property.**

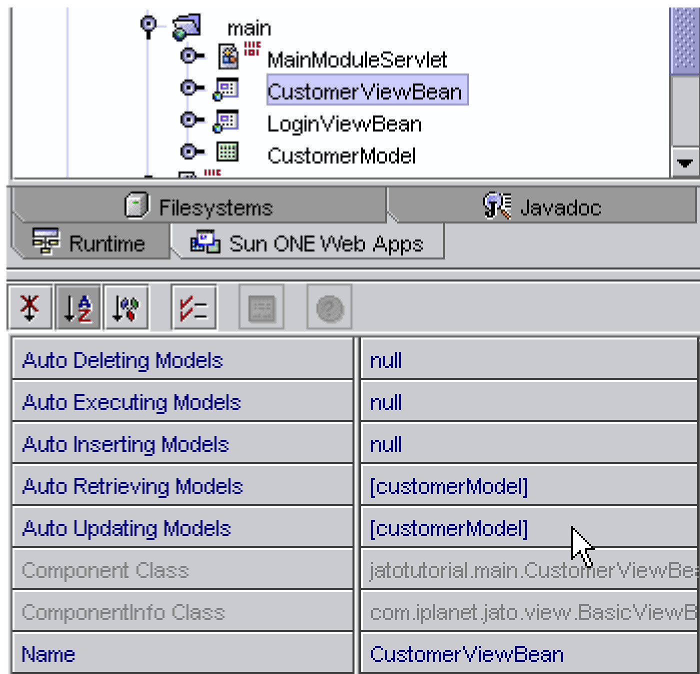
   The ellipsis button ("...") displays.
3. **Click the ellipsis button.**

   The Auto Updating Models custom editor launches.

   Note that the Properties area is blank when this editor first displays.
4. **Click New.**

   This adds an entry.
5. **Select customerModel from the Auto Updating Models combo box.**
6. **Click OK.**

   The property should now have the *[customerModel]* entry.



Note – The combination of the button's update Web action command descriptor and the auto retrieving or updating models configuration causes the CustomerModel to be executed when the Customer page is displayed or when the CustomerPage's Update button is clicked.

As an alternative to the declarative auto execution of this model, you can also write some code to perform the same purpose. Commonly, this code would be implemented in the Update button's handle*Update*Request event (similar to how the code was implemented for the Login button on the Login page).

Add a Hidden Field to the Customer Page

1. **Expand the CustomerPage node.**
2. **Expand the Visual Components node.**
3. **Select the Visual Components node.**
4. **Select add a Hidden Field component using the Component Palette.**

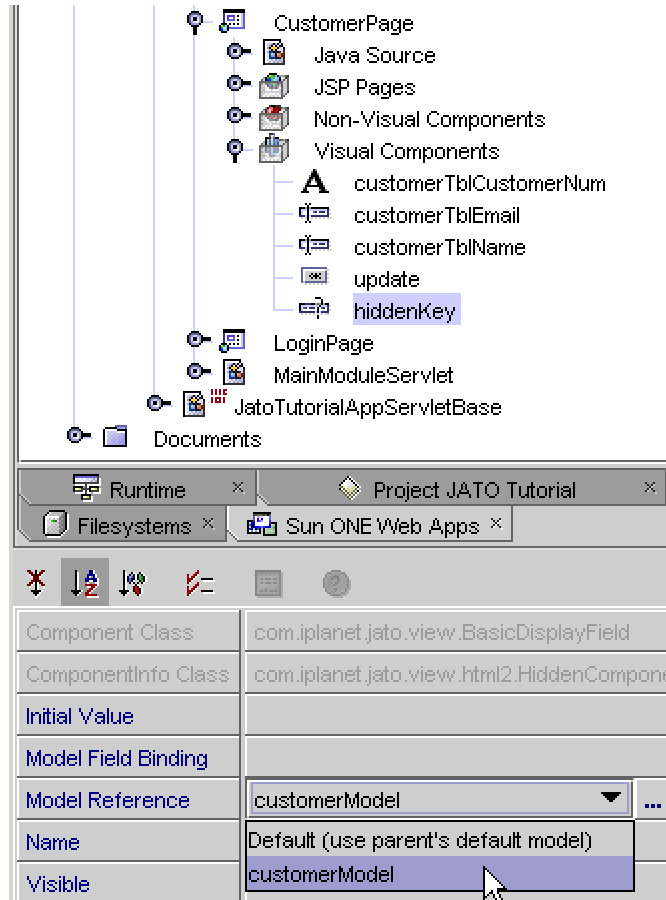
   A hidden field is added to the CustomerPage's Visual Components node.

5. **Rename the hidden field as *hiddenKey*.**

6. **Bind the hiddenKey field to the same model field that the customerTblCustomerNum static text field is bound.**

7. **Select the hiddenKey field.**

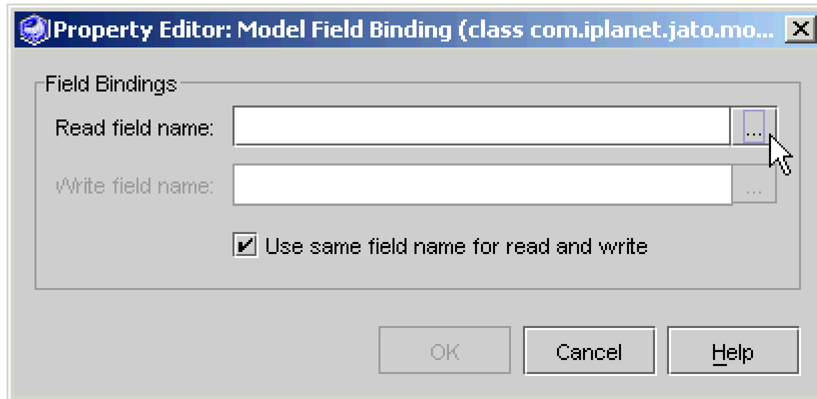
8. **On the property sheet, set the Model Reference property to customerModel by selecting from the drop down box.**



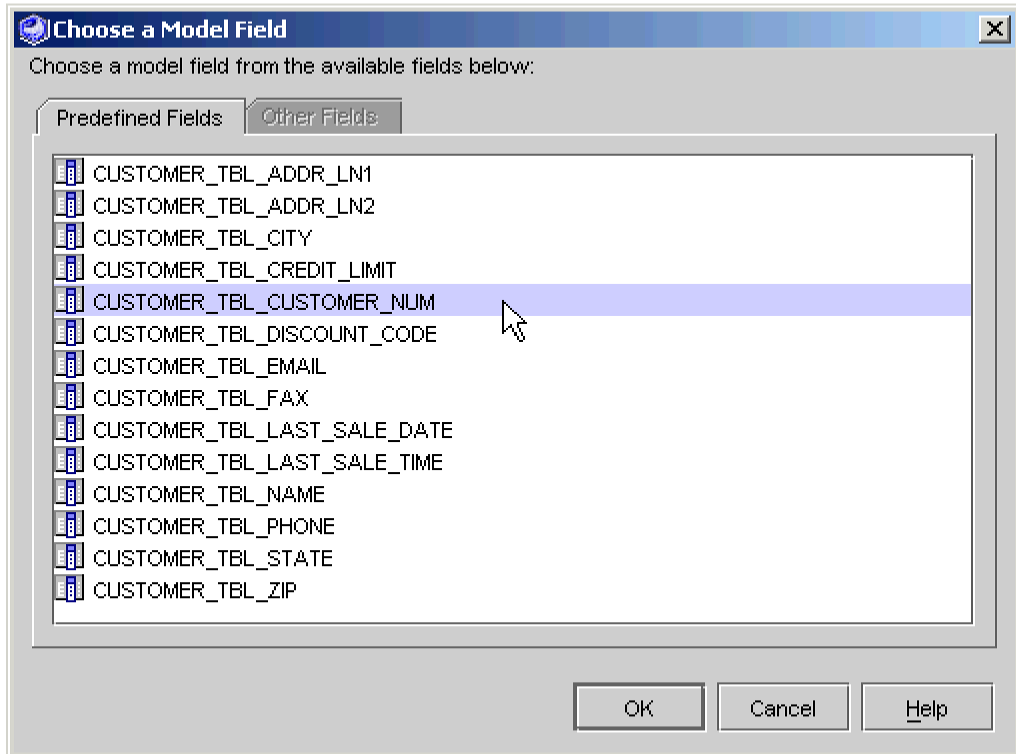
9. **Set the Model Field Binding property.**

   a. **Click in the value area of the Model Field Binding property.**

   b. **Click the ellipsis button to launch the Model Field Binding property editor.**



10. **Click the ellipsis button of the *Read field name* property to launch the Model Field Chooser editor.**



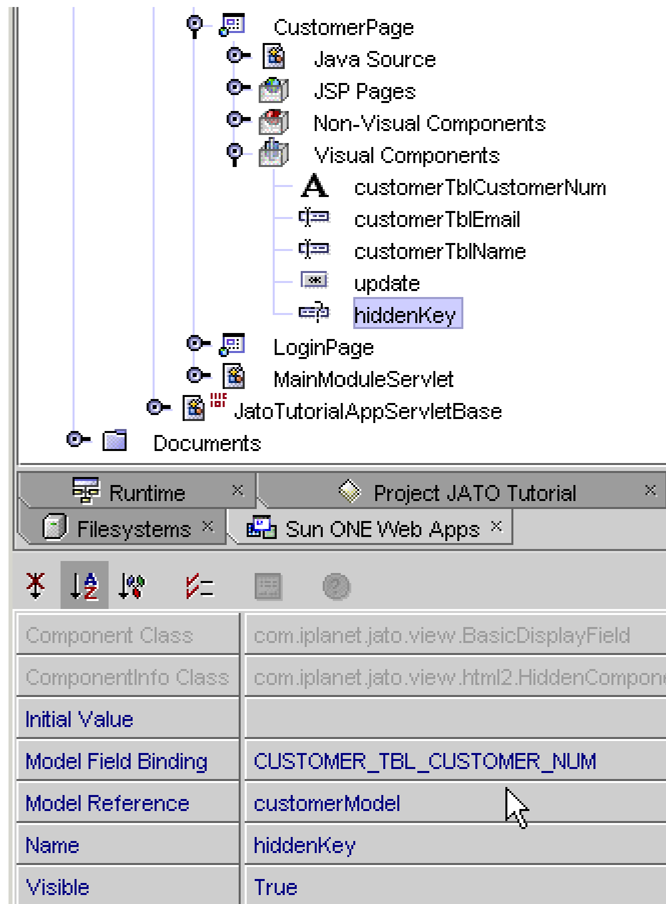
11. **Select *CUSTOMER_TBL_CUSTOMER_NUM*.**

12. Click OK.

The read and write fields are populated with the *CUSTOMER_TBL_CUSTOMER_NUM* model field.

13. Click OK.

This completes setting the Model Field Binding property for the *hiddenKey* display field.



Because the static text field is not an HTML input field, it's value will not be submitted back to the server when the update button is clicked. And because the customer number is the key field in the database table, the update logic needs this key value 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 rather than updating every record in the table.

To achieve this, you will preserve the customer number field value in a hidden field, which will be posted back on form submit and mapped back into the *CUSTOMER_TBL_CUSTOMER_NUM* model field.

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.

This is not exactly the way you would implement this in the real world. For security reasons, you would not want to expose the key field in the HTML as an input field that hackers could modify.

Note – This is not an issue specific to the Sun ONE Application Framework, but rather one that must be addressed by any Web application, no matter which framework (or no framework) is used to implement Web applications

The Sun ONE Application Framework provides a value add feature called Page Session that provides a technique to implement this solution more securely, but is outside the scope of this tutorial. Refer to the JatoSample application and the *Sun ONE Application Framework Developer's Guide* for more details.

Format the JSP

1. **Expand the JSPs node under CustomerPage node.**
2. **Double-click the CustomerPage JSP to open it in the editor window.**
3. **Provide propercase labels for the fields.**

   Following is an example of minimal JSP formatting (only pertinent code is shown here). Some of the HTML source code is shown in **bold** for clarity.

```
<jato:form name="CustomerPage" method="post">

<table border=0 cellspacing=2 cellpadding=2 width="100%">
<tr>
  <td align=right valign=middle width="20%"><b>Customer #:</b></td>
  <td align=left valign=middle><jato:text name="customerTblCustomerNum"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"><b>Email:</b></td>
  <td align=left valign=middle><jato:textField name="customerTblEmail"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"><b>Name:</b></td>
  <td align=left valign=middle><jato:textField name="customerTblName"/></td>
</tr>
</table>

<jato:button name="update"/>
<jato:hidden name="hiddenKey"/>
</jato:form>
```

CHAPTER **10**

Tutorial—Section 2.4
Test Run the Customer Page

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

Task 4: Test Run the Customer Page

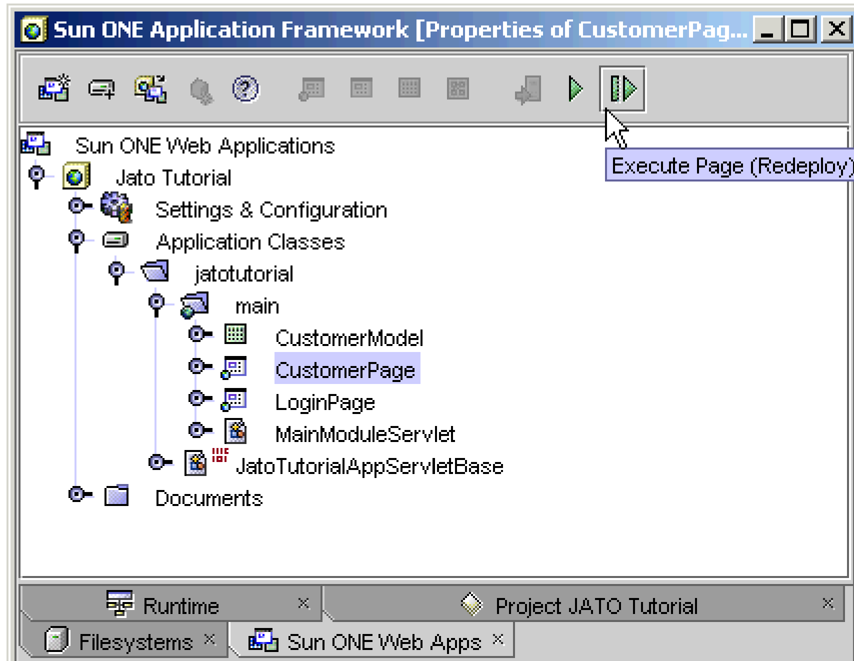
Important: Make sure the PointBase Network Server is running. If it is not, you can start it by doing the following:

1. **Select menu option Tools -> PointBase Network Server -> Start Server.**
2. **Right-click the Application Classes node.**
3. **Select the Compile All action.**

   If you are running on Sun™ ONE Application Server, you must Deploy the application when changes are made.

4. **Select the Sun ONE Application Framework application node (JatoTutorial), and click the Deploy button on the Sun ONE Application Framework toolbar.**
5. **Select the CustomerPage node, and click the Execute Page (Redeploy) button.**

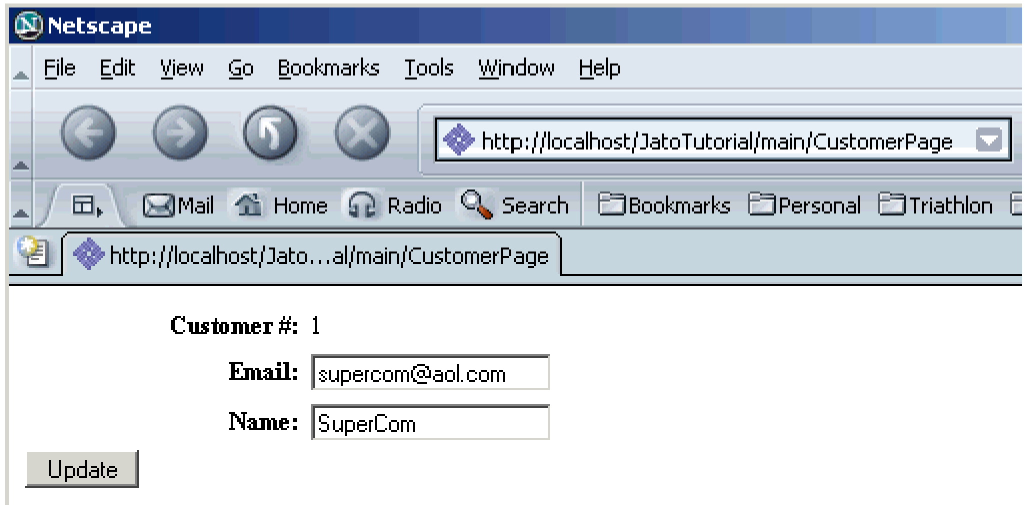
   Using this execute and redeploy option restarts the server to ensure the server picks up all changes and does not use any cached resources.



A default browser starts the application.

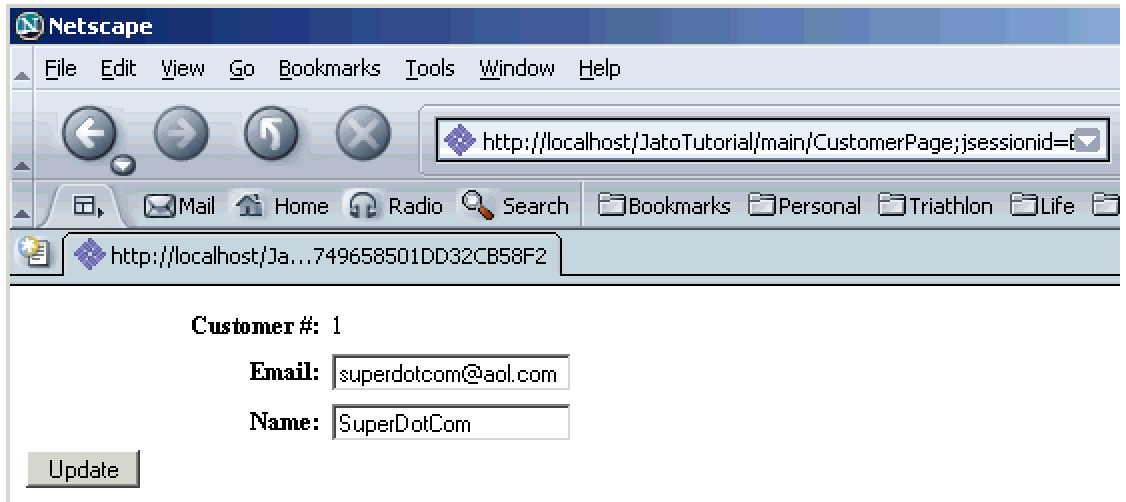
Test a Customer Update

1. **Make a change to one or both of the fields.**



In this figure, the email name and customer name were changed.

2. **Click Update.**



CHAPTER **11**

Tutorial—Section 2.5 Link Login Page to Customer Page

This chapter describes how to link the LoginPage to the CustomerPage in the Sun™ ONE Application Framework application, filtering the data the Customer page displays based on the customer's login.

Task 5: Link the Login Page to the Customer Page

Edit the handleLoginRequest Method in LoginPage

Edit the `LoginPage.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 from the customer table.

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 will likely lead to errors. It is best to just delete the current code and replace with the following.

