



# Web Application Framework Tutorial

---

Sun Java™ Studio Enterprise 7 2004Q4

Sun Microsystems, Inc.  
www.sun.com

Part No. 819-0727-10  
December 2004, Revision A

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

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

U.S. Government Rights - Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties. Sun, Sun Microsystems, the Sun logo and Java 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 U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

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

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

---

Copyright © 2004 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Cette distribution peut comprendre des composants développés par des tierces parties. Sun, Sun Microsystems, le logo Sun et Java 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 portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

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

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

DOCUMENTATION EST FOURNIE "EN L'ETAT" 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.



Adobe PostScript

# Contents

---

- Before You Begin 9**
- 1. Before You Start the Tutorial 15**
  - Primary Features of the Web Application Framework 15
- 2. Getting Started 17**
  - Introduction 17
  - Writing Web Application Framework Applications 18
    - J2EE/Web Application Framework Terminology 19
    - How Web Application Framework Applications Are Organized 20
  - About the Web Application Framework Tutorial 22
- 3. Tutorial Sections (Links to) 23**
  - Sections 1.1—1.3 23
  - Sections 2.1—2.6 24
  - Sections 3.1—3.3 24
  - Sections 4.1—4.5 25
- 4. Tutorial—Section 1.1**
  - Application Infrastructure 27**
  - Task 1: New Sun Java Studio Web Application 27
    - Create an Application Wizard 27

- Application Servlet 33
- Module Servlet 34
- Advanced Tip - Modules 34
  
- 5. Tutorial—Section 1.2**
  - Create Login Page 35**
  - Task 2: Create the Login Page 35
    - Add a ViewBean 35
    - Add Display Fields to the Login Page 39
    - Add Code to the Login Button 44
  
- 6. Tutorial—Section 1.3**
  - Test Run the Login Page 47**
  - Task 3: Test Run the Login Page 47
    - Compile the Web Application 47
    - Test Run the Login Page 48
    - Test a Successful Login 49
    - Test an Unsuccessful Login 49
    - Alternative Runtime Environments 50
  
- 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 52
    - Tomcat (and other non-JNDI containers) SQL Connection Preparation 55
  
- 8. Tutorial—Section 2.2**
  - Create the CustomerModel 57**
  - Task 2: Create the CustomerModel 57
    - Create a JDBC™ SQL Model 57
    - Mark the Model's Key Field(s) 62

Add Connection Code for Non-JNDI Enabled Containers 63

**9. Tutorial—Section 2.3**

**Create Customer Page 65**

Task 3: Create the Customer Page 65

Add a ViewBean 65

Add a Button Component 73

Making a Model Auto Update 75

Add a Hidden Field to the Customer Page 77

Format the JSP 80

**10. Tutorial—Section 2.4**

**Test Run the Customer Page 83**

Task 4: Test Run the Customer Page 83

Test a Customer Update 84

**11. Tutorial—Section 2.5**

**Link Login Page to Customer Page 85**

Task 5: Link the Login Page to the Customer Page 85

Edit the handleLoginRequest Method in LoginPage 85

**12. Tutorial—Section 2.6**

**Run Application 89**

Task 6: Run the Application 89

**13. Tutorial—Section 3.1**

**Create a Command Component 91**

Task 1: Create a Command Component 91

Create the UserAccessCommand Component 91

Add Code to the execute Method 94

Configure a Button's Command Descriptor 96

- 14. **Tutorial—Section 3.2**
  - Add a Logout Link to the Customer Page 99**
  - Task 2: Add an HREF to a Customer Page 99
    - Configure an HREF's Command Descriptor 100
    - Format the HREF tag in the Customer JSP 102
  
- 15. **Tutorial—Section 3.3**
  - Test Run the Login/Logout Command Component 105**
  - Task 3: Test Run the Login/Logout Command 105
  
- 16. **Tutorial—Section 4.1**
  - Prepare to Create a Web Service Model 107**
  - Task 1: Web Service User Registration and Downloading 107
    - Download the Web Service SDK 107
    - Register to Use the Web Service 108
    - Create the Web Service Model 108
  
- 17. **Tutorial—Section 4.2**
  - Create the Google Search Page 113**
  - Task 2: Create the Google Search Page 113
    - Add a Page Component 113
    - Add More Visual Components to the Page 119
    - Enable the Search Button 124
    - Manual Code Technique 125
    - Point & Click Technique (code-free) 125
    - Format the JSP Content 130
  
- 18. **Tutorial—Section 4.3**
  - Test Run the Google Search Page 133**
  - Task 3: Test Run the Google Search Page 133
    - Try a Search 134