Following is the code you need to enter to modify the logic in the `handleLoginRequest()` method.

```
public void handleLoginRequest(RequestInvocationEvent event)
{
    // 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

        // 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(CustomerPage.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 chapter 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

Important: Make sure the PointBase Network Server is running.

If it is not running, you can start it in the Sun™ ONE Studio as follows:

1. **Select menu option Tools -> PointBase Network Server -> Start Server.**

   Since you have made modifications to a few classes, be sure to compile the application.

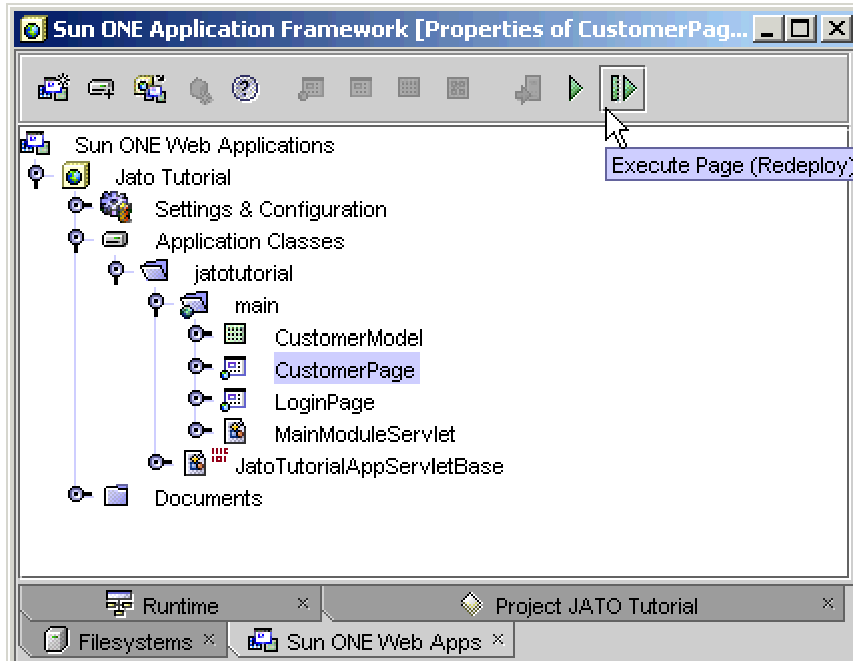
2. **Right-click the Application Classes node, and select the Compile All action.**

   If you are running on Sun™ ONE Application Server, you must Deploy the application when changes are made.

3. **Select the Sun ONE Application Framework Application node (JatoTutorial), and click the Deploy button on the Sun ONE Application Framework toolbar.**

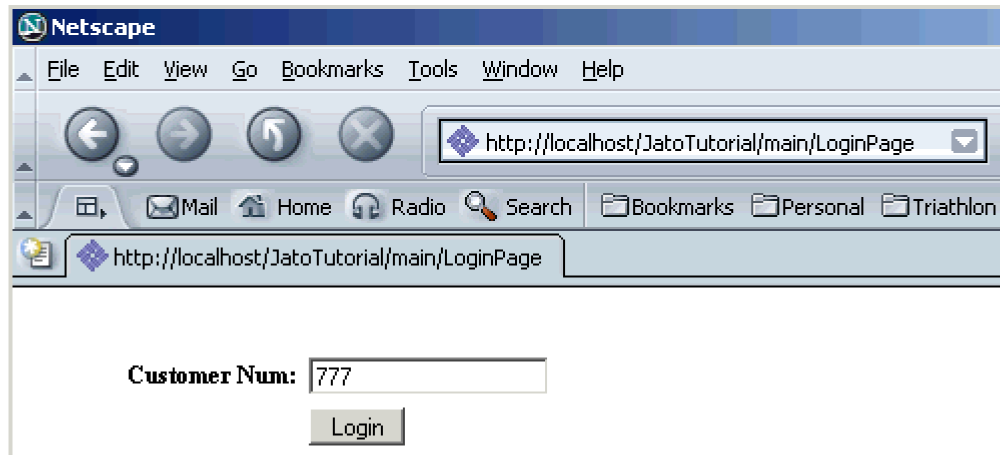
4. **Select the LoginPage node, and click the Execute Page (Redeploy) button**

   Using this execute and redeploy option restarts the server to ensure that the server picks up all changes and does not use any cached resources.



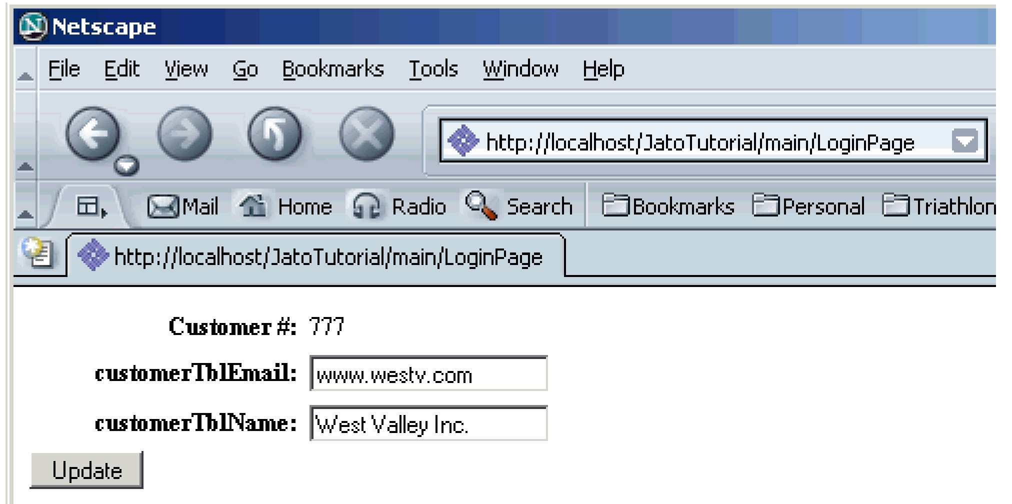
A default browser starts the application.

5. **Enter a valid customer number (1, 777, or 410).**



6. **Click Login.**

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


Netscape
File Edit View Go Bookmarks Tools Window Help
http://localhost/JatoTutorial/main/LoginPage
Mail Home Radio Search Bookmarks Personal Triathlon
http://localhost/JatoTutorial/main/LoginPage
Customer #: 777
customerTblEmail: www.westv.com
customerTblName: West Valley Inc.
Update

CHAPTER 13

Tutorial—Section 3.1 Create a Command Component

This chapter describes how to create a Command component that can be reused by many command fields (button and HREF components) within the same application. This is the alternative technique to implementing request handling code in the command field's handle request event inside its parent page/pagelet class (handleLoginRequest in LoginPage, for example).

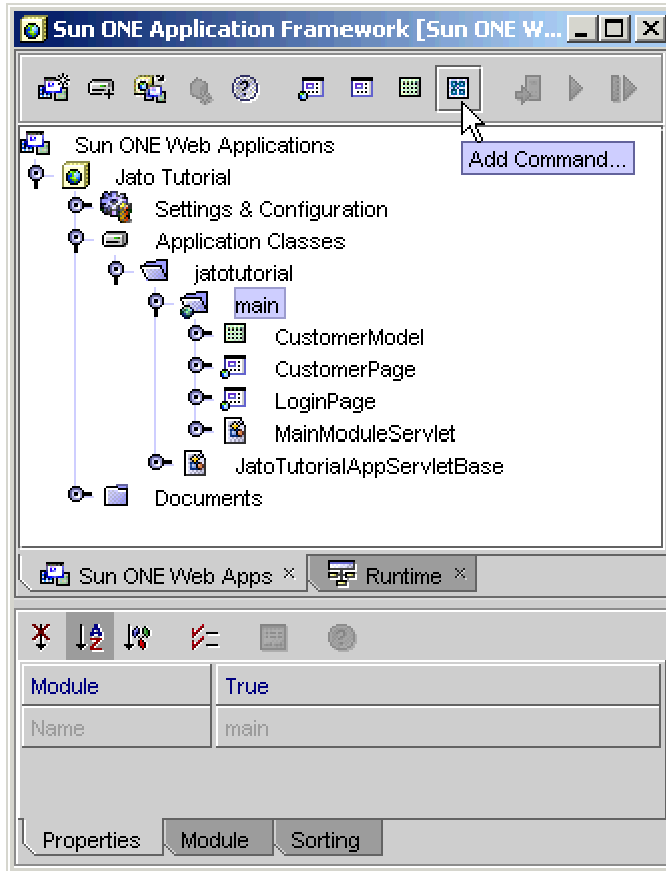
Commands provide great power and flexibility when it comes to code reuse. Any arbitrary Java class can become a Command component simply by implementing the com.iplanet.jato.command.Command interface. In this tutorial, you will create a new Command class using the Sun™ ONE Application Framework Command wizard to create a login/logout command which will replace the request handler event for the Login button. This Command component can then be reused by command fields on other pages and pagelets if required.

Task 1: Create a Command Component

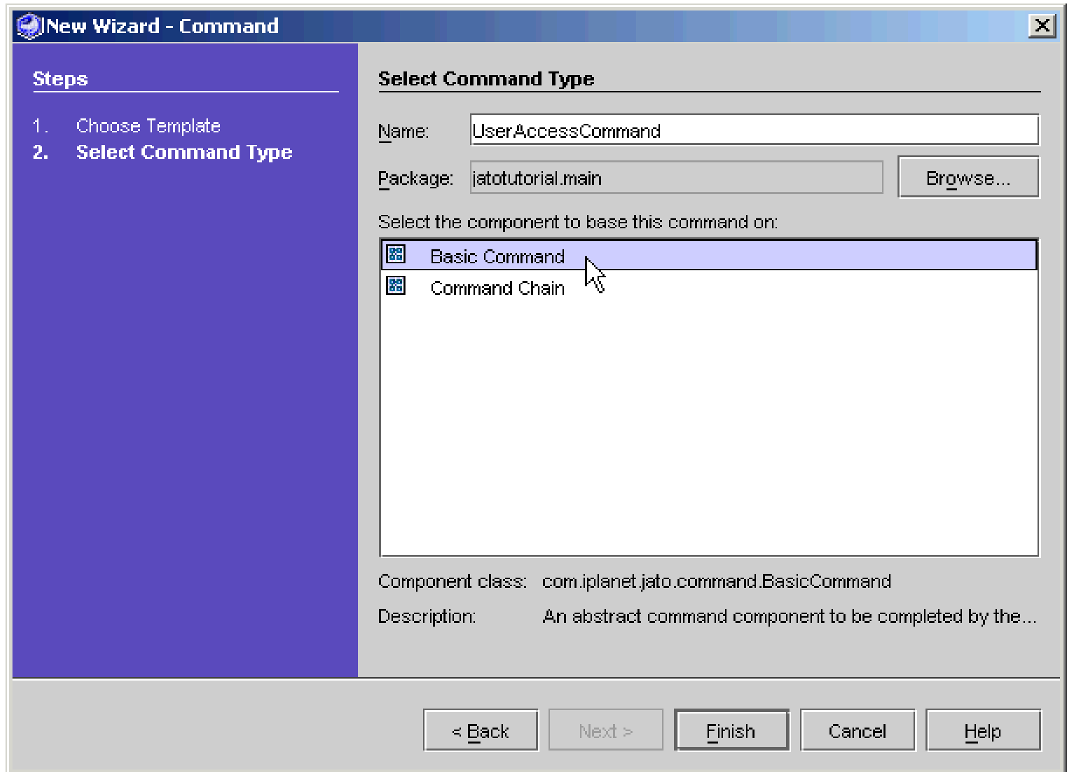
Create the UserAccessCommand Component

Create a Command component using the Sun ONE Application Framework Command wizard.

1. **Select the main module folder, then click the Add Command button on the Sun ONE Application Framework toolbar.**

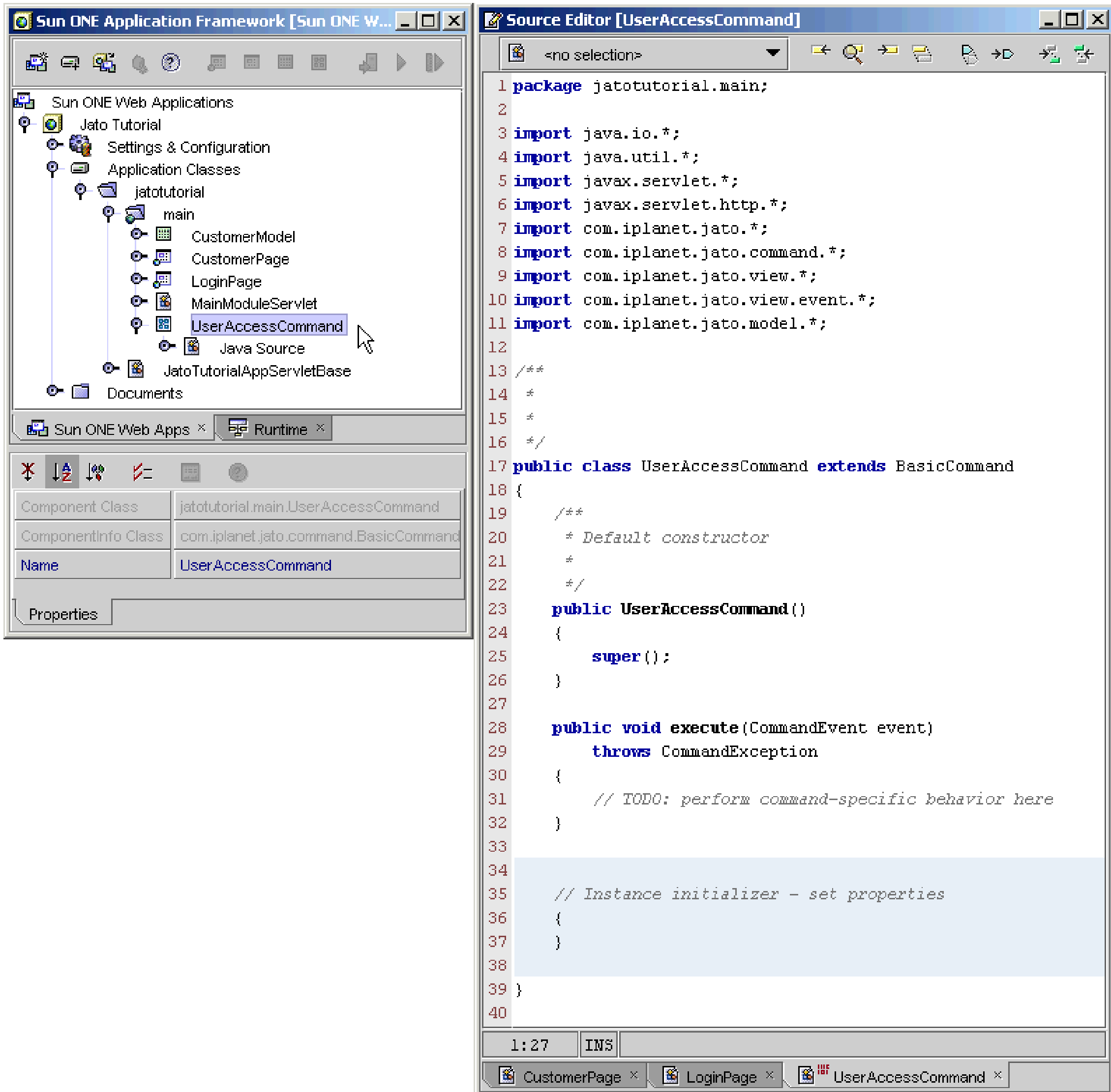


The Select Command Type panel displays.



2. **Enter *UserAccessCommand* in the Name textbox.**
3. **Select Basic Command.**
4. **Click Finish.**

   The UserAccessCommand component is added to the application.



Double-click the UserAccessCommand to open the Java source code for this component in the editor window.

It is quite a simple class that extends BasicCommand. BasicCommand implements the Command interface which declares only one method:

```
public void execute(CommandEvent) throws CommmandException
```

This command currently does nothing. You will need to add some code to the execute method to do what you need it to do, which is, to perform a user login or logout based on the *operation name* that is passed in via the CommandEvent parameter. The operation names are completely up to the developer (you). The following steps and tasks instruct you on how to pass and evaluate your custom operation names.

Add Code to the execute Method

This step requires nothing more than writing a little code. This might seem like a lot of code, but much of it is reimplementation of the handleLoginRequest event from the LoginPage. This replaces the need for that button event.

Add the following code to the execute method of the UserAccessCommand class.

```
public void execute(CommandEvent event) throws CommandException
{
    // get the RequestContext
    RequestContext requestContext = event.getRequestContext();

    // get the J2EE HttpSession
    javax.servlet.http.HttpSession session =

        requestContext.getRequest().getSession();

    // get the operation name that was passed in
    // by the commandfield object (button/href)
    String opName = event.getOperationName();

    // get the LoginPage
    LoginPage loginVB = (LoginPage)requestContext
        .getViewBeanManager().getViewBean(LoginPage.class);

    // perform user login
    if (opName.equals("login"))
    {
        // get the customer number that was entered
        int custNum = loginVB.getDisplayFieldIntValue(
            LoginPage.CHILD_CUSTOMER_NUM);

        // get the Customer model
        CustomerModel customerModel = (CustomerModel)requestContext
            .getModelManager().getModel(CustomerModel.class);

        // execute the CustomerModel with the customer number as criteria
        // to see if the user exists in the database
```

```
        customerModel.clearUserWhereCriteria();
        customerModel.addUserWhereCriterion(
            "CUSTOMER_TBL_CUSTOMER_NUM", new Integer(custNum));

        try
         {
            customerModel.executeSelect(null);
         }

        catch (ModelControlException e)
         {
            Log.log("Exception caught in UserAccessCommand.execute(): "
                 + e.toString());
         }

        catch (java.sql.SQLException e)
         {
            Log.log("Exception caught in UserAccessCommand.execute(): "
                 + e.toString());
         }

        // valid customer number entered
        if (customerModel.getNumRows() == 1)
         {
            // Display the Customer page
            requestContext.getViewBeanManager().getViewBean(
                 CustomerPage.class).forwardTo(requestContext);
            // put the customer number into an HttpSession attribute
            // for potential use in a later request
            session.setAttribute("hsaCustNum", new Integer(custNum));
         }

        // invalid customer number entered
        else
         {
            String msg = "Sorry, " + custNum +
                 " is not a valid customer number.";

            // Set the output status message
            loginVB.getDisplayField(
                 LoginPage.CHILD_MESSAGE).setValue(msg);
            loginVB.forwardTo(requestContext);
         }
    } // if opName = login

    // perform user logout
    else if (opName.equals("logout"))
    {
         // get the customer number from session to use in the logout message
```

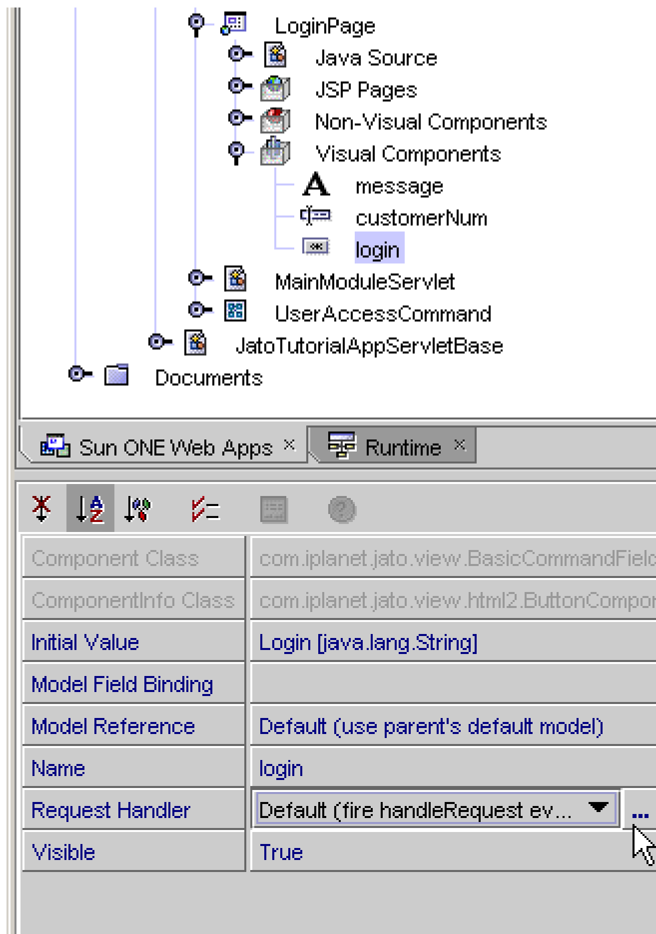
```
        String hsaCustNum = session.getAttribute("hsaCustNum").toString();

        String msg = "Customer " + hsaCustNum +
           ", you have logged out successfully.";

        // invalidate the user's HttpSession
        session.invalidate();

        // Set the logout message and display the Login page
        loginVB.getDisplayField(LoginPage.CHILD_MESSAGE).setValue(msg);
        loginVB.forwardTo(requestContext);
    } // else if opName = logout

    else
        throw new CommandException(
                "Unknown UserAccessCommand operation name: " + opName);
}
```

Before you can test run this code, you need to configure a command field (Button or HREF) to use it.

Configure a Button's Command Descriptor

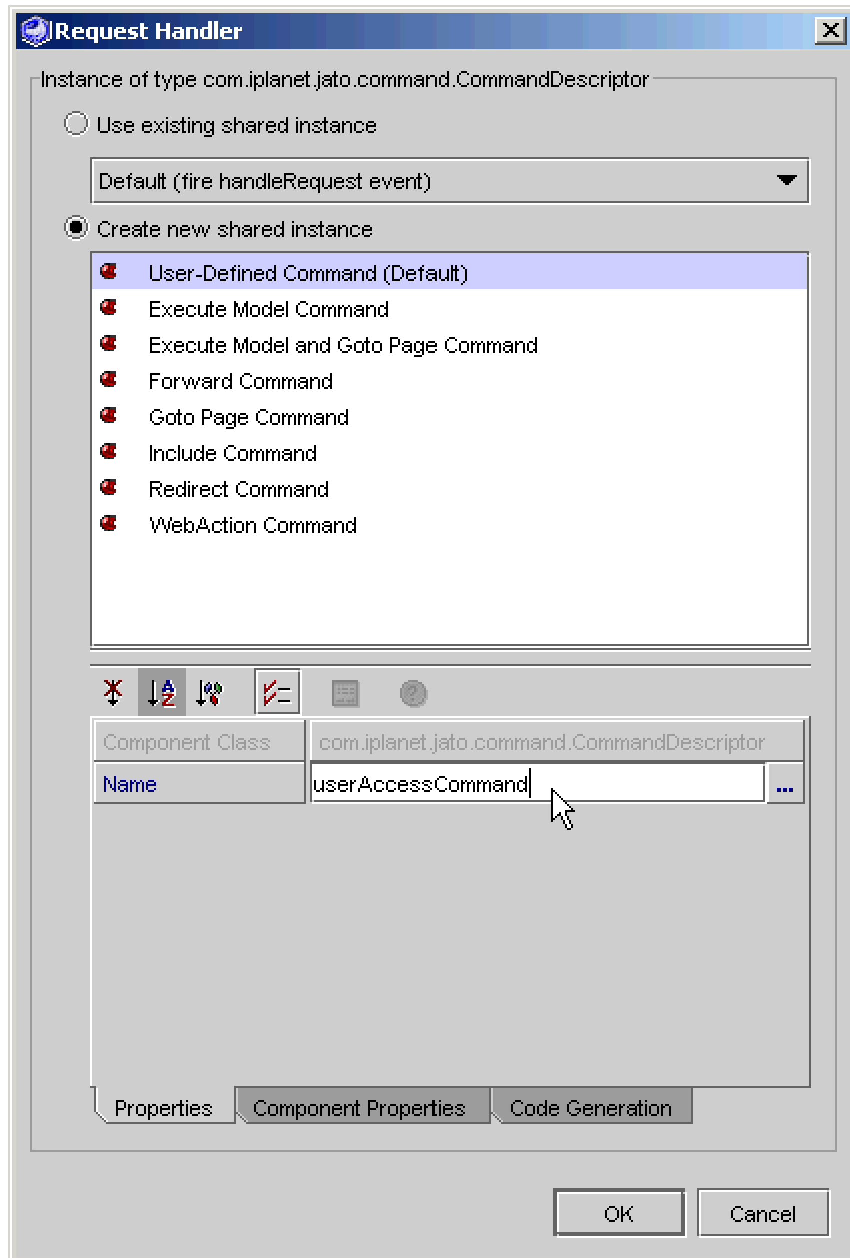
Configure the login button to use the UserAccessCommand component via the Command Descriptor property of the button. This also works the same for HREFS.

1. **Expand the LoginPage node, and expand the Visual Components node.**
2. **Select the login button under Visual Components.**



3. **Click the ellipsis button for the *Request Handler* property.**

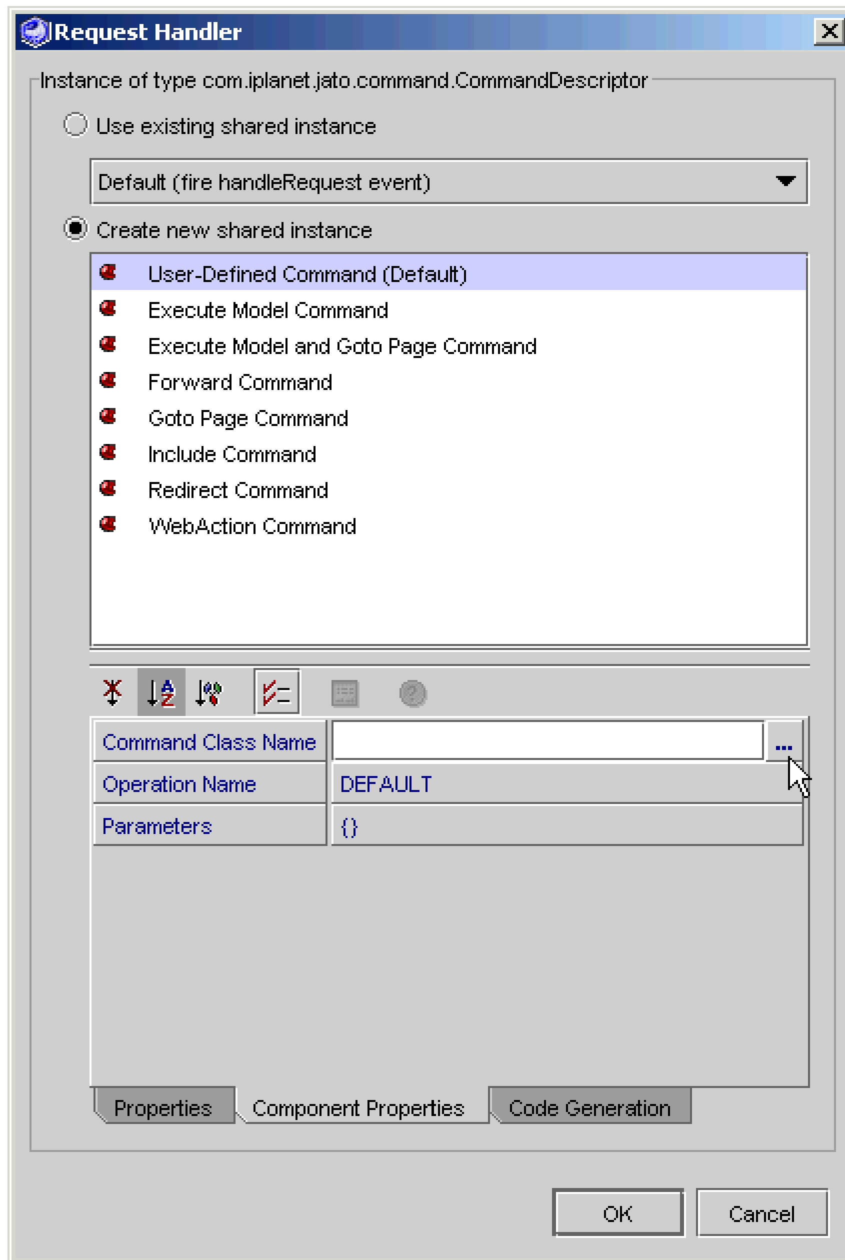
   This displays the Command Descriptor editor.



4. **Select *User-Defined Command (Default)* from the list under the *Create new shared instance* radio button choice.**

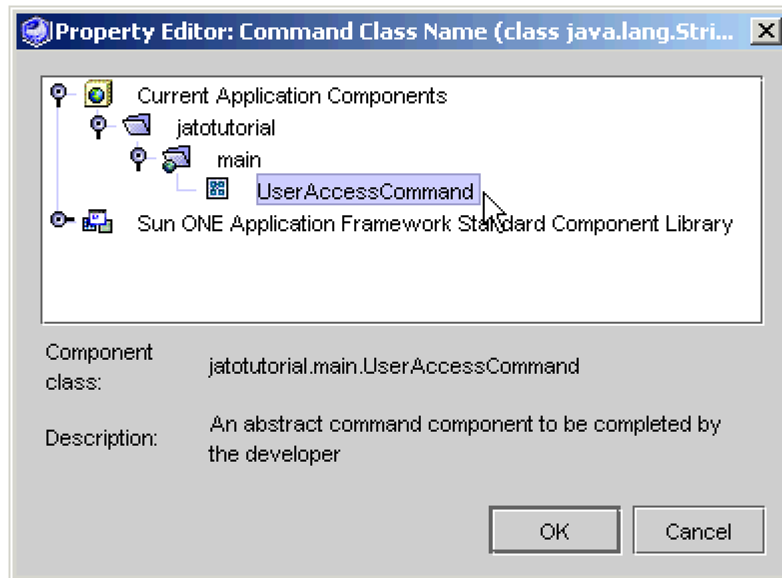
5. **Change the Name property to userAccessCommand**

6. **Select the Component Properties tab at the bottom of the editor.**



7. **Click the ellipsis button for the *Command Class Name* property.**

   This displays the Command Class Chooser dialog.



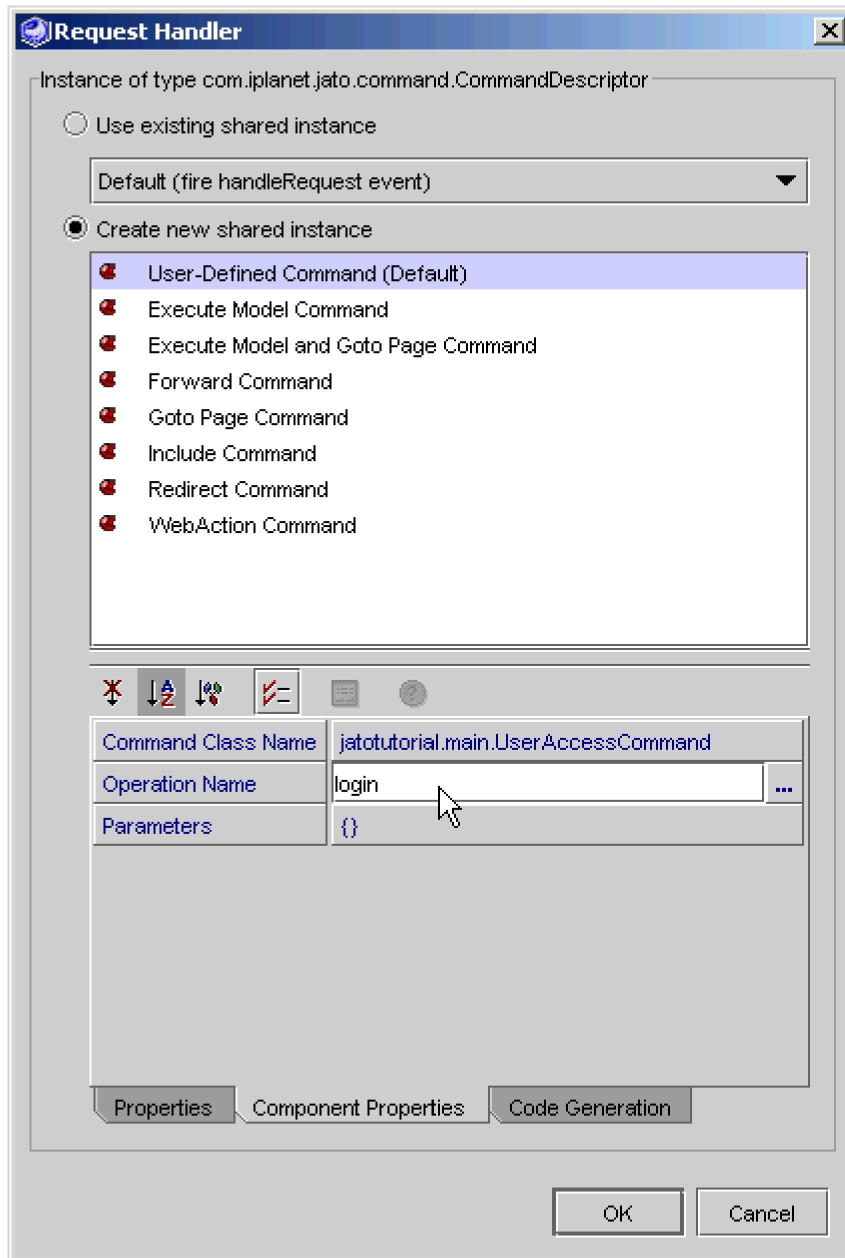
8. **Expand the Current Application Components node, then jatoturial, and then main.**

9. **Select the UserAccessCommand command component.**

10. **Click OK.**

11. **Change the Operation Name from *DEFAULT* to *login*.**

    Recall in the code what you implemented for the execute method in the UserAccessCommand class. You have an if/else block that is expecting either *login* or *logout* as an operation name. These are case sensitive, so you need to be sure you set this correctly, or you will receive the CommandException (Unknown operation name) when you test run this command.



12. **Click OK to finish setting the Command Descriptor property for the login button.**

    Now, when you run the Login page and click the Login button, the UserAccessCommand component handles the request instead of the code in the handleLoginRequest event in the LoginPage.

You can leave the code in the handleLoginRequest event as is, because it will be never be invoked, unless you reconfigure the login button to use the request handler event instead of the command component.

This is because the Sun ONE Application Framework first looks for a Command Descriptor for the command field. If the Command Descriptor is not implemented, it then attempts to invoke the handle<CommandField>Request event. If the event is not implemented, you receive a *request handler not found* exception.

CHAPTER 14

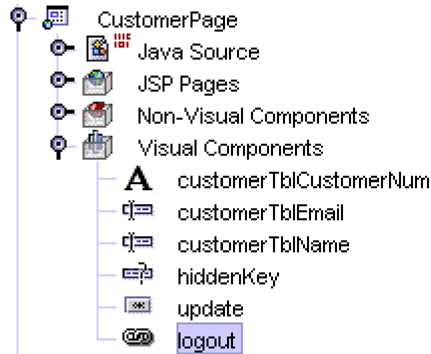
Tutorial—Section 3.2
Add a Logout Link to the Customer Page

This chapter describes how to add an HREF to a page that uses a Command component.

Task 2: Add an HREF to a Customer Page

1. **Select the CustomerPage node.**
2. **Add a Hyperlink (HREF) component using the Component Palette.**

   An HREF command field is added to the CustomerPage's Visual Components node.
3. **Rename the HREF as *logout*.**

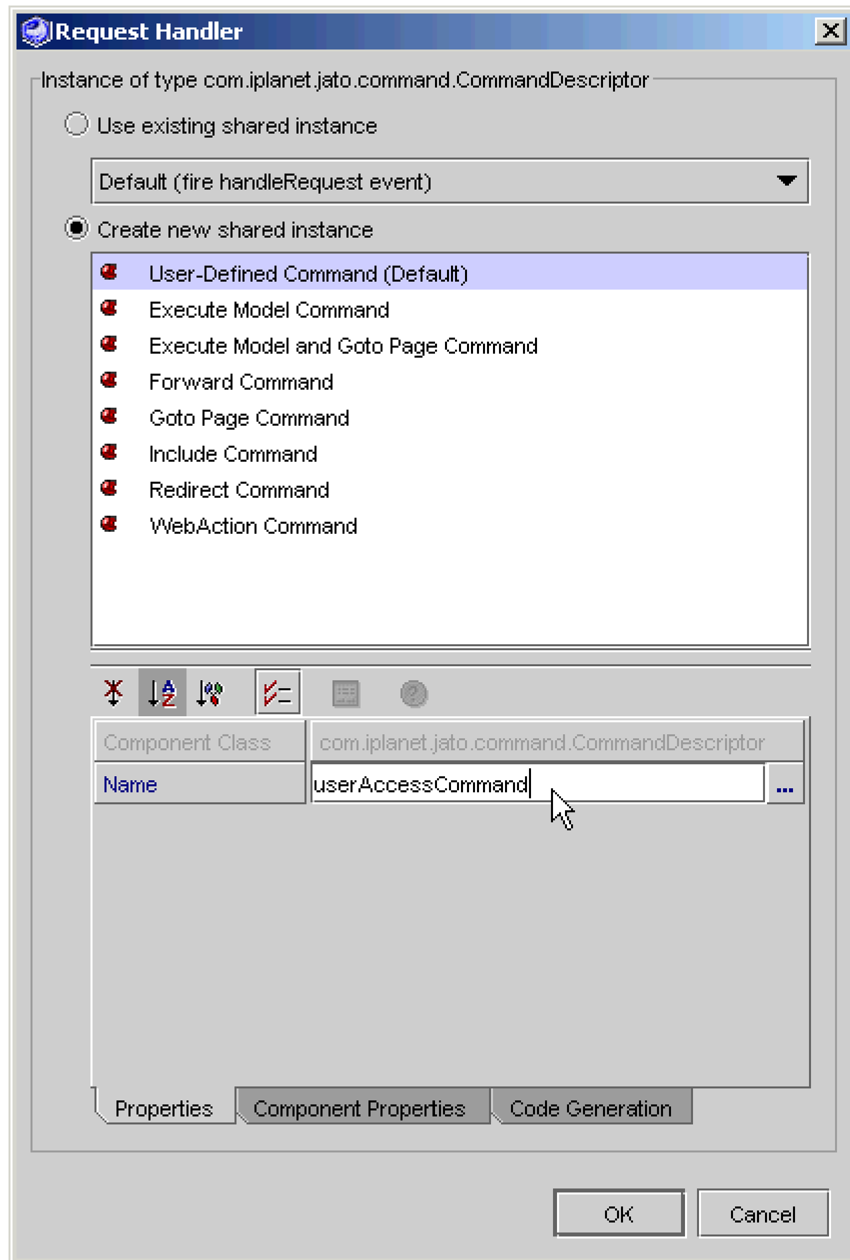


Configure an HREF's Command Descriptor

Configure the logout HREF to use the UserAccessCommand component via the Command Descriptor property of the button. This is identical to the button Command Descriptor configuration in the previous task, except the *operation name* will be *logout* instead of *login*.

1. **Select the logout HREF under CustomerPage's Visual Components node.**
2. **Click the ellipsis button for the Request Handler property.**

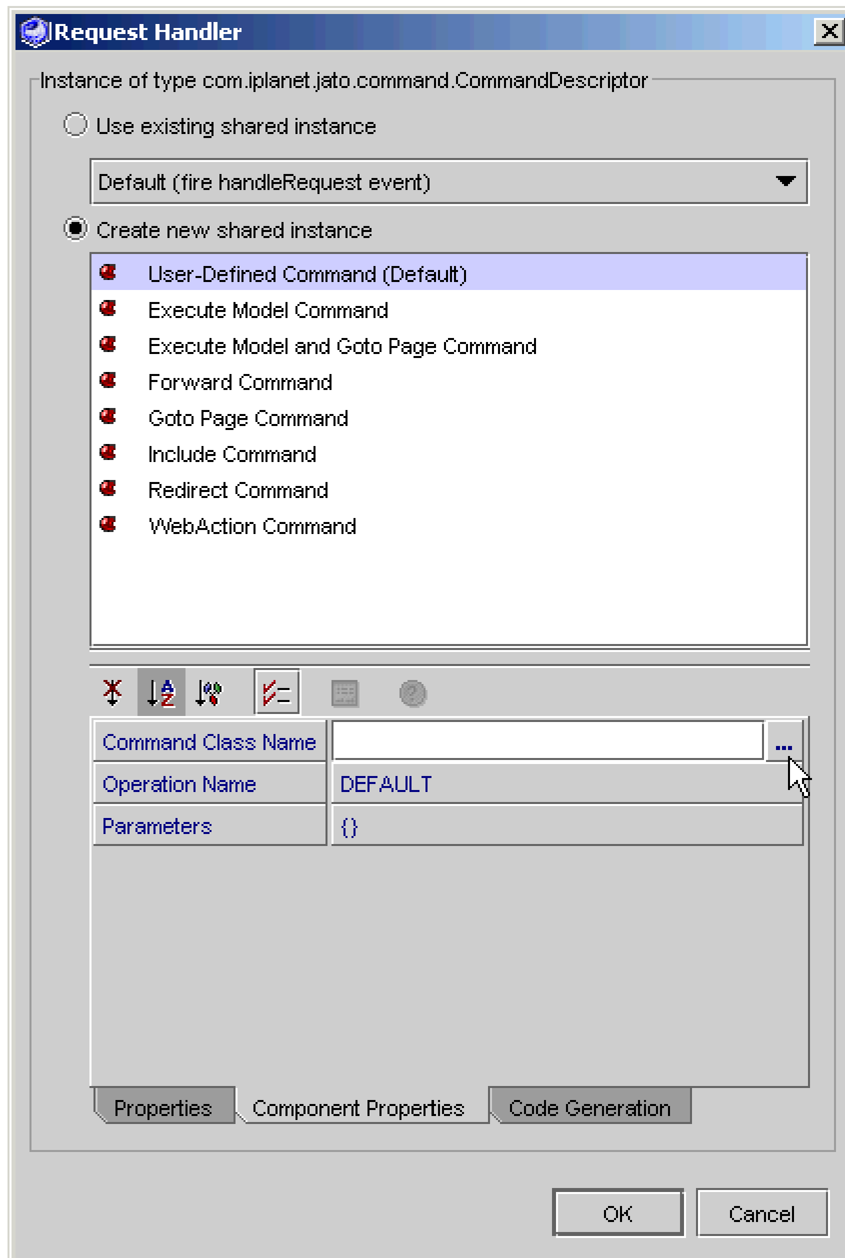
   This displays the Request Handler editor.



3. **Select *User-Defined Command (Default)* from the list under the *Create new shared instance* radio button choice.**

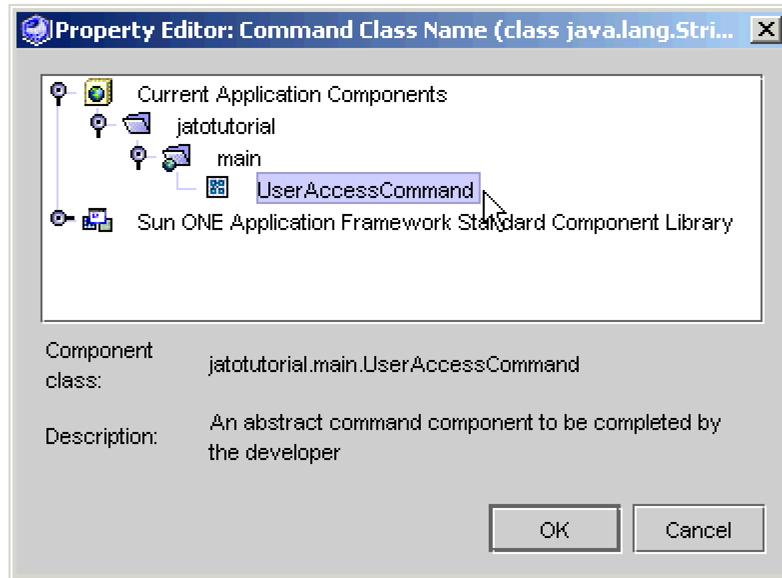
4. **Change the name to userAccessCommand**

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



6. **Click the ellipsis button for the *Command Class Name* property.**

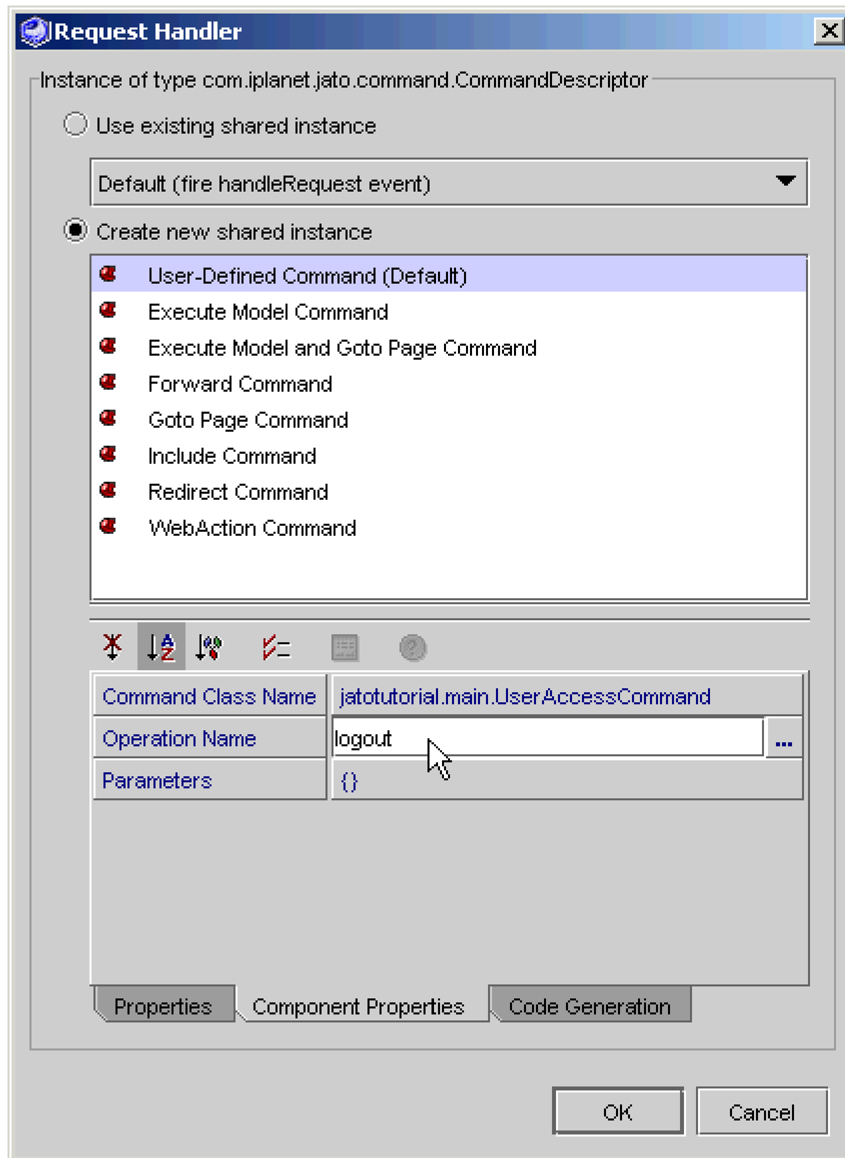
This displays the Command Class Chooser dialog.



7. **Expand the Current Application Components node, then expand jatoturial, and then main.**
8. **Select the UserAccessCommand command component.**
9. **Click OK.**
10. **Change the Operation Name from *DEFAULT* to *logout*.**

    Recall in the code what you implemented for the execute method in the UserAccessCommand class. You have an if/else block that is expecting either *login* or *logout* as an operation name.

    These are case sensitive, so be sure you set this correctly, or you will receive the CommandException (Unknown operation name) when you test run this command.



11. Click OK to finish setting the Request Handler property for the logout HREF.

When you login with a valid customer number, you are taken to the Customer page. The logout link displays. When clicked, the logout operation name is passed into the UserAccessCommand to invalidate the user's session and display the Login page with the logout message.

Format the HREF tag in the Customer JSP

When you added the *logout* HREF field to the CustomerPage, an HREF tag was added to the `CustomerPage.jsp` file. However, the link displays with the default name of the HREF, *href1*, which is not the required text.

1. **Expand the JSPs node under CustomerPage.**
2. **Double-click the CustomerPage JSP node to open the JSP in the editor window.**
3. **Find the logout HREF tag and modify the body content portion to display *Logout* instead of *href1*.**