<b>19. Tutorial—Section 4.4</b>	
<b>Add Results Listing to the Google Search Page</b>	<b>135</b>
Task 4: Create a TiledView Pagelet	135
Add a TiledView	135
Configure the TiledView Pagelet Component	142
Getting the Correct Primary Model Dataset Name	142
Add the Pagelet to a Page	144
Formatting the JSP	147
Formatting the JSP and Pagelet JSP Fragment (alternative)	149
<b>20. Tutorial—Section 4.5</b>	
<b>Test Run the Google Search Page</b>	<b>153</b>
Task 5: Test Run the Google Search Page with Results	153
Try a Search	153
<b>Index</b>	<b>155</b>





# Before You Begin

---

The *Web Application Framework Tutorial* introduces developers to the mechanics and techniques used to build Web applications using the Web Application Framework tools.

It is intended for developers who are at least somewhat familiar with building web applications using existing Java 2 Platform, Enterprise Edition (J2EE™ platform) web technologies, but new to building web applications with the Web Application Framework.

---

## Before You Read This Book

This tutorial helps the developer become more familiar with using the Web Application Framework tool within the Sun Java Studio Enterprise 7 developer environment (hereafter referred to as the IDE).

Before starting, you should be familiar with concepts used in building web applications using existing J2EE web technologies, such as servlets and JavaServer Pages™ (JSP™ pages).

The following resources can provide additional information :

- *Java 2 Platform, Enterprise Edition Specification*  
<http://java.sun.com/j2ee/download.html#platformspec>
- *The J2EE Tutorial*  
<http://java.sun.com/j2ee/tutorial>
- *Java Servlet Specification Version 2.3*  
<http://java.sun.com/products/servlet/download.html#specs>
- *JavaServer Pages Specification Version 1.2*  
<http://java.sun.com/products/jsp/download.html#specs>

---

**Note** – Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

---

## How This Book Is Organized

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

- [Chapter 1, “Before You Start the Tutorial” on page 15.](#)

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

- [Chapter 2, “Getting Started” on page 17.](#)

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

- [Chapter 4, “Tutorial—Section 1.1 Application Infrastructure” on page 27.](#)
- [Chapter 5, “Tutorial—Section 1.2 Create Login Page” on page 35.](#)
- [Chapter 6, “Tutorial—Section 1.3 Test Run the Login Page” on page 47.](#)

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 57.](#)
- [Chapter 9, “Tutorial—Section 2.3 Create Customer Page” on page 65.](#)
- [Chapter 10, “Tutorial—Section 2.4 Test Run the Customer Page” on page 83.](#)
- [Chapter 11, “Tutorial—Section 2.5 Link Login Page to Customer Page” on page 85.](#)
- [Chapter 12, “Tutorial—Section 2.6 Run Application” on page 89.](#)

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 91.
- Chapter 14, “Tutorial—Section 3.2 Add a Logout Link to the Customer Page” on page 99.
- Chapter 15, “Tutorial—Section 3.3 Test Run the Login/Logout Command Component” on page 105.

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 107.
- Chapter 17, “Tutorial—Section 4.2 Create the Google Search Page” on page 113.
- Chapter 18, “Tutorial—Section 4.3 Test Run the Google Search Page” on page 133.
- Chapter 19, “Tutorial—Section 4.4 Add Results Listing to the Google Search Page” on page 135.
- Chapter 20, “Tutorial—Section 4.5 Test Run the Google Search Page” on page 153.

---

## Typographic Conventions

Typeface	Meaning	Examples
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.cvspass</code> file. Use <code>DIR</code> to list all files. <code>Search is complete.</code>
<b>AaBbCc123</b>	What you type, when contrasted with on-screen computer output	<code>&gt; login</code> <code>Password:</code>
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> save your changes.
<code>AaBbCc123</code>	Command-line variable; replace with a real name or value	To delete a file, type <b>DEL</b> <i>filename</i> .

---

## Related Documentation

Java Studio Enterprise documentation includes books and tutorials delivered in Acrobat Reader (PDF) format, release notes, online help, and tutorials delivered in HTML format.

## Documentation Available Online

The documents described in this section are available from the `docs.sun.com`<sup>SM</sup> web site and from the Documentation link from the Sun Java Studio Enterprise Developers Source portal (<http://developers.sun.com/jsenterprise>).

The `docs.sun.com` web site (<http://docs.sun.com>) enables you to read, print, and buy Sun Microsystems manuals through the Internet.