```
<jato:href name="logout">Logout</jato:href>
```

You can position the HREF tag anywhere you prefer, so long as it is nested between the *useViewBean* tags and is part of the HTML's body section (between the *body* tags).

Unlike the button, an HREF is not required to be part of the form, so it can be positioned outside the *form* tags (*<jato:form>*).

CHAPTER 15

Tutorial—Section 3.3 Test Run the Login/Logout Command Component

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

Task 3: Test Run the Login/Logout Command

Important: Make sure the PointBase Network Server is running. If it is not, you can start it in the Sun™ ONE Studio as follows:

1. **Select menu option Tools -> PointBase Network Server -> Start Server.**

   Since you have created a new class and made modifications to two other classes, be sure to compile/deploy the application.

2. **Right-click the Application Classes node, and select the Compile All action.**

3. **If you are running on Sun™ ONE Application Server, you must Deploy the application when changes are made.**

   Select the Sun ONE Application Framework Application node (JatoTutorial), and click the Deploy button on the Sun ONE Application Framework toolbar.

4. **Select the LoginPage node, and click the Execute Page (Redeploy) button**

   Using this execute and redeploy option restarts the server to ensure the server picks up all changes and does not use any cached resources.

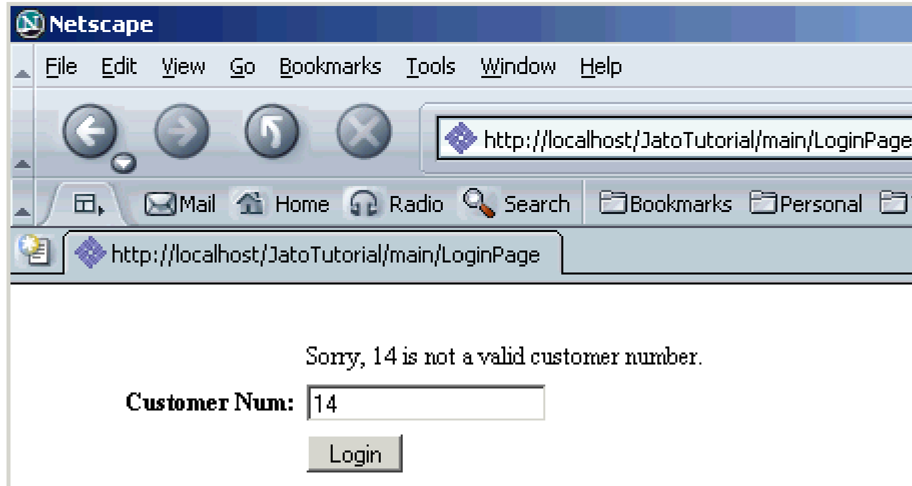
   A default browser starts the application.

Note – In sections 3.1 and 3.2, you hardcoded three customer numbers into the login validation. The new UserAccessCommand will validate the entered customer number against the database.

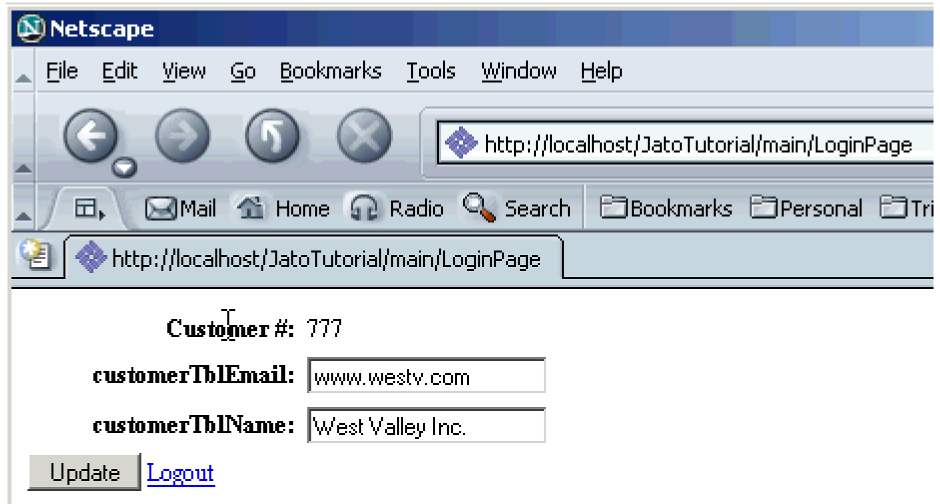
For your convenience, a list of the valid customer numbers from the sample PointBase database is as follows: 1, 2, 3, 25, 36, 106, 149, 409, 410, 722, 753, 777, 863

---

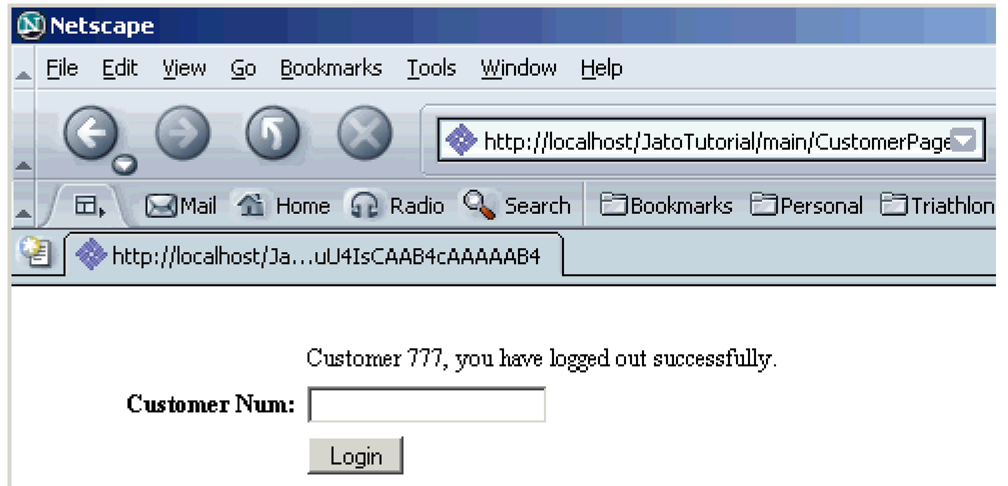
5. **Enter an invalid customer number first.**



6. **Enter a valid customer number.**



7. **Try the Logout link.**



CHAPTER 16

Tutorial—Section 4.1 Prepare to Create a Web Service Model

This chapter describes how to expand the application to access data via a Web service. You must be running a version of the Sun™ ONE Studio that includes the Web service model wizard. You must also have a connection to the Internet without a proxy/firewall which will interfere with communication to the Web service.

You can expand the existing application by adding a Web service-based model and a page to display that model's data. First, there are some resources you need to download to build a model for a Web service, and you need to register as a user for this Web service.

Task 1: Web Service User Registration and Downloading

Download the Web Service SDK

Download the Google Web service software SDK that contains the WSDL file that the Sun™ ONE Application Framework needs to create the Web Service model.

1. **To download the Google Web Service SDK, go to** http://www.google.com/apis/download.html
2. **Accept the agreement.**
3. **Click the Download button.**
4. **Save the file on your hard drive.**

Open the zip file and extract the `googleapi/GoogleSearch.wsdl` file to your application's lib directory (`.../JatoTutorial/WEB-INF/lib`). There are three versions of this file in the zip file. Be sure to get the only one that is *not* under the *dotnet* directory. That is all you need to build the Web service model.

Note – When you copy a new file into your applications file structure, occasionally it takes the Sun ONE Studio some time to refresh its state.

If it appears that the Sun ONE Studio is taking too long to recognize the new file, you can go to the Filesystems or Project tab, right-click the lib directory, and select the Refresh Folder action.

Register to Use the Web Service

To use the Google Web service, you must register as a user to receive a key that is passed to the Web service with each query.

1. **To register with Google, go to**

   `https://www.google.com/accounts/NewAccount?continue=`
   `http://api.google.com/createkey&followup=`
   `http://api.google.com/createkey`

2. **Enter an email address and password to register a new account.**

   You will receive an email to verify your account. Once you verify your account, you will receive another email with your key (it is a long string of letters and numbers). Keep this email handy because you will need it when you create the Web service model.

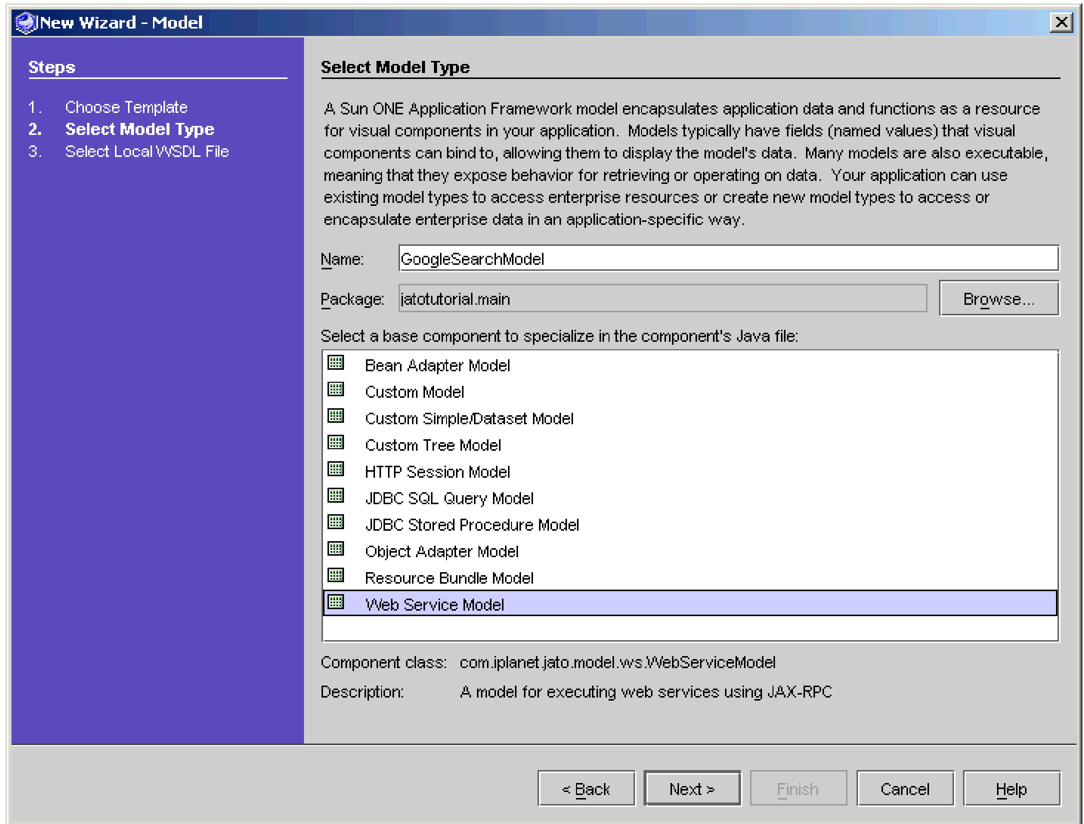
Create the Web Service Model

Using the WSDL file, you will create a Web service model that will perform an Internet search using the Google Internet search engine via their Web service.

1. **Select the main module folder.**

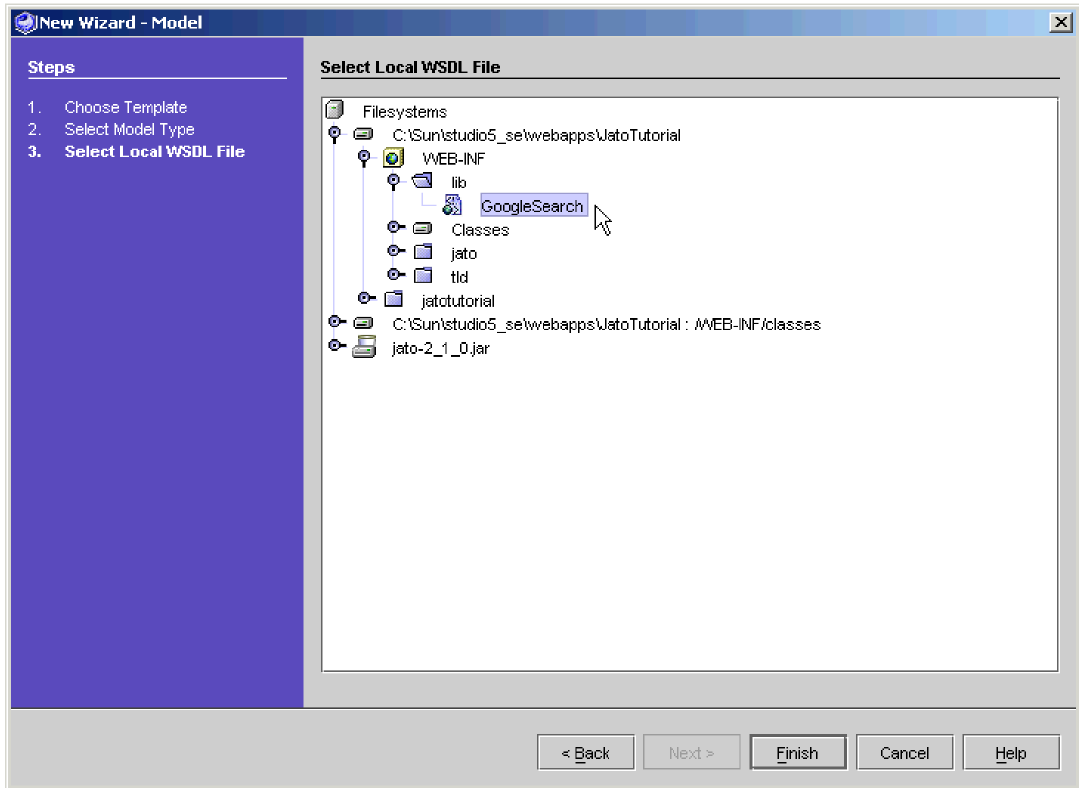
2. **Click the Add Model button on the Sun ONE Application Framework toolbar.**

   The Select Model Type panel displays.



3. **Enter *GoogleSearchModel* in the Model name textbox.**
4. **Select *Web Service Model* from the model component list.**
5. **Click Next.**

   The Select Local WSDL File panel displays.

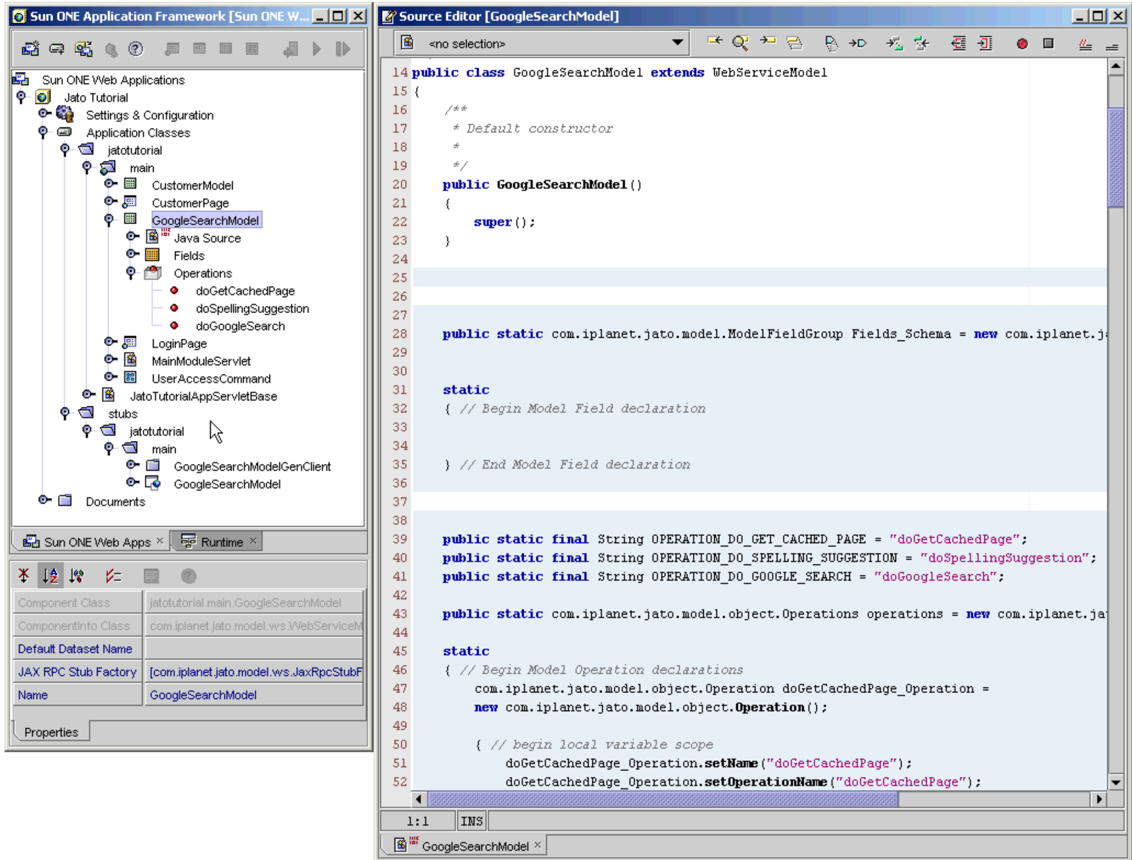


Navigate down the JatoTutorial application directory structure to the `lib` directory (`JatoTutorial/WEB-INF/lib`), and select GoogleSearch (the WSDL file).

Your file might be in a subdirectory of the `lib` directory. When you extracted it from the zip file, it was in a base directory called `googleapi`.

6. **Click Finish to create the Web service model.**

   The GoogleSearchModel object is created in the main module.



7. **Double-click the GoogleSearchModel node to view the code in the GoogleSearchModel class.**

   This Web service has a few operations that can be used. The following tasks focus only on the *doGoogleSearch* operation.

Note – When you look at your application file structure, you see a new folder named stubs. This folder was created by the Web Service Model wizard as a package to store any stub classes that are needed to support the use of Web services.

This is one of many great benefits provided the Web service model wizard. To use a Web service, there are many classes that need to be created. Browse this package folder to see how much work was actually performed.

Rest assured that you will not have to look at these files. All of the tedious work is done for you. You only need to work with the Web service model class and, even then, it requires only minimal manual coding, or most times, none at all.

CHAPTER 17

Tutorial—Section 4.2
Create the Google Search Page

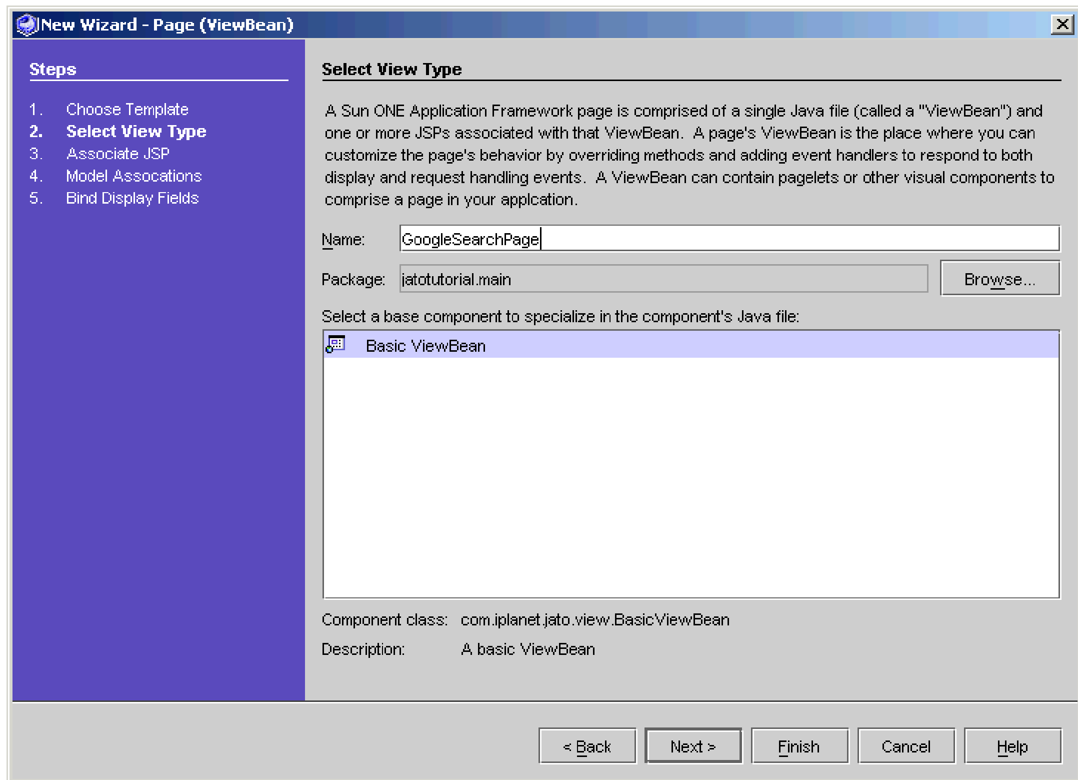
This chapter describes how to create a page in the Sun™ ONE Application Framework that displays data it gets from a model that accesses data from a Web service.

Task 2: Create the Google Search Page

Add a Page Component

1. **Select the main module folder.**
2. **Click the Add Page button on the Sun ONE Application Framework toolbar.**