- *Sun Java Studio Enterprise 7 Release Notes* - part no. 819-0905-10  
Describes last-minute release changes and technical notes.
- *Sun Java Studio Enterprise 7 Installation Guide* (PDF format) - part no. 817-7971-10  
Describes how to install the Sun Java Studio Enterprise 7 integrated development environment (IDE) on each supported platform and includes other pertinent information, such as system requirements, upgrade instructions, server information, command-line switches, installed subdirectories, database integration, and information on how to use the Update Center.
- *Building J2EE Applications* - part no. 819-0819-10  
Describes how to assemble EJB modules and web modules into a J2EE application and how to deploy and run a J2EE application.
- Web Application Framework documentation (PDF format)
  - *Web Application Framework Component Author's Guide* - part no. 819-0724-10  
Describes the Web Application Framework component architecture and the process to design, create, and distribute new components.
  - *Web Application Framework Component Reference Guide* - part no. 819-0725-10  
Describes the components available in the Web Application Framework Library.
  - *Web Application Framework Overview* - part no. 819-0726-10  
Introduces the Web Application Framework and what it is, how it works, and what sets it apart from other application frameworks.

- *Web Application Framework Tutorial*- part no. 819-0727-10  
Introduces the mechanics and techniques to build a web application using the Web Application Framework tools.
- *Web Application Framework Developer's Guide* - part no. 819-0728-10  
Provides the steps to create and use application components that can be assembled to develop an application using the Web Application Framework and explains how to deploy the application in most J2EE containers.
- *Web Application Framework IDE Guide* - part no. 819-0729-10  
Describes the various parts of the Sun Java Studio Enterprise 7 2004Q4 IDE and emphasizes the use of the visual tools for developing a Web Application Framework application.
- *Web Application Framework Tag Library Reference* - part no. 819-0730-10  
Gives a brief introduction to the Web Application Framework tag library, as well as a comprehensive reference to the tags available within the library.

## Tutorials

Sun Java Studio Enterprise 7 tutorials help you understand the features of the IDE. Each tutorial provides techniques and code samples that you can use or modify in developing more substantial applications. All tutorials illustrate deployment with Sun Java System Application Server.

All tutorials are available from the Tutorials and Code Camps link off the Developers Source portal, which you can access from within the IDE by choosing Help > Examples and Tutorials.

- **QuickStart guides** provide an introduction to the Sun Java Studio IDE. Start with a QuickStart tutorial if you are either new to the Sun Java Studio IDE or want a quick introduction to a particular feature. These tutorials describe how to develop simple web and J2EE applications, generate web services, and how to get started with UML modeling and Refactoring. QuickStarts take minutes to complete.
- **Tutorials** focus on a single feature of the Sun Java Studio IDE. Try these if you are interested in the details of a particular feature. Some tutorials build an application from the ground up, while others build on provided source files, depending on the focus of the example. You can complete a tutorial in an hour or less.
- **Narrated Tutorials** use video to illustrate a feature or technique. Try a narrated tutorial for a visual overview of the IDE or an in-depth presentation of a particular feature. You can complete a narrated tutorial in a few minutes. You can also start and stop a narrated tutorial at any point you wish.

## Online Help

Online help is available in the Sun Java Studio Enterprise 7 IDE. You can open help by pressing the help key (F1 in Microsoft Windows environments, Help key in the Solaris environment), or by choosing Help → Contents. Either action displays a list of help topics and a search facility.

## Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table.

---

Type of Documentation	Format and Location of Accessible Version
Books and tutorials	HTML at <a href="http://docs.sun.com">http://docs.sun.com</a>
Tutorials	HTML at the Examples and Code Camps link from the Developers Source portal at <a href="http://developers.sun.com/jsenterprise">http://developers.sun.com/jsenterprise</a>
Release notes	HTML at <a href="http://docs.sun.com">http://docs.sun.com</a>

---

---

## Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

Please include the book's title (*Web Application Framework Tutorial*) and its part number (819-0727-10) in the subject line of your email.

## Before You Start the Tutorial

---

Welcome to the Web Application Framework, the J2EE web application framework, and Sun Java Studio Enterprise 7 toolset (IDE) for enterprise web application development.

This chapter discusses the [Primary Features of the Web Application Framework](#).

---

## Primary Features of the Web Application Framework

The primary features of the Web 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

The Web 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 Web 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 entry into the J2EE API world

This tutorial shows you how to use the Web Application Framework features to do the following:

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



# Getting Started

---

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

This chapter contains the following topics:

- [Introduction](#)
- [Writing Web Application Framework Applications](#)
- [About the Web Application Framework Tutorial](#)

---

## Introduction

This document introduces developers to the mechanics and techniques used to build Web applications with the Web 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 JSP™ pages), but new to building web applications using the Web 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 Web 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 Web Application Framework application, by extending existing Web Application Framework implementation base classes and manually constructing certain application objects. This is only one possible way to create a Web 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 Web 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 Web Application Framework.

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

---

## Writing Web Application Framework Applications

Writing a Web Application Framework application consists of first laying out an application structure, and incrementally adding Web Application Framework objects to that structure. Although this can be done entirely by hand and from scratch, the task has been simplified by creating the Web Application Framework tools module for the Sun Java Studio Enterprise 7 that assists developers in writing their Web Application Framework applications. With the assistance of these tools, creating a Web Application Framework application becomes a simple process of generating Web Application Framework components using wizards, and customizing them to an application.

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

# J2EE/Web 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.

<b>Term</b>	<b>Description</b>
*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™ (JSP™); there are components including HTML and Applets which are also J2EE components but these are not relevant to the Web 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 of 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 Java System Application Server Standard Edition 7 2004Q2 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 <code>.ear</code> (enterprise archive) filename extension.
Web Application Framework module	Refers to both a logical and physical partition of content and components within a Web Application Framework application (not to be confused with a J2EE module).
Web Application Framework application	In informal terms, a Web Application Framework application is a J2EE web application that has been written using the Web 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 Web Application Framework application is a J2EE web application consisting of one WAR file. In formal terms, a Web Application Framework application is a collection of related Web Application Framework modules, all running in the same servlet context. In this sense, Web Application Framework application refers only to this logical Web 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 Web Application Framework Applications Are Organized

The Web Application Framework provides formal application and module entities. A Web Application Framework application is a base Java package that contains one or more sub-packages (Web 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 Web 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 Web Application Framework application.

Avoid over-categorizing your application into several modules simply because the Web 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 Java System 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 Web 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 Web Application Framework Tutorial

You will now develop a simple application so you can experience using the Web 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.

## Tutorial Sections (Links to)

---

This chapter outlines the sections contained in this *Web 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 Web Application Framework page.

- Section 1.1
  - [Task 1: New Sun Java Studio Web Application](#)
- Section 1.2
  - [Task 2: Create the Login Page](#)
- Section 1.3
  - [Task 3: Test Run the Login Page](#)

---

## Sections 2.1—2.6

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

- Section 2.1
  - [Task 1: Accessing a SQL Database](#)
- Section 2.2
  - [Task 2: Create the CustomerModel](#)
- Section 2.3
  - [Task 3: Create the Customer Page](#)
- Section 2.4
  - [Task 4: Test Run the Customer Page](#)
- Section 2.5
  - [Task 5: Link the Login Page to the Customer Page](#)
- Section 2.6
  - [Task 6: Run the Application](#)

---

## 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](#)



# Tutorial—Section 1.1

## Application Infrastructure

---

This chapter describes how to create the Sun Java Studio Web Application Framework (also known as Web Application Framework, App Framework, SJSAF, and JATO) application infrastructure needed for all subsequent tasks.

---

### Task 1: New Sun Java Studio Web Application

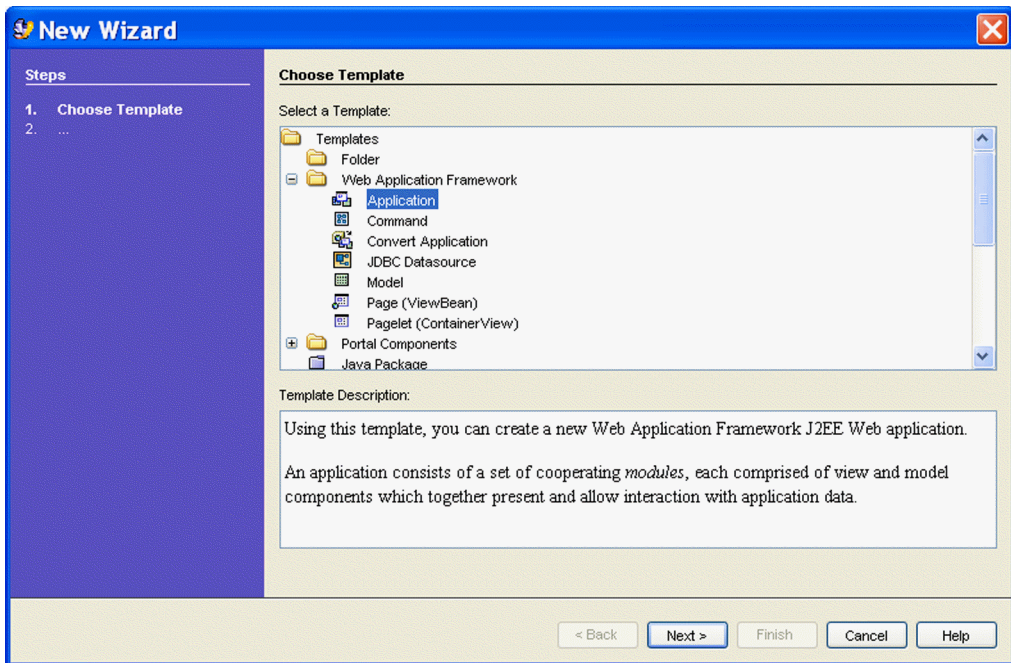
Before developing any pages, you need to create the Web Application Framework application infrastructure (the WAR directory structure and supporting files). This is a onetime requirement for each Web Application Framework application.