   The Select View Type panel displays.

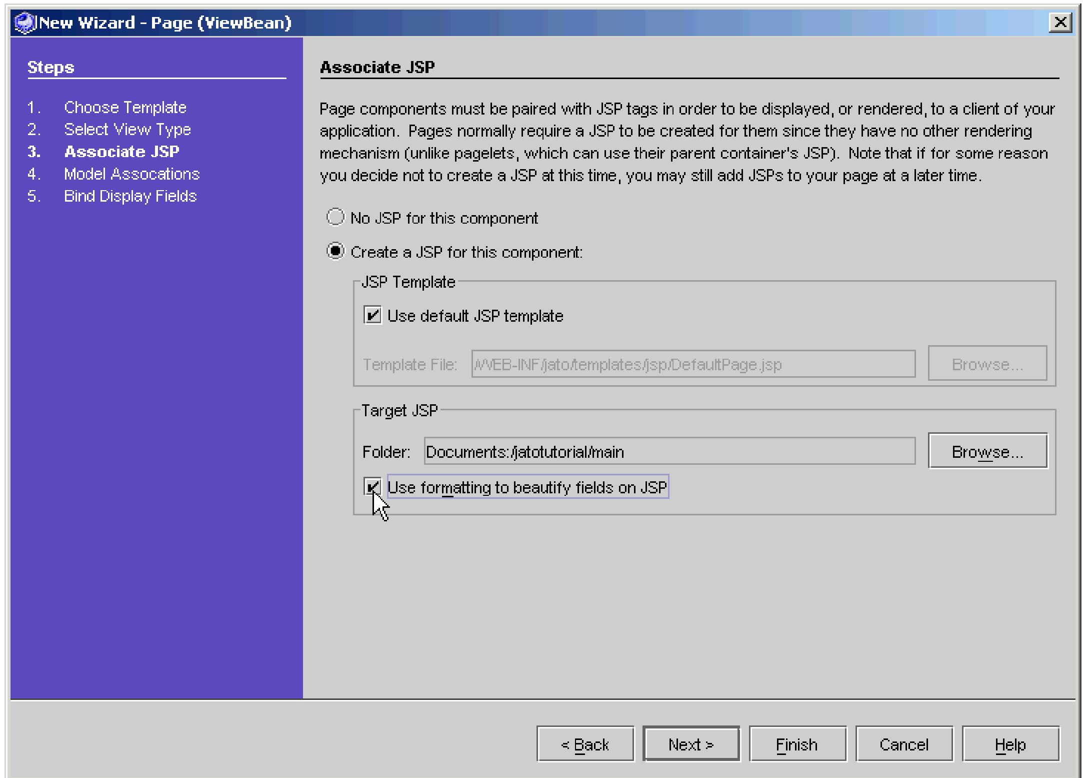


3. **Enter GoogleSearchPage in the Name field (to replace <*default*>).**

4. **Select Basic ViewBean.**

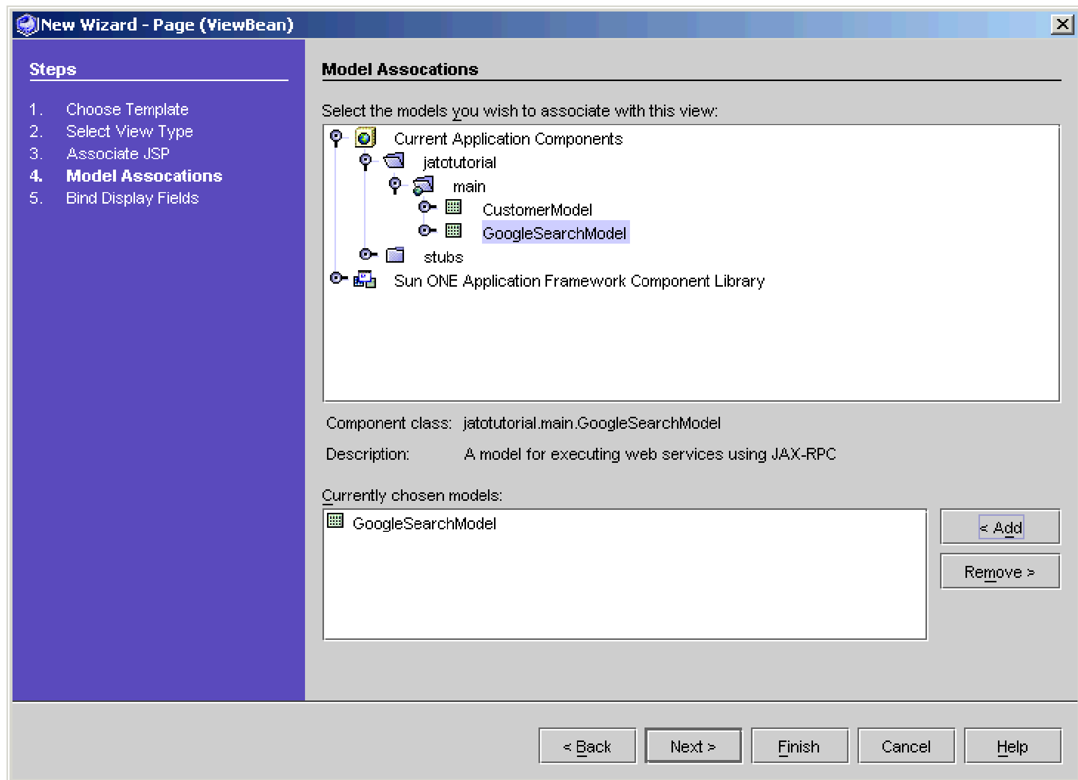
5. **Click Next.**

   The Associate JSP panel displays.



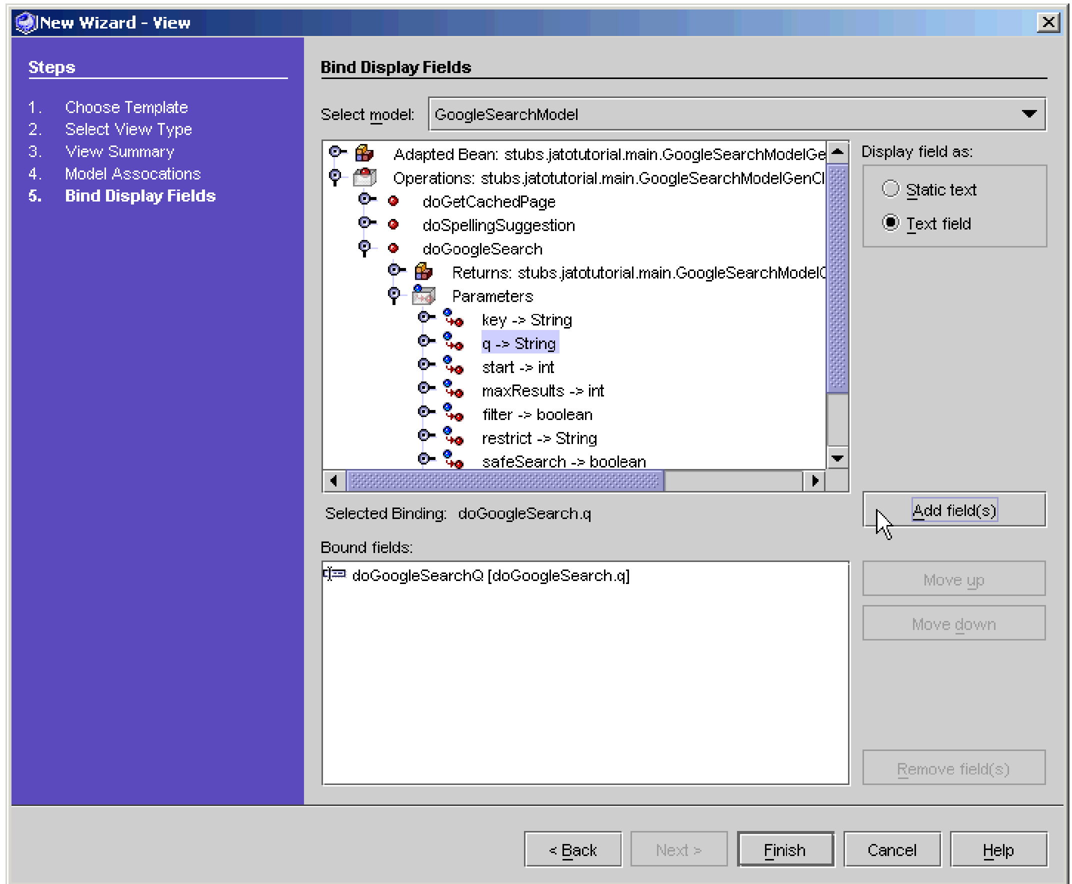
6. **Check the *Use formatting to beautify fields on JSP* option.**
7. **Click Next.**

   The Model Associations panel displays.



8. **Expand Current Application Components to expose jatotutorial -> main.**
9. **Select GoogleSearchModel.**
10. **Click Add.**
11. **Click Next.**

    The Bind Display Fields panel displays.



12. **Add the first field (as seen above):**

    a. **Expand the doGoogleSearch operation node.**

    b. **Expand the Parameters node.**

    c. **Select the *q* field (q for query string: q -> *String*).**

    d. **Select the Text field option.**

    e. **Click *Add field(s)*.**

    The q field is added to the *Bound fields* list box.

Note – You are not finished with this wizard panel yet.

---

13. **Add the following fields as Text fields (WebService model fields are not multi selectable like JDBC model fields):**

a. start

b. maxResults

c. restrict

14. Add the following fields as Static text fields (not Text fields):

(Expand the *Returns: ...* node [above the Parameters node])

a. searchTime

b. startIndex

c. endIndex

d. estimatedTotalResultsCount

15. Click Finish.

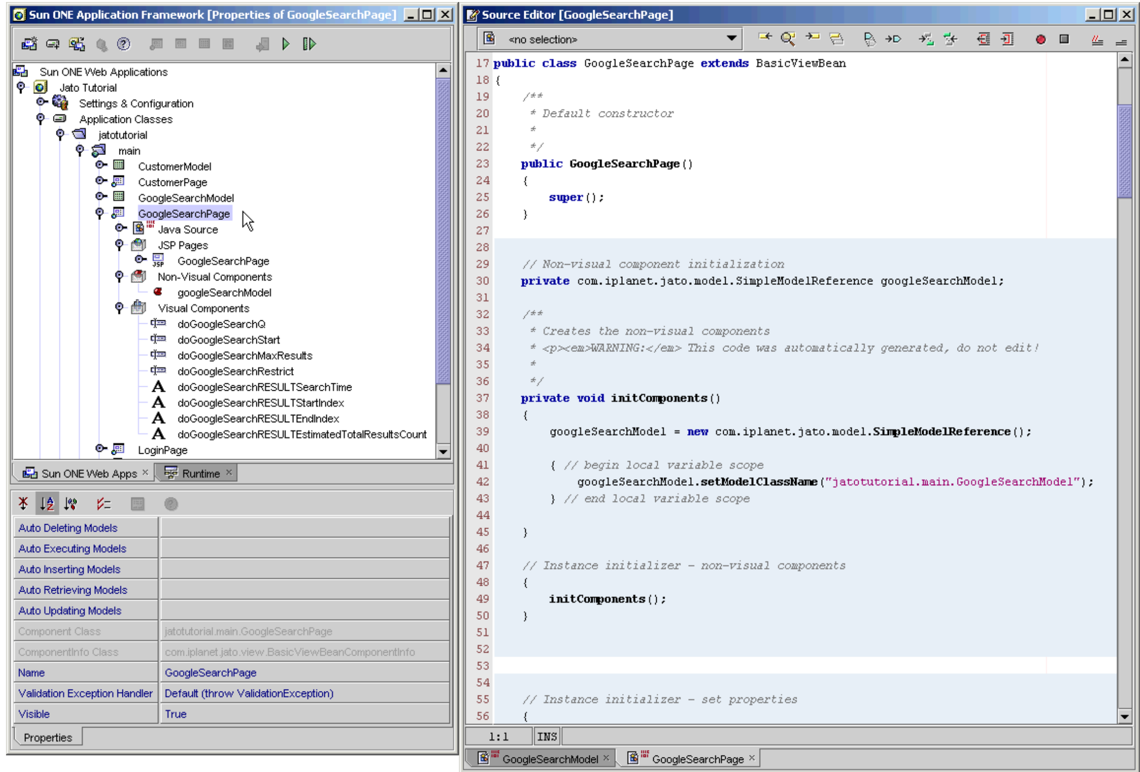
You have created the GoogleSearchViewBean.

16. The wizard automatically sets the Auto Retrieving Models property with the model that was used in the Page wizard. Normally, this is a desired setting, and therefore, the wizard assumes that it should be configured that way. However, with the Web service model, this is not the case.

a. Launch the Auto Retrieving Models custom editor by clicking its ellipsis button.

b. Select the googleSearchModel reference in the indexed list, and click the Delete button to remove it from the list.

c. Click the OK button to save the modifications.



17. **Rename the fields to have shorter, simpler names (select the field and click F2 to rename).**

    The following table shows the longer field names in the left column, and the shorter names in the right column.

doGoogleSearch**Q**	to queryString
doGoogleSearch**Start**	to start
doGoogleSearch**MaxResults**	to max
doGoogleSearch**Restrict**	to restrict
doGoogleSearchRESULTS**SearchTime**	to searchTime
doGoogleSearchRESULTS**StartIndex**	to startIndex
doGoogleSearchRESULTS**EndIndex**	to endIndex
doGoogleSearchRESULTS**EstimatedTotalResultsCount**	to estTotal

Set the properties for the **start** and **max** text fields according to the table shown below.

Note that you do not need to set properties for the **restrict** field.

Important: Launch the Initial Value editor by clicking the ellipsis button so you can select the Integer type. If you type the value in place in the property sheet, it is treated as a String.

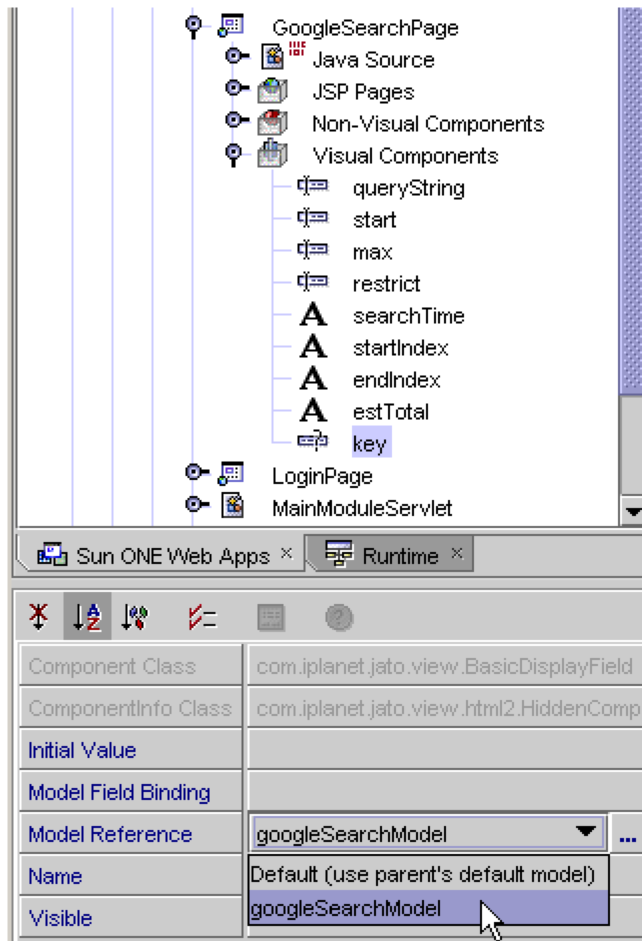
Name	Initial Value
start	Type: Integer Value: 0
maxResults	Type: Integer Value: 5

You have created four *search* fields and four *result* fields for this page component, but you will need a few more search fields (required fields by the Google Web service). These are added and bound to the GoogleSearchModel one at a time. You want these fields to be added as something other than text or static text fields, which is why you add these outside of the Page wizard.

Add More Visual Components to the Page

1. **Select the** `GoogleSearchPage`**.**
2. **Add a Basic Hidden Field using the Component Palette.**

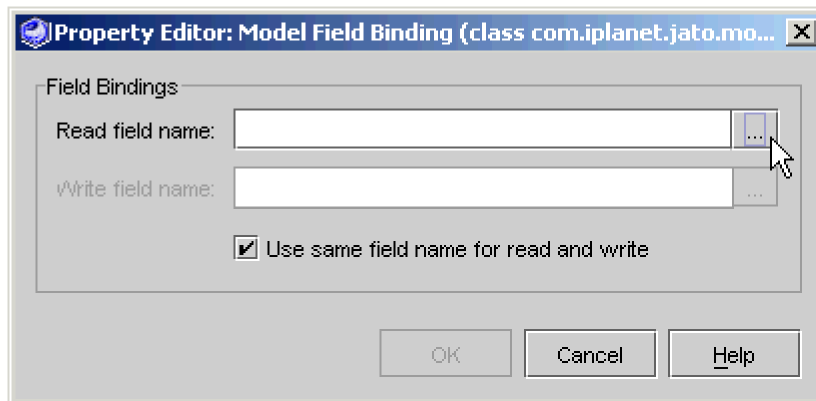
   The default name is *hidden1*.
3. **Rename the default as *key*.**
4. **Set the Model Reference property for the *key* field.**



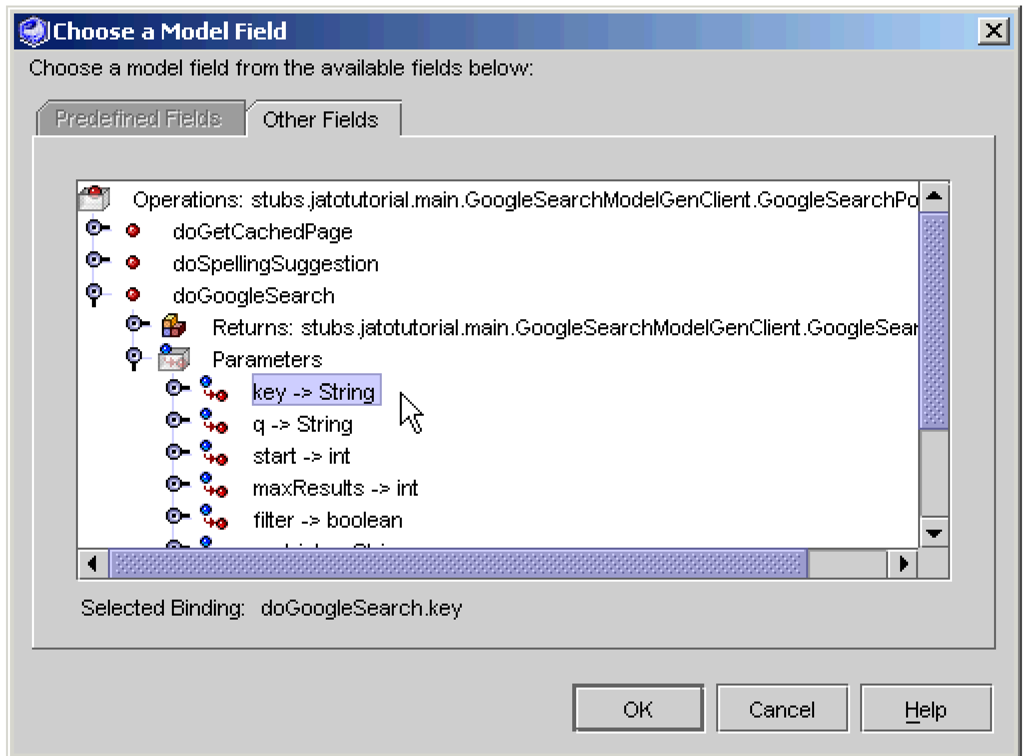
5. **Select googleSearchModel from the drop down list.**
6. **Set the Model Field Binding property.**

Component Class	com.iplanet.jato.view.BasicDisplayField
ComponentInfo Class	com.iplanet.jato.view.html2.HiddenComp
Initial Value	
Model Field Binding	...
Model Reference	googleSearchModel
Name	hidden1
Visible	True

7. **Click the ellipsis button to launch the Model Field Binding editor.**

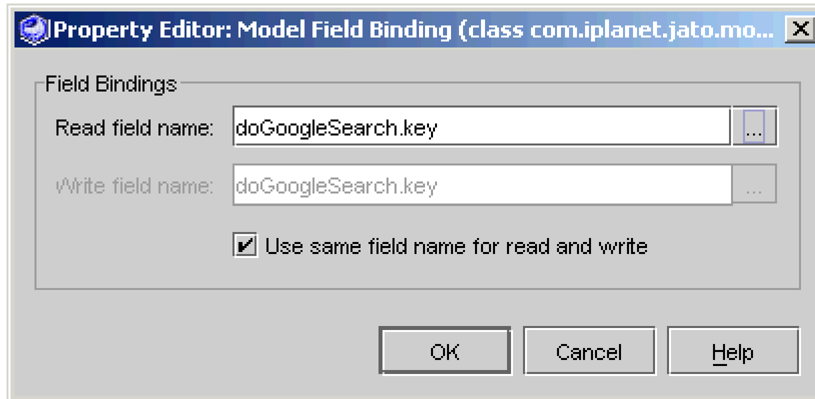


8. **Click the ellipsis button for the Read field name property in this editor.**



9. **Expand the doGoogleSearch operation node, then expand the Parameters node.**
10. **Select key -> String.**
11. **Click OK.**

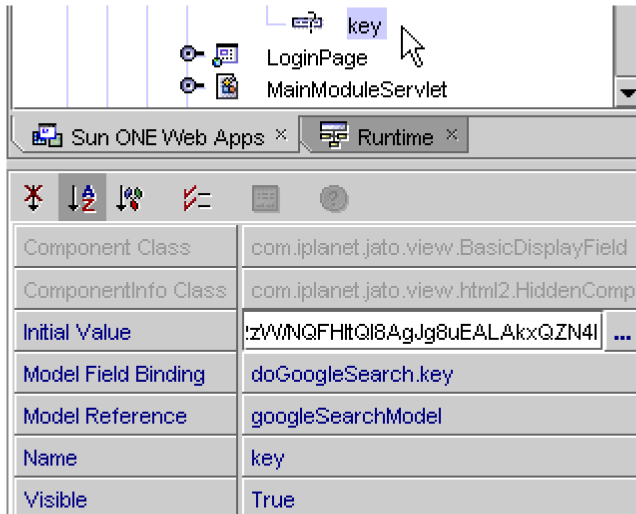
    The read and write fields are populated with the doGoogleSearch.key model field.



12. **Click OK to finish setting the Model Field Binding property for the key hidden display field.**

13. **Set the Initial value property (just above the Model field binding property) for the key field using the key that was emailed to you from Google.**

    The default Type for the Initial Value property is String. You do not need to launch the editor. Just enter the string value directly in the property cell.



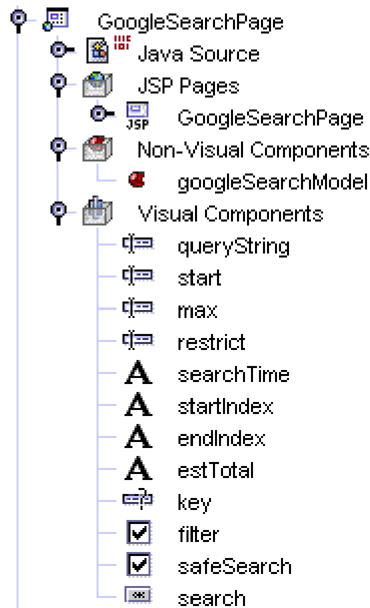
The key field's properties should look something as shown in the figure above, however, your key's initial value will be different.

14. **Using the Component Palette, add three more display fields.**

    The table shown below contains a list of the three display fields and the desired property settings.

Type	Name	Initial Value	Model Reference	Model Field Binding
Basic Check Box	filter		googleSearchModel	doGoogleSearch/Parameters/filter
Basic Check Box	safeSearch		googleSearchModel	doGoogleSearch/Parameters/safeSearch
Basic Button	search	Type: String Value: Search		

Your GoogleSearch Page node structure should look something the following figure:



Enable the Search Button

Currently, the search button has not been implemented to do anything when it is clicked. When the search button is clicked, you need to execute the Web service model and then reload the page to see the results. All you see now is the statistical information:

- start/end index
- estimated results count
- query time

In the next task, you add visual components to show a list of actual search results.

For the button, there are two techniques from which you can choose to execute the Web service model and reload the page. One technique is to write a few lines of code. The other technique is all point-and-click. Choose only one technique to implement.

Manual Code Technique

1. **Right-click the search button.**
2. **Select Events.**
3. **Select handleRequest.**

   This inserts the handleSearchRequest event stub into the GoogleSearchPage class.
4. **Implement the search button handle request code.**

   Replace the following default code:

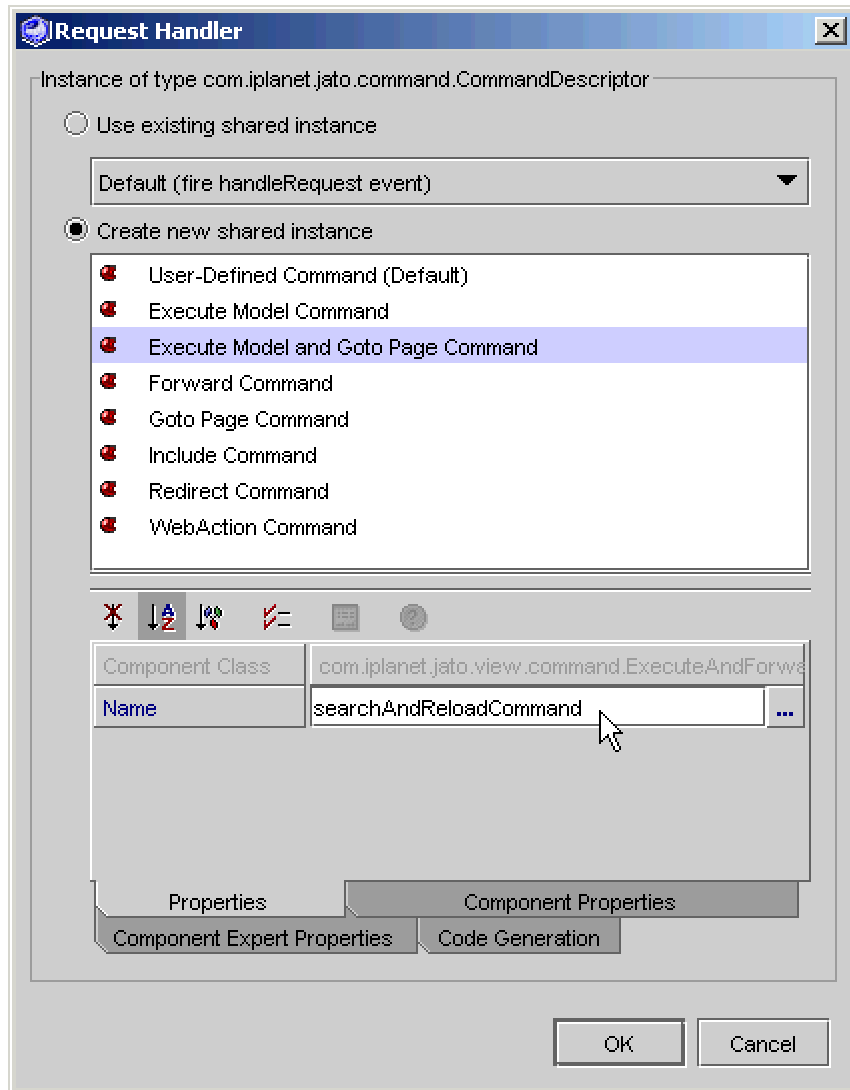
   `getParentViewBean().forwardTo(getRequestContext());`

   with the code shown in **bold** below:

```
public void handleSearchRequest(RequestInvocationEvent event) throws Exception
{
      // get a reference to the Google web service model
      GoogleSearchModel model = (GoogleSearchModel)getModel(
                                   GoogleSearchModel.class);

      // execute the model using the doGoogleSearch operation
      // (the model execution context)
      model.execute(new ModelExecutionContextBase("doGoogleSearch"));

      // redisplay the page which will now show the query statistical results
      forwardTo();
}
```

Point & Click Technique (code-free)

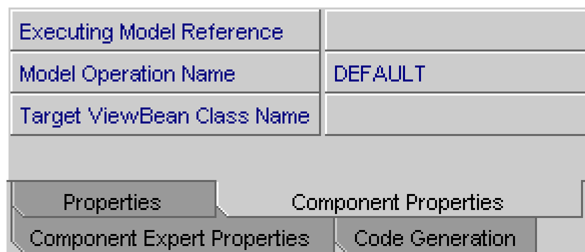
1. **Select the search button.**
2. **Launch the editor for its Request Handler property by clicking the ellipsis button.**



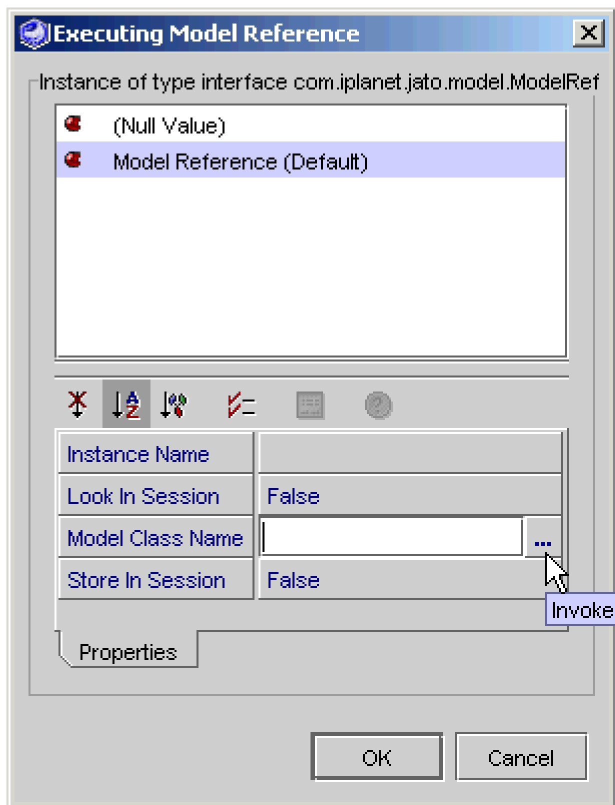
3. **Select the *Execute Model and Goto Page Command* option.**

4. **Set the Name to *searchAndReloadCommand*.**

   On the Component Properties tab, you need to set all three properties.



5. **For the Executing Model Reference property, launch its editor by clicking the ellipsis button.**



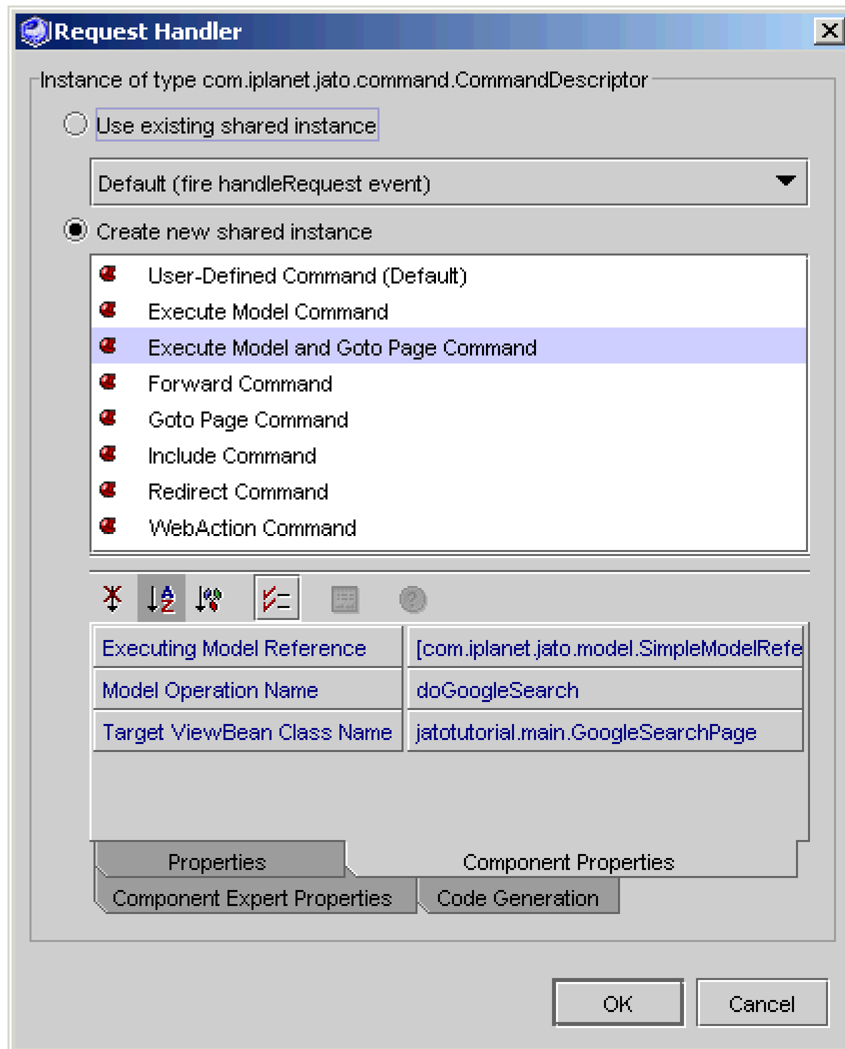
6. **Select the Model Reference (Default) option.**
7. **Launch the Model Class Name property editor.**
8. **Browse and select the GoogleSearchModel.**

9. **When you return to the editor above, and the Model Class Name property is set properly, click OK.**

   The Model Operation Name is a drop down control.

10. **Select *doGoogleSearch* from its option list.**
11. **Launch the *Target ViewBean Class Name* editor.**
12. **Browse and choose the *GoogleSearchPage*.**

    Your Request Handler editor should now appear as shown in the figure below.



13. **Click OK to finish setting the button's Request Handler property.**

    An addition to the GoogleSearchPage's Non-Visual Components node displays: *searchAndReloadCommand*.

Format the JSP Content

Before you test run this page, be sure to format the JSP as you prefer.

1. **Under the GoogleSearchPage, expand the JSPs node and double-click the GoogleSearchPage JSP to open it in the Sun ONE Studio editor.**
2. **Give the fields proper case names.**
3. **Add a label attribute to the checkbox fields and delete the label that has been created automatically.**
4. **Give the page a title, and split it into two sections with a horizontal rule: input fields on top and display-only static text fields on the bottom.**

   The most interesting pieces of the JSP/HTML code are shown in **bold** below:

```
<jato:form name="GoogleSearchPage" method="post">
<h2>Google Search</h2>
<table border=0 cellspacing=2 cellpadding=2 width="100%">
<tr>
  <td align=right valign=middle width="20%"><b>Search for:</b></td>
  <td align=left valign=middle><jato:textField name="queryString"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"><b>Start:</b></td>
  <td align=left valign=middle><jato:textField name="start"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"><b>Max Results:</b></td>
  <td align=left valign=middle><jato:textField name="max"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"><b>Restrict Search:</b></td>
<td align=left valign=middle><jato:textField name="restrict"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"><b>Filter:</b></td>
  <td align=left valign=middle><jato:checkbox name="filter" label=
"Filter?"/></td>
</tr>
<tr>
  <td align=right valign=middle width="20%"></td>
  <td align=left valign=middle><jato:checkbox name="safeSearch" label="Safe
Search?"/></td>
</tr>
</table>
<jato:button name="search"/>
<hr>
Search Time: <jato:text name="searchTime"/>
Results <jato:text name="startIndex"/>
 to <jato:text name="endIndex"/>
 of <jato:text name="estTotal"/>
<jato:hidden name="key"/>
</jato:form>
```

CHAPTER **18**

Tutorial—Section 4.3 Test Run the Google Search Page

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

Task 3: Test Run the Google Search Page

Since you have made modifications to a few classes, be sure to compile the application.

1. **Right-click the Application Classes node, and select the Compile All action.**

   If you are running on Sun ONE Application Server, you must Deploy the application when changes are made.

2. **Select the Sun ONE Application Framework Application node (JatoTutorial), and click the Deploy button on the Sun ONE Application Framework toolbar.**

3. **Select the GoogleSearchPage node, and click the Execute Page (Redeploy) button.**

   Using this execute and redeploy option restarts the server to ensure the server picks up all changes and does not use any cached resources.

   A default browser starts the application.

   The results portion of the page initially has zeroes for values.

   The search will return values for those fields.

Caution – If you receive the following exception, you might have forgotten to do step 15 of part 4.2.1 (remove the googleSearchModel from the GoogleSearchPage's Auto Retrieving Models property):