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

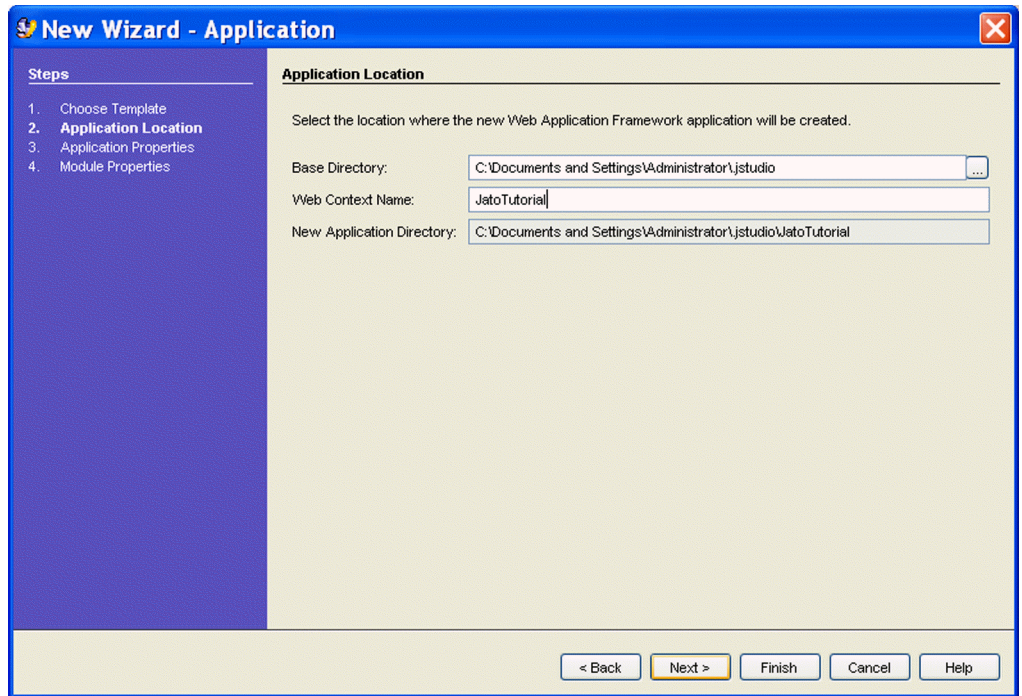
1. Select the Sun Java Studio Enterprise 7 menu option File -> New Web Application Framework Application.

The Choose Template panel displays.



2. Expand the Web Application Framework folder.
3. Select Application.
4. Click Next.

The Application Location panel displays.



The default base directory is your Sun Java 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 Web 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 Web Application Framework application using the Sun Java Studio, so you can put your Web application anywhere you want.

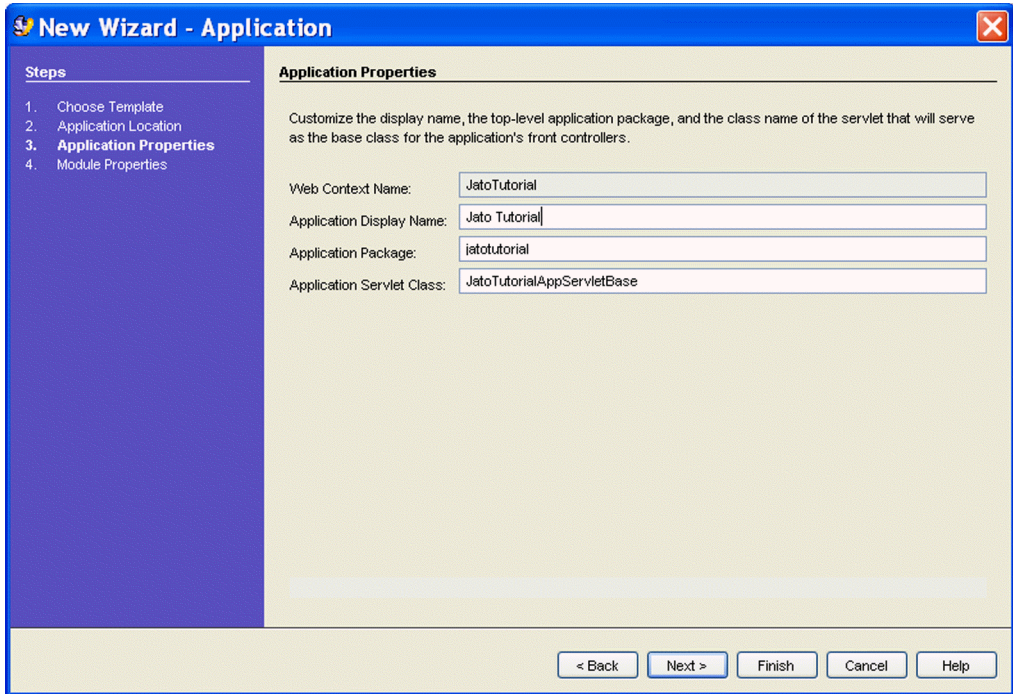
---

**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.

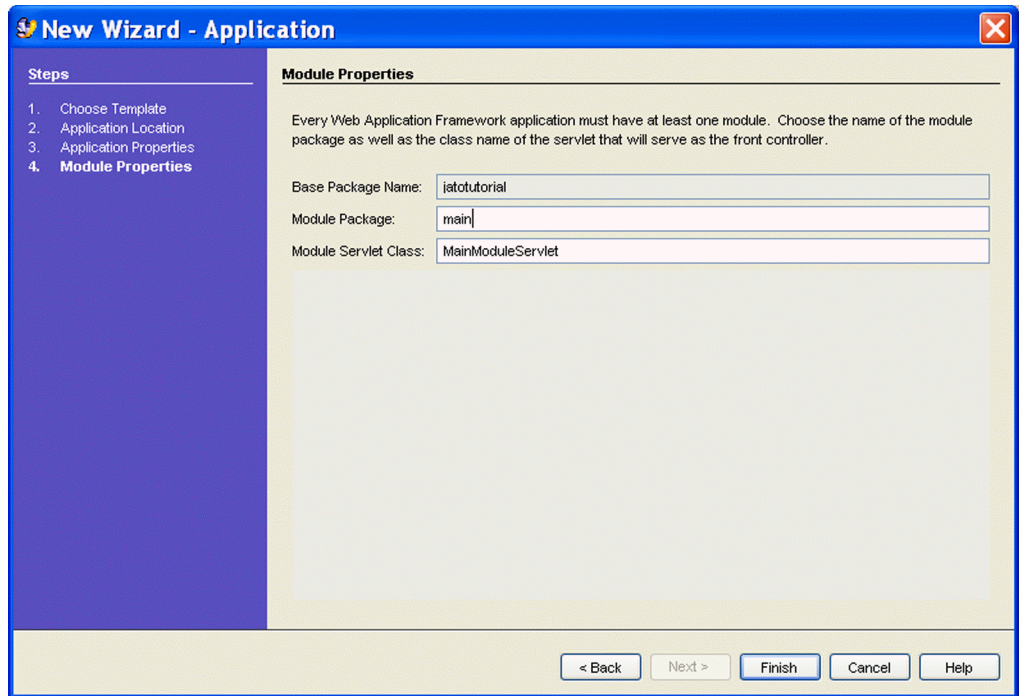


The fields on this panel are populated using the value of the *Web Context Name* field from the previous panel.

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. Close the pop-up dialogs.**