```
com.iplanet.jato.NavigationException: Exception encountered
```

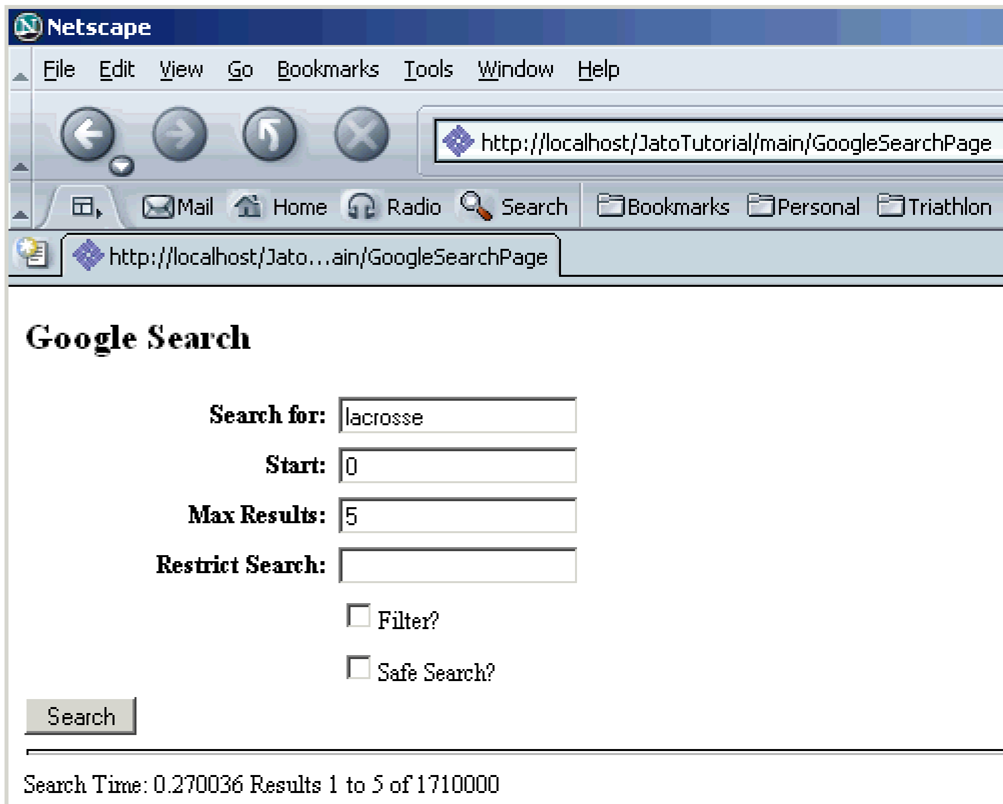
```
during forward
Root cause = [com.iplanet.jato.model.ModelControlException: no
current dataset assigned yet]
```

Try a Search

1. **Enter a search query string.**

   In the figure shown below, the *Search for* is *lacrosse*.

2. **Click the Search button.**



3. **Try other searches to see what results you receive.**

CHAPTER 19

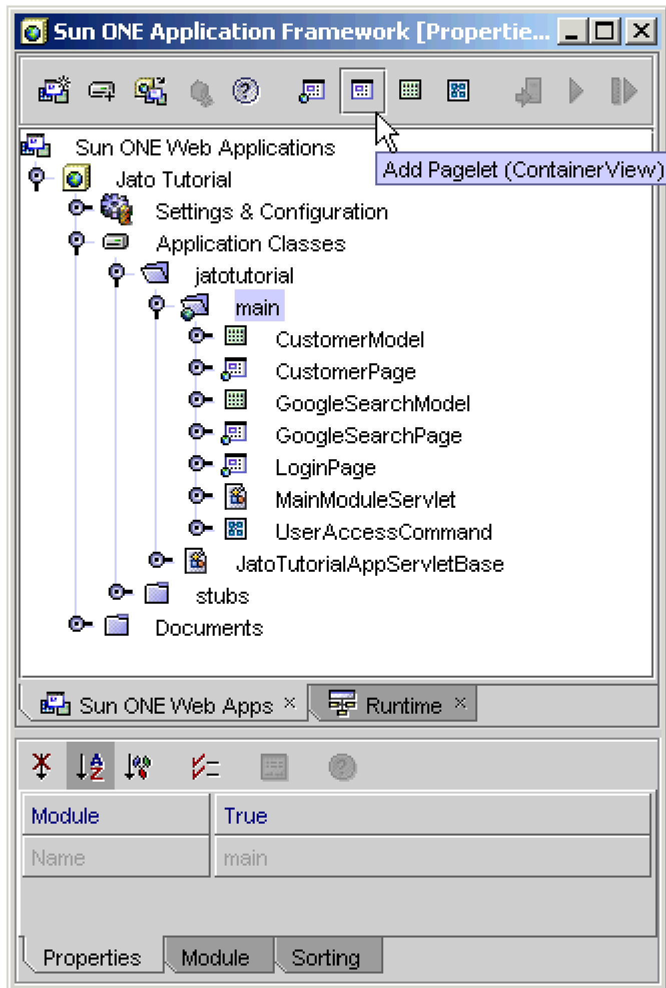
Tutorial—Section 4.4 Add Results Listing to the Google Search Page

This chapter describes how to create a TiledView pagelet component to display a list of results from a Web service model. The TiledView will be added to an existing page component.

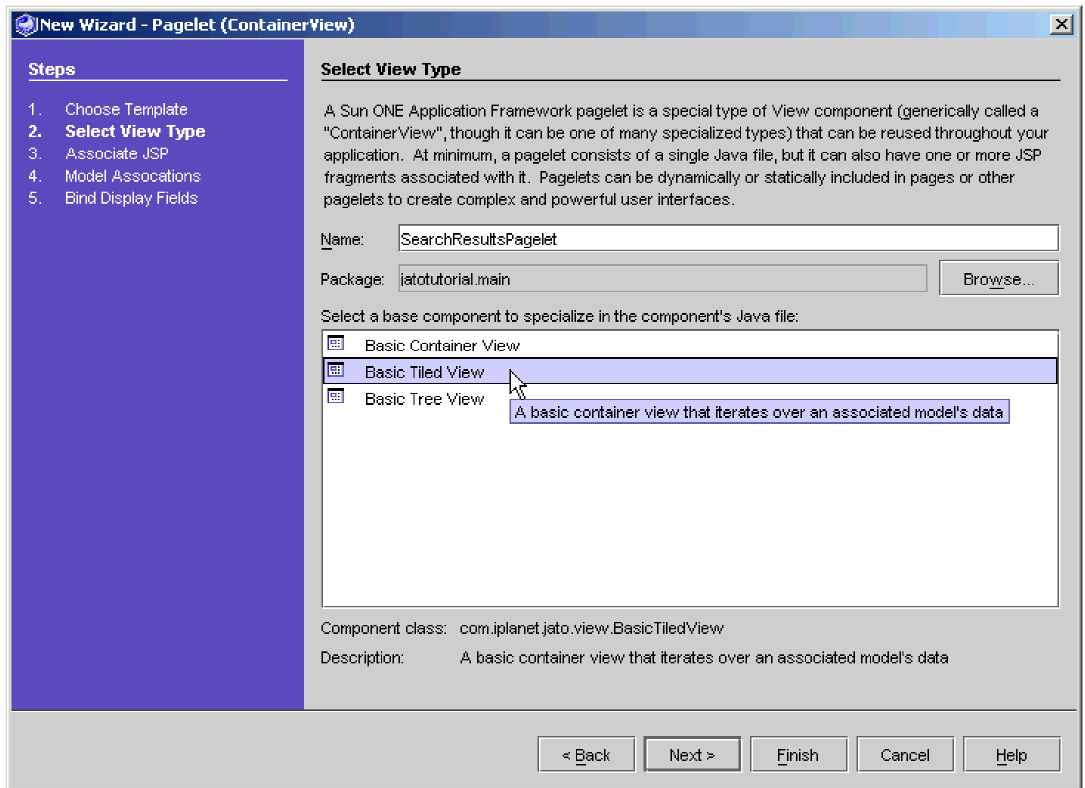
Task 4: Create a TiledView Pagelet

Add a TiledView

1. **Select the main module, and click the Add Pagelet button on the Sun ONE Application Framework toolbar.**

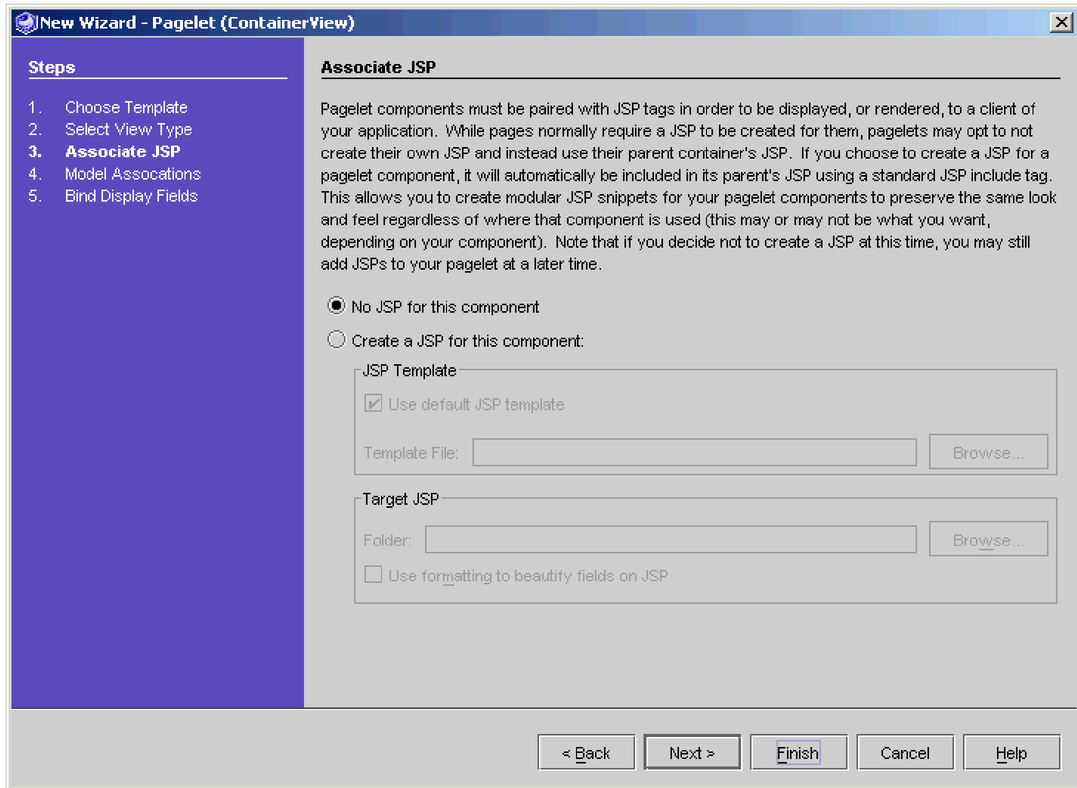


The Select View Type panel displays.



2. **Enter *SearchResultsPagelet* in the *Name* field (to replace <*default*>).**
3. **Select *Basic Tiled View* from the pagelet component type list.**
4. **Click Next.**

   The Associate JSP panel displays.



A matching JSP is not be created for this pagelet component. This pagelet component's JSP tags and content will automatically be added directly to the parent page component's JSP page.

Note – The question is whether to create a JSP for a pagelet component or not. There are pros and cons to both possibilities. The deciding factor is how the pagelet component will be reused on the JSP side. If the pagelet is meant to be rendered the same regardless of what page (or another pagelet) parents it, then you should create a JSP for the pagelet. This single JSP pagelet file will be included (JSP file include directive) in every parenting page and pagelet JSP that requires it. Therefore, any change made to the JSP pagelet file will be reflected wherever it was included.

However, if the pagelet requires the flexibility of being rendered in a different way for various parenting JSP pages and pagelets, the JSP pagelet content must be inlined and customized in each of its parenting JSP page or pagelet files.

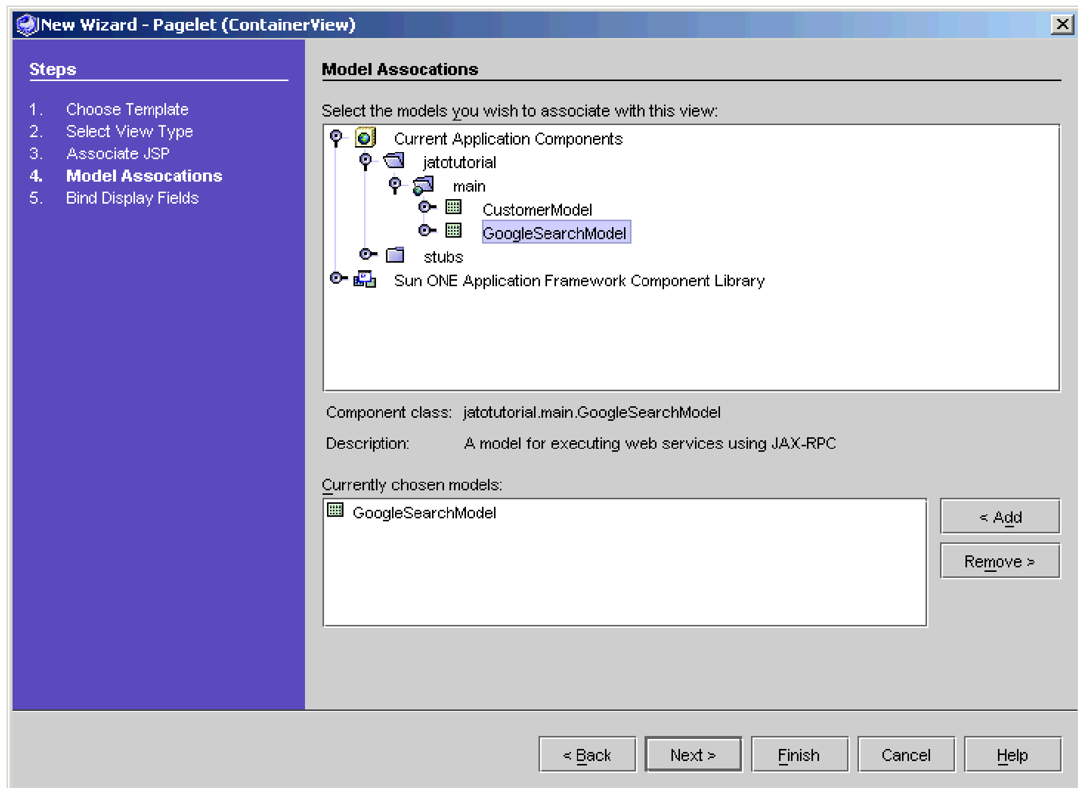
The nuances of these techniques might not be clear to you right away, but do not be concerned. As you become more skilled with JSPs and the Sun ONE Application Framework, you will begin to fully understand the flexibility and power of the reusability of the Sun ONE Application Framework page and pagelet components.

For more information, read the notes that are included on this wizard panel, and refer to the *Sun ONE Application Framework Developer's Guide* and the *Sun ONE Application Framework Component Developer's* guide.

---

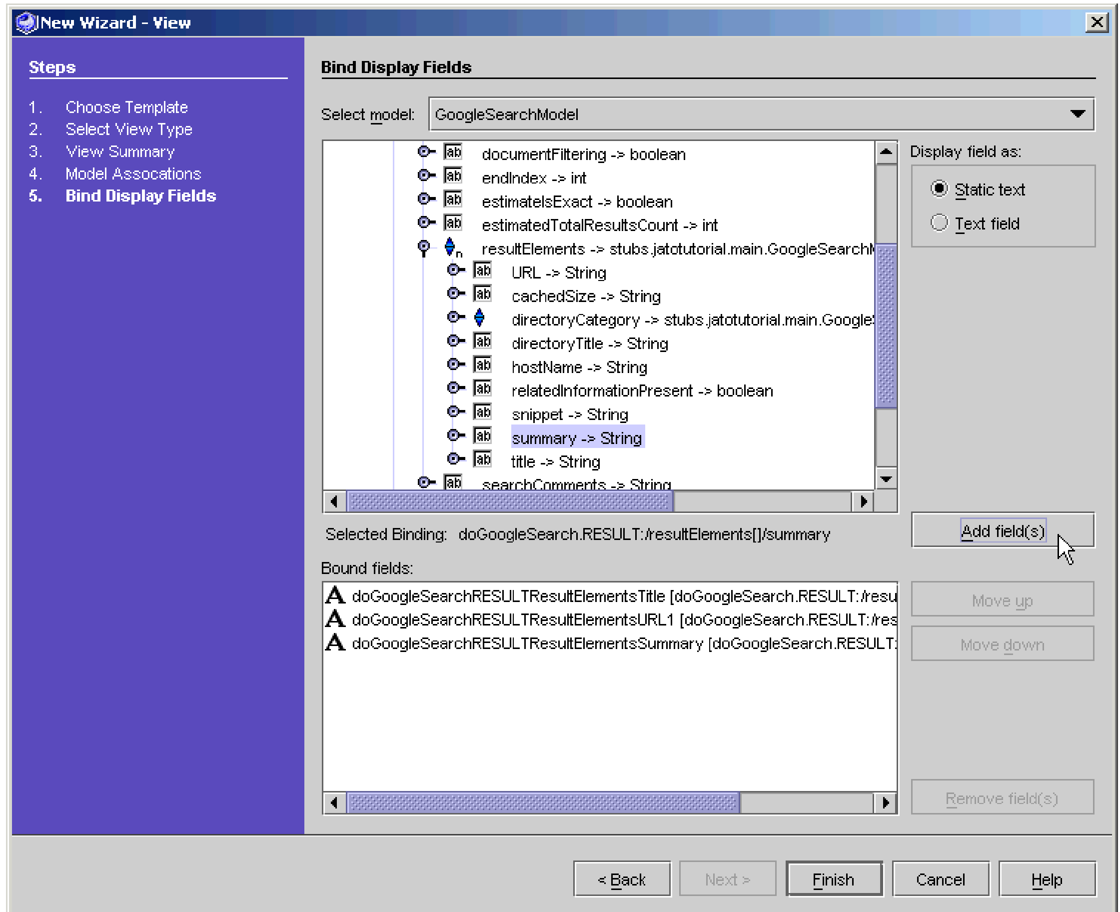
5. **Click Next.**

   The Model Associations panel displays.



6. **Expand Current Application Components to expose** `jatotutorial/main`**.**
7. **Select *GoogleSearchModel*.**
8. **Click Add.**
9. **Click Next.**

   The Bind Display Fields panel displays.



10. **Expand the *doGoogleSearch* node.**

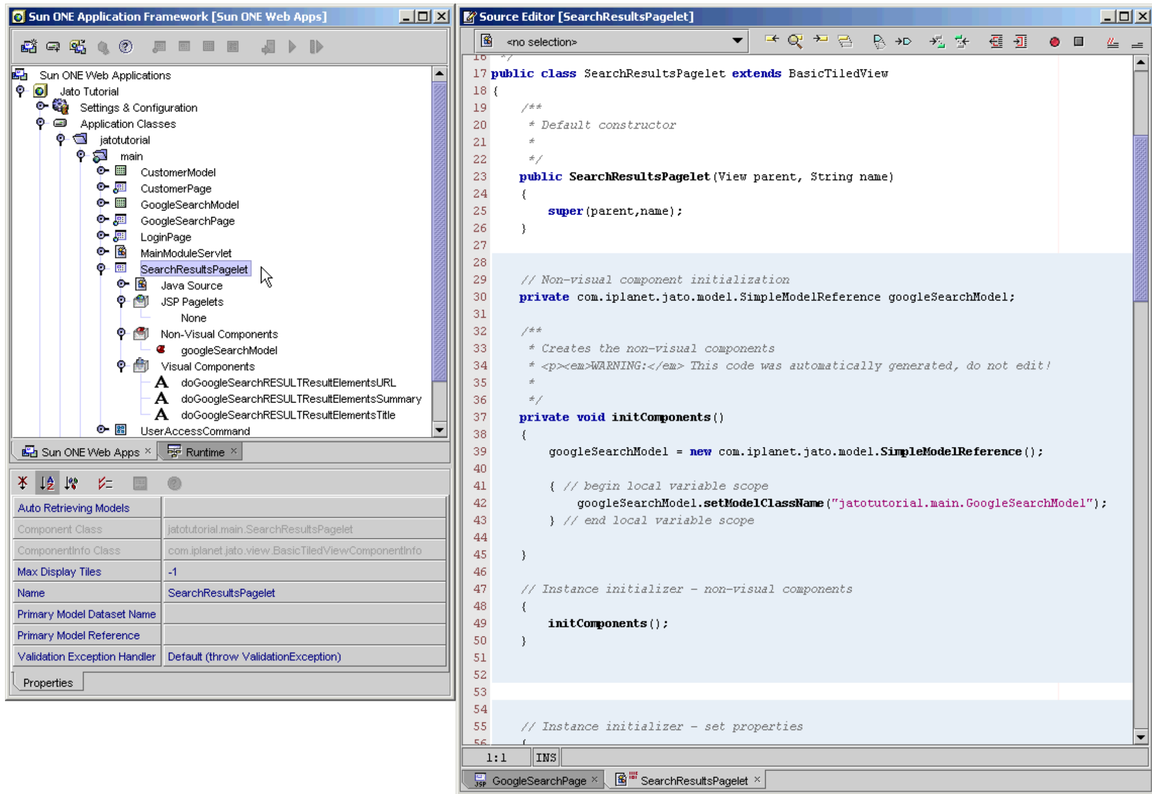
11. **Expand the *Returns* node.**

12. **Expand the *resultElements* node.**

13. **Add the following three return parameters as Static text fields:**
    - title
    - URL
    - summary

14. **Click Finish.**

    You have created the SearchResultsPagelet TiledView with three fields that are bound to some return parameters in the GoogleSearchModel.



15. **Rename the fields to have shorter, simpler names.**

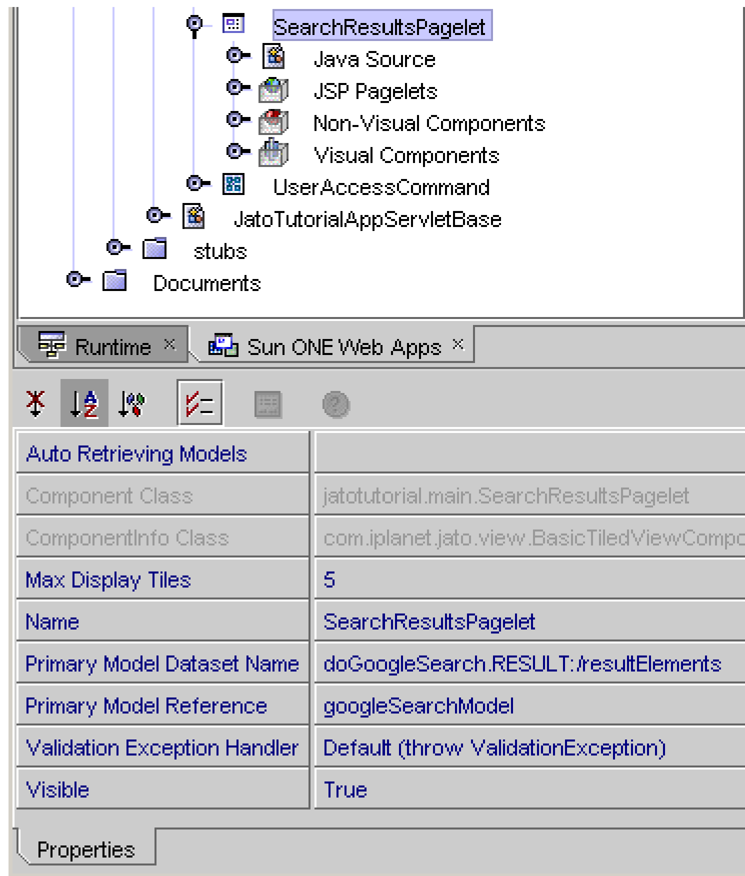
    The following table shows the longer field names in the left column, and the shorter names in the right column.

doGoogleSearchRESULTS**Title**	`to title`
doGoogleSearchRESULTS**URL**	`to url`
doGoogleSearchRESULTS**Summary**	`to summary`

Configure the TiledView Pagelet Component

You need to set three properties for a TiledView pagelet component.

Properties are filled in with the necessary values in the figure shown below.



1. **Select the SearchResultsPagelet TiledView.**
2. **In the property sheet, set the Primary Model Reference by selecting** ***googleSearchModel*** **from the drop down list.**

   The primary model is the model that controls the iteration of the TiledView when it is being displayed.
3. **Set the** ***Max Display Tiles*** **to 5.**

   This will limit the number of displayed results to 5 items.

   A value of -1 (the default) means retrieve/display all possible results.
4. **Set the** ***Primary Model Dataset Name*** **property to be** ***doGoogleSearch.RESULT:/resultElements*****.**

Getting the Correct Primary Model Dataset Name

A TiledView requires a primary model of type *DatasetModel* so that the view will have a domain for the tiles. In the case that the primary model is a *MultiDatasetModel*, you might optionally specify the *Primary Model Dataset Name* so that the TiledView will automatically set the *CurrentDatasetName* on the MultiDatasetModel in both the display and submit cycle.

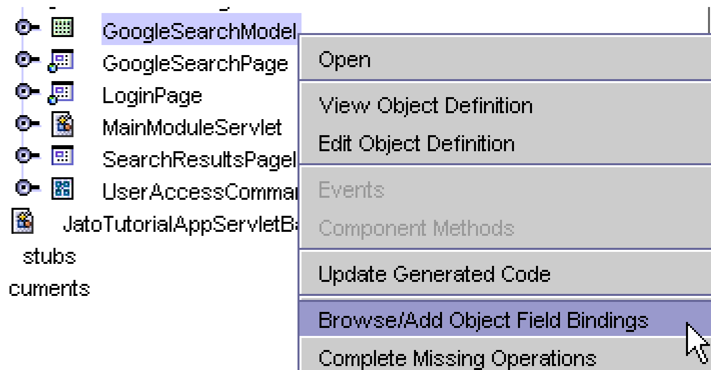
It is not imperative that you fully understand this concept, yet. To put it into simpler terms, a Web service model can have more than one result set. The Primary Model Dataset Name property just lets you specify which one to use, by default, for a particular TiledView.

The Primary Model Dataset Name value is provided for you in this tutorial, but how would you know what this value should be if you were to do this on your own? If you know Web services well enough, you probably know this answer without a problem. You worst enemy would be a typo causing some nasty runtime exceptions.

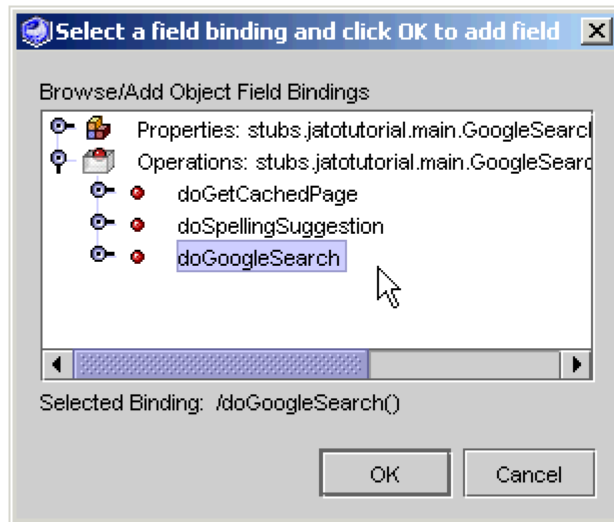
Currently, the Application Framework tools do not have a direct way to set this value by browsing the Web service, and selecting a *key path*. But there is a one-off Web service browsing technique for getting to this value so that you can copy it, then paste it into the property.

Start by selecting the GoogleSearchModel in your application.

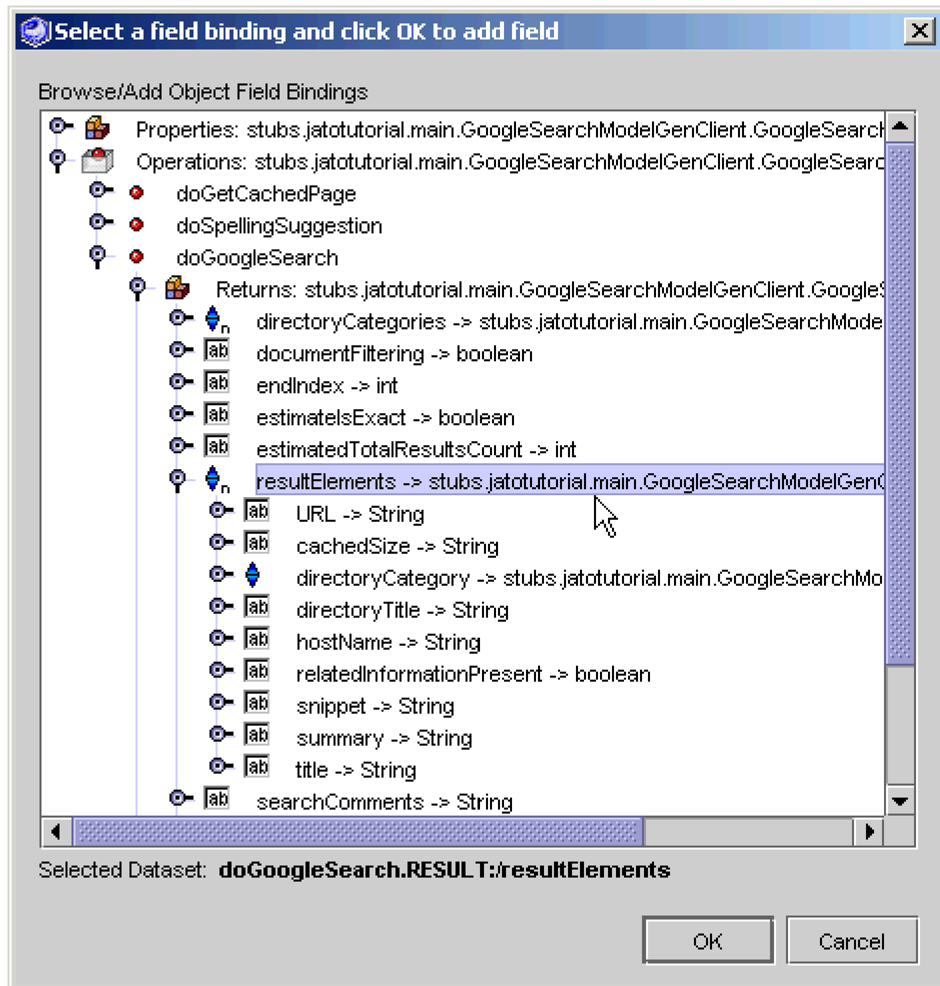
1. **Right click the model, and select** `Browse/Add Object Field Bindings`**.**



The Web Service Field Bindings editor is launched.



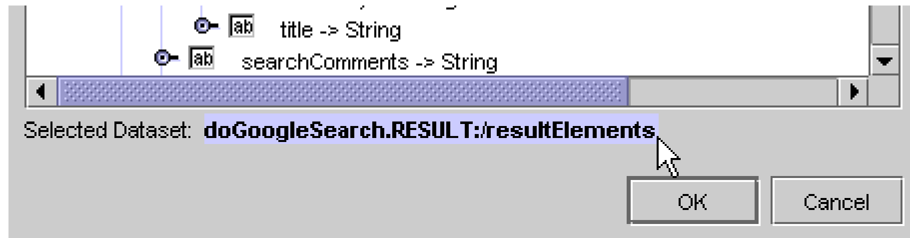
2. **Navigate down the operation path where the fields in the TiledView are bound.**



3. **Select the parent node of the fields:** ***resultElements***

   Notice the bold-faced Selected Dataset value at the bottom of this editor: *doGoogleSearch.RESULT:/resultElements*.

   Although it does not appear to be selectable, you can use your mouse to click/drag select it.



4. **Highlight it and press Ctrl-C to copy the value to the buffer.**
5. **You can now Cancel out of this editor.**

   Do not click OK, as doing so adds a field to your Web service model. Although there is no harm in this, you do not need the field.
6. **Select the SearchResultsPagelet TiledView node, and paste the value into its Primary Model Dataset Name property.**

Add the Pagelet to a Page

A pagelet cannot display without the help of a *root view*. A page (a ViewBean) is a root view. A root view is a container view that can contain other views, but cannot be contained by another container view. All view hierarchies must have a root view. How many levels of views below the root view is completely arbitrary and up to the developer.

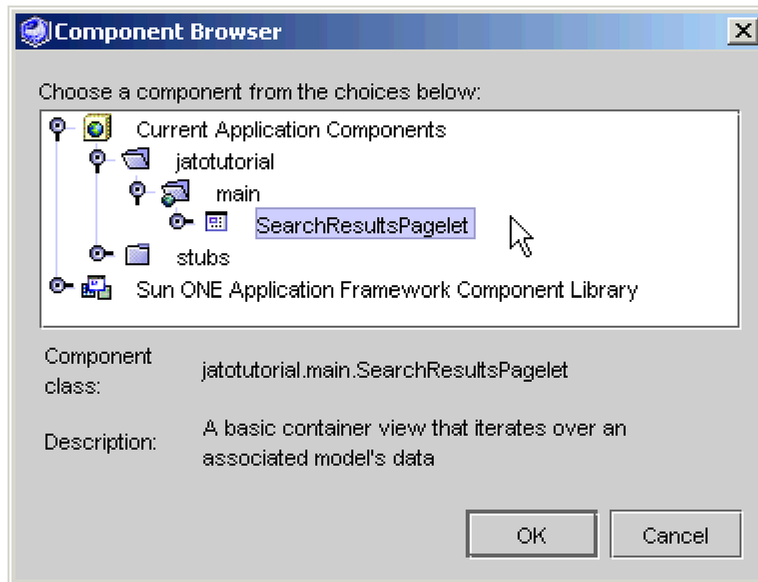
This is much like PC filesytems with drives and directories. A drive (analogous to a page) is always at the top of every absolute path (the root), and there are never any drives that are not at the top level of the path.

Directories (analogous to pagelets) must be contained under drives or other directories. These directories can be nested arbitrarily deep under a drive.

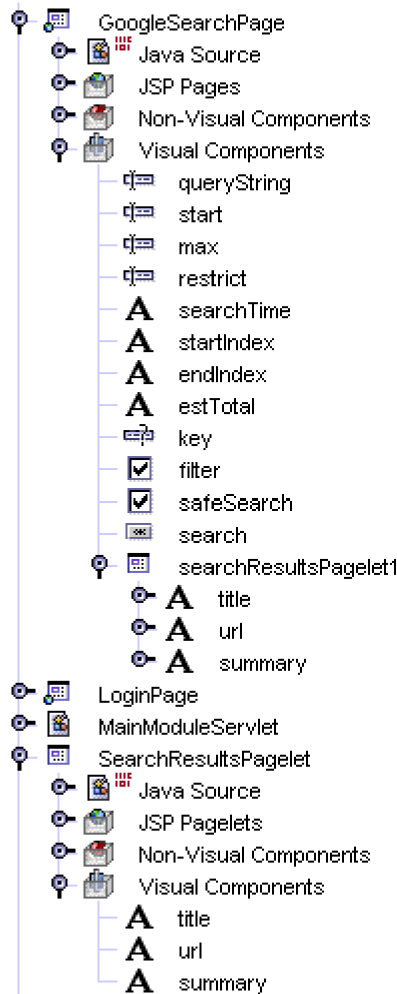
Files (analogous to display fields) must be contained by drives or directories. Files cannot contain other files, directories, or drives.

1. **Expand the GoogleSearchPage node.**
2. **Right-click the GoogleSearchPage's Visual Components node, and select Add Visual Component.**

   This launches the Component Browser.



3. **Expand the Current Application Components node.**
4. **Expand jatotutorial.**
5. **Expand the main node.**
6. **Select the SearchResultsPagelet TiledView component.**
7. **Click OK.**



The SearchResultsPagelet TiledView is added as a visual component under the GoogleSearchPage just like the other visual components. Notice that the pagelet does not have a JSP itself. The pagelet and the other visual components that are contained by the pagelet will have tags added to the parenting page component's JSP page. A pagelet component can be reused by multiple page components, but this is a topic outside the scope of this tutorial.

Formatting the JSP

1. **Open the JSP by double clicking the GoogleSearchPage JSP page node.**

   At the bottom of the JSP, the TiledView pagelet and contained display field tags have been inserted, but without any formatting.

   You can add all the HTML markup you want, but the display field tags that belong to the SearchResultsPagelet TiledView must be nested inside of the `jato:tiledView` tags.

2. **Feel free to be creative with the HTML formatting or use the following content to get you started.**

Note – All of this content is inserted just before the ending JATO form tag (`</jato:form>`), leaving the current content untouched. All of the content between the body tags is being presented here. However, only the **bold** code needs to be added.

```
<body>
<jato:form name="GoogleSearch" method="post">

<jato:hidden name="key"/>
<h1>Google Search</h1>

<h2>Search Criteria</h2>
Search for: <jato:textField name="queryString"/><br>
Start: <jato:textField name="start"/><br>
Max results: <jato:textField name="max"/><br>
<jato:checkbox name="filter" label="Filter?"/><br>
<jato:checkbox name="safeSearch" label="Safe Search?"/><br>
Restrict: <jato:textField name="restrict"/><br>
<jato:button name="search"/>
<hr size="3">

<h2>Results</h2>
Search Time: <jato:text name="searchTime"/><br>
<jato:text name="startIndex"/> to <jato:text name="endIndex"/>
of <jato:text name="estTotal"/>

<jato:tiledView name="searchResultsPagelet1">
<hr size="1">
Title: <jato:text name="title" escape="false"/><br>
<a href="<jato:text name='url'/>" target="_blank">
  <jato:text name="url"/>
</a><br>
Summary: <jato:text name="summary" escape="false"/><br>
<br>
</jato:tiledView>

</jato:form>
</body>
```

A few things above need to be explained:

First, take a look at the *escape* attributes for the *title* and *summary* field tags. The default is *true*, which means escape all special characters. This means that any HTML markup that is returned in the value of this field will be visible to the end user. This Web service places bold tags (<b>) around any words that match the query string that was entered by the end user. So, by specifying `escape=false`, you are telling this tag that you want the markup to be rendered as HTML markup, not displayed to the end user. Experiment with this attribute by making one tag *false* and the other *true* to see the difference.

Second, notice that there are two occurrences of the tag that represents the *url* field. A simple copy/paste of the tag allows you to display the same dynamic data many times. In this case, the first instance of the tag will populate the *href* attribute of the anchor tag (`<a href="..."`).

Notice also that the name attribute of this instance of the *url* field tag uses single quotes. This is because this tag is contained inside of double quotes for the href attribute value. Using double quotes nested inside of double quotes should render perfectly fine at runtime, but many HTML editors will indicate an error condition for this case. Using the single quotes inside the double quotes should remedy these situations.

The second instance of the *url* field's tag is used to display it as the link text to the user.

Everything between the two `jato:tiledView` tags will appear once per row of data that is returned (five times in this example).

CHAPTER **20**

Tutorial—Section 4.5 Test Run the Google Search Page

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

Task 5: Test Run the Google Search Page with Results

Since you have created a new class and made modifications to two other classes, be sure to compile/deploy the application.

1. **Right-click the Application Classes node, and select the Compile All action.**

   If you are running on Sun ONE Application Server, you must Deploy the application when changes are made.

2. **Select the Sun ONE Application Framework Application node (JatoTutorial), and click the Deploy button on the Sun ONE Application Framework toolbar.**

3. **Select the GoogleSearchPage node, and click the Execute Page (Redeploy) button**

   Using this execute and redeploy option restarts the server to ensure the server picks up all changes and does not use any cached resources.

   A default browser starts the application.

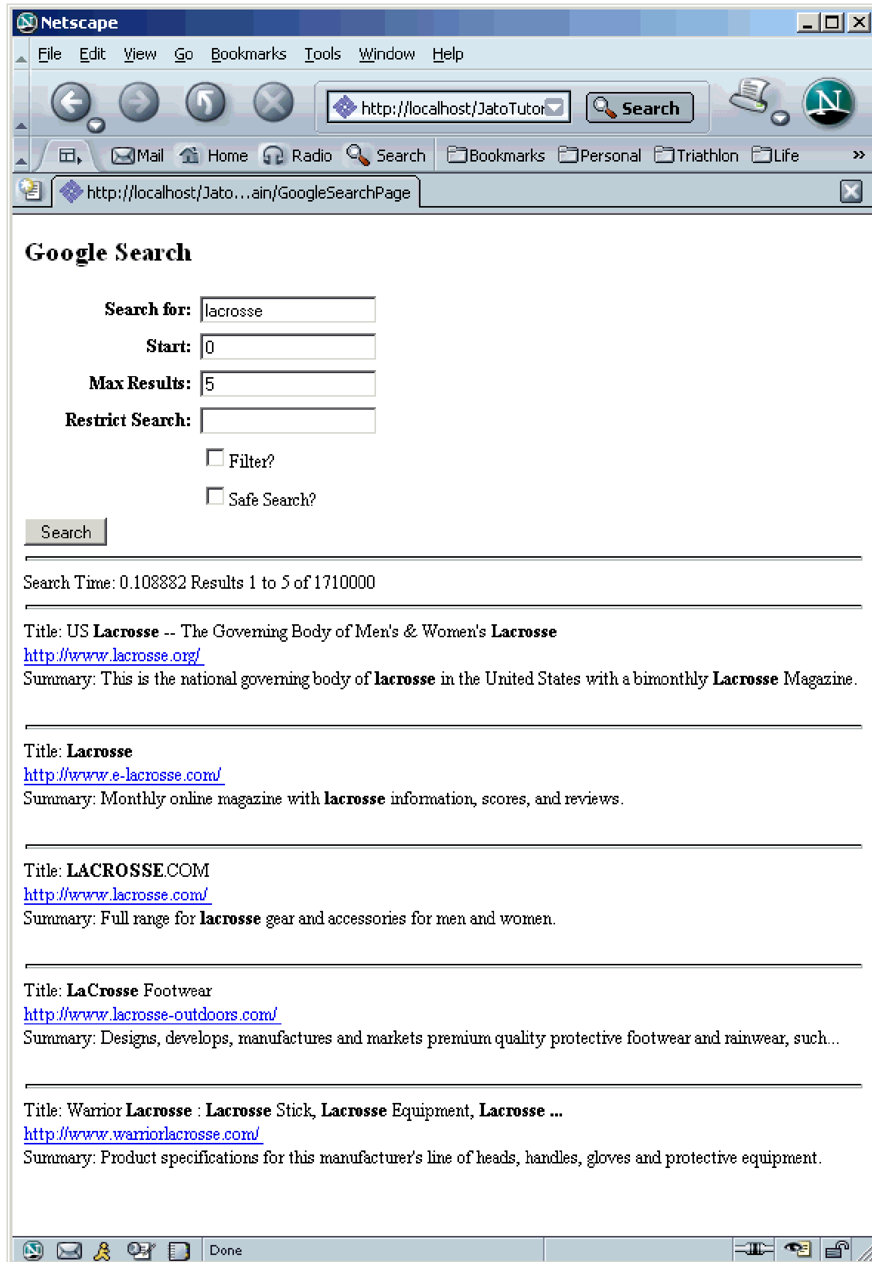
   The results portion of the page initially have zeroes for values. The search returns values for those fields and a list of five links that satisfy the query string.

Try a Search

1. **Enter a search query string.**

   In the figure shown below, the *Search for* is *lacrosse*.

2. **Click the Search button.**



3. **Try other searches to see what results you receive.**

Index

A

Add a ViewBean, 33
Add JDBC Datasource option, 53
Add Model button, 61
Add Page button, 71
Advanced Tip - Modules, 31
Alternative runtime environments, 49
Application Framework application, description, 18
Application Framework applications, how organized, 18
Application Framework JDBC Datasource wizard, 53
Application Framework module, description, 18
application layout, observe, 29
Application Location panel, 26
Application Name folder, select, 45
application pages, link, 51
Application Properties panel, 27
Application Servlet, 30
application servlet not required, 31
application servlet, JatoTutorialAppServletBase, super class, 30
Application, Run the, 99
application, tutorial, about, 19
Associate JSP panel, 35, 73
assumptions for this tutorial, 15
audience for this tutorial, 15
Auto Retrieving Models combo box, 83
Auto Retrieving Models custom editor, 83
Auto Retrieving Models property, 83

B

base directory, default, 27
Basic ViewBean option, select, 35
basics covered in tutorial, 15
Before You Begin, 13 to 14
Bind Display Fields panel, 75
Bound fields list box, 76
Button Command Descriptor, configure, 109
Button Component, add, 78

C

Certification, QA, 14
Choose Template panel, 25
Columns node of the CustomerModel, 68
Command Component, create, 103
Command Descriptor editor, 79
Command Descriptor property, set, 81
comments, send to Sun, 12
Component Palette, 78
Create an Application Wizard, 25
Current Application Components, 75
Customer Page, add hidden field, 84
Customer Page, create, 71
Customer Page, test run, 91
customer record, enable button to update, 78

Customer Update, test, 92
CustomerMode, Create, 61
CustomerModel object, created in main module, 66
CustomerPage, 77
CustomerPage node, 83

D

deployment step required, 46
Design-Time Resources folder, 53
directory location for new application, 25
Display Fields, add to Login page, 37
documentation
  accessing, 11

E

events, module servlet, about, 30
execute Method, add code to, 107
Execute Page (Redeploy) button, 47
Execute Page button, 47

F

Features, Application Framework, 13
feedback, for Sun, 12
first page of application, create, 33
front controller servlet, 31

G

Getting Started, 15, 15 to 20
Google Search Page, create, 135
Google Search Page, test run, 155
Google Search Page, test run, with results, 175
Google Web service software SDK, download, 129

H

handleLoginRequest Method in LoginPage,
  edit, 95
HREF Command Descriptor, configure, 118
HREF tag in Customer JSP, format, 123
HREF, add to a Customer page, 117

I

infrastructure, need to create, 25

J

J2EE application, description, 18
J2EE component, description, 17
J2EE module, description, 17
J2EE Web application, description, 17
J2EE/Sun ONE Application Framework
  Terminology, 17
JDBC datasource panel, 54
JDBC Datasources, 53
JDBC Datasources node, 53
JDBC Datasources, create additional, 53
JDBC SQL Model, create, 61
JDBC URL, makes connection to database, 59
JSP Content, format, 152
JSP, format the, 88
JSP, formatting, 172

L

location, directory, for new application, 25
login name, invalid, 48
Login Page, create, 33
Login Page, link to Customer Page, 95
Login Page, test run, 45, 46
Login, test a successful, 48
Login/Logout Command, test run, 125
LoginPage, 46
LoginPage node, 37

M

main module folder, 61
MainModuleServlet, 31

Manual Code Technique, 148
Model Associations panel, 74
Model Auto Retrieve, making, 82
Model Field Properties tab in property sheet, 68
Model's Key Field(s), mark, 67
Module Properties panel, 28
Module Servlet, 31
module servlet hierarchy, can be customized, 31
module, only one in this application, 31

N

New App Directory field, 27
New datasource name textbox, 55
Non-JNDI Enabled Containers, add connection code for, 68
Non-Visual Components node, 81

O

Operation Name property, 81
organization of Application Framework applications, 18

P

Page Component, add, 135
Pagelet, add to a page, 169
part number, IDE Guide, 12
Point & Click Technique (code-free), 148
PointBase driver, 49
PointBase Network Server (database server), starting, 51
Preface, 9 to 12
Primary Model Dataset Name, getting correct, 166

Q

QA Certification, 14

R

RDBMS database, assumption, 51
related documentation, 11
Rename option, 40
RequestHandler property, 79
Runtime Environments, alternative, 49

S

Sample Database, connect to, 51
Search Button, enable, 147
Search, try a, 156, 176
Section 4.1, Prepare to Create a Web Service Model, 129 to 134
Section 4.2, Create the Google Search Page, 135 to 154
Section 4.3, Test Run the Google Search Page, 155 to 156
Section 4.4, Add Results Listing to the Google Search Page, 157 to 174
Section 4.5, Test Run the Google Search Page, 175 to 177
Select connection combo box, 55
Select Database Tables page, 64
Select Datasource page, 63
Select Model Type panel, 62
Select Table Columns page, 65
Select View Type panel, 72
servlet classes created, 29
Settings & Configuration folder, 53
SQL Database, accessing, 51
SQL-based model, add, 51
SQL-based model, add page to display data, 51
Static Text Field option, 38
success message, 48
successful login, test, 48
Sun documentation, accessing, 11
Sun ONE Application Framework Primary Features, 13
Sun ONE Studio editor display, 37
Sun ONE Web Application, new, 25
Sun technical support, 12

super class, application servlet, JatoTutorialAppServletBase, 30
support, technical, 12

T

technical support, 12
Terminology, J2EE/Sun ONE Application Framework, 17
TiledView Pagelet Component, configure, 164
TiledView Pagelet, create, 157
TiledView, add, 157
Tomcat (and other non-JNDI containers) SQL Connection Preparation, 57
tree, Sun ONE Application, 29
tutorial basics, 15
Tutorial Sections (Links to), 21 to 23
Tutorial, about, 19
tutorial, goal of, 16
Tutorial-Section 1.1, Application Infrastructure, 25 to 31
Tutorial-Section 1.2, Create Login Page, 33 to 44
Tutorial-Section 1.3, Test Run the Login Page, 45 to 49
Tutorial-Section 2.1, Prepare Application to Access SQL Database, 51 to 59
Tutorial-Section 2.2, Create the CustomerModel, 61 to 69
Tutorial-Section 2.3, Create Customer Page, 71 to 88
Tutorial-Section 2.4, Test Run the Customer Page, 91 to 93
Tutorial-Section 2.5, Link Login Page to Customer Page, 95 to 97
Tutorial-Section 2.6, Run Application, 99 to 100
Tutorial-Section 3.1, Create a Command Component, 103 to 115
Tutorial-Section 3.2, Add a Logout Link to the Customer Page, 117 to 123
Tutorial-Section 3.3, Test Run the Login/Logout Command Component, 125 to 127

U

Unsuccessful Login, test, 48
Use formatting to beautify fields on JSP - option, 36
Use formatting to beautify fields on JSP check box, 74
UserAccessCommand Component, create, 103

V

View beans tab, Basic ViewBean option, 35
View Location panel, 34
ViewBean - created, 77
ViewBean, add, 71
Visual Components node, 37
Visual Components, add more to page, 142

W

Web Application, compile, 45
Web Service Model, create, 130
Web Service SDK, download, 129
Web Service User Registration and Downloading, 129
Web Service, register to use, 130
writing Application Framework applications, discussion, 16