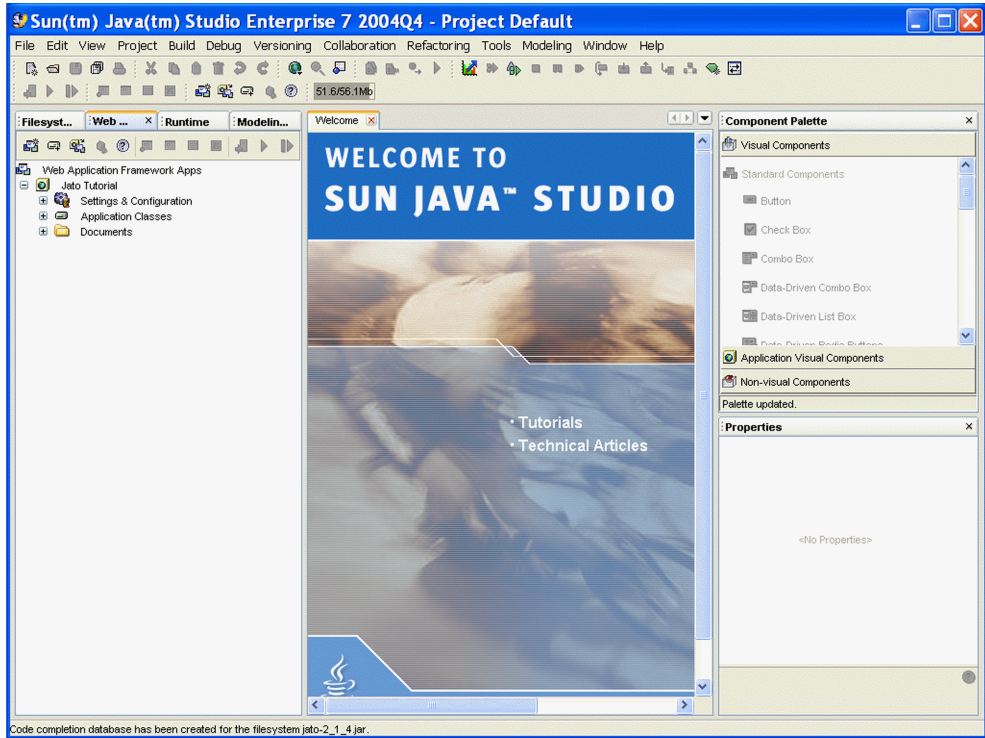
The application is created.

---

**Note** – The processing time depends upon your machine.

---

The new application displays in the Web Application Framework Apps tree in the IDE's Explorer window, which is labeled Web Application Framework Apps.



---

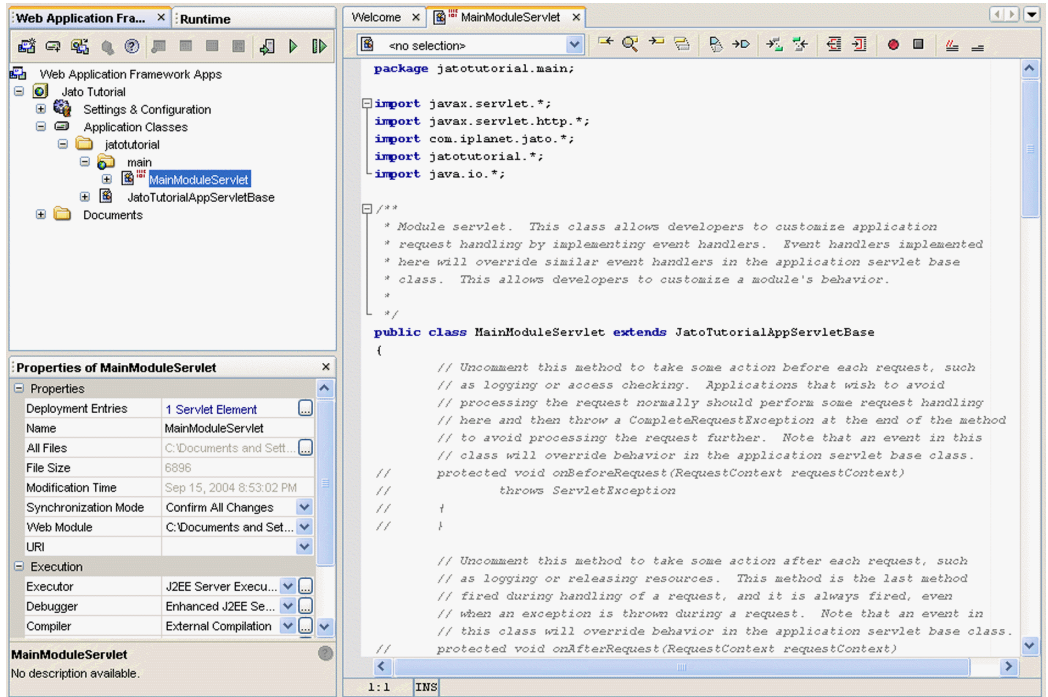
**Note** – The figure above is the result after clicking OK. You may not have the Properties window visible at this time, especially after a new installation of the IDE. In order to toggle the display of windows use the Window menu and select the window you want to show or hide.

---

10. Expand the Application Classes node in the Web Application Framework Apps Explorer tab to see the application layout and observe the code in the two servlet classes that were created.
  - JatoTutorialAppServletBase
  - MainModuleServlet
11. The Properties Window may not be located where you like it. In order to move windows around in the new IDE, select the window title bar (click and hold down left mouse button) and drag to the location you would like.

While you are dragging the IDE will display a red outline of the new window location demonstrating where and how it will look. Windows can be docked to quadrants of the screen or docked as tabbed panels with other windows. The figure below shows the Properties window placed at the lower left under the Explorer view.





## 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 Web 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 Web Application Framework's `com.ipplanet.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 Web Application Framework's `com.ipplanet.jato.ApplicationServletBase` file. Advanced developers can gain some insight on how requests are handled by reviewing the source code in the `com.ipplanet.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 Web 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.

# Tutorial—Section 1.2

## Create Login Page

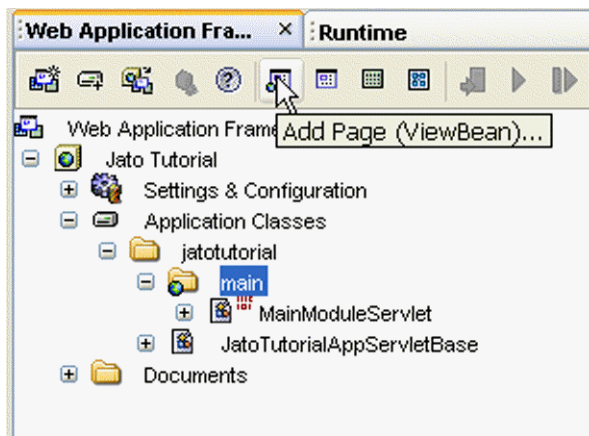
This chapter describes how to add your first Web Application Framework page to the application infrastructure you created.

### Task 2: Create the Login Page

Create the first page of the application.

#### Add a ViewBean

1. Select the main module folder from the Web Application Framework Apps Explorer tab.



2. Click the Add Page (ViewBean) button on the Web Application Framework toolbar

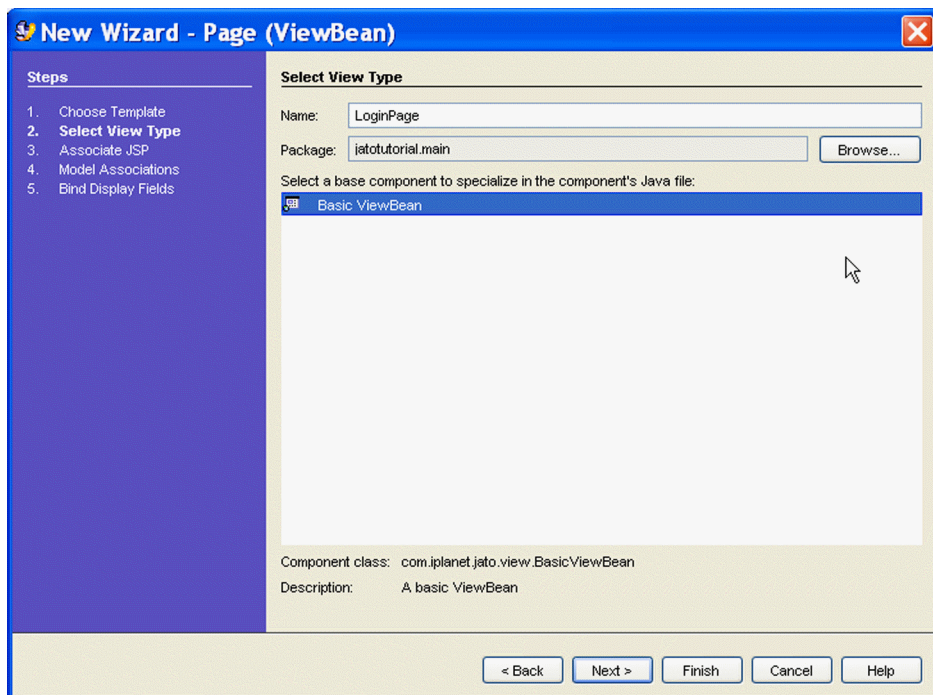
Or:

- a. Select the IDE's menu option File -> New.
- b. Expand Web Application Framework node
- c. Select Page (ViewBean)
- d. Click Next.

Or:

- a. Right-click the main module folder.
- b. Select Add.
- c. Select Page (ViewBean).

The Select View Type panel displays.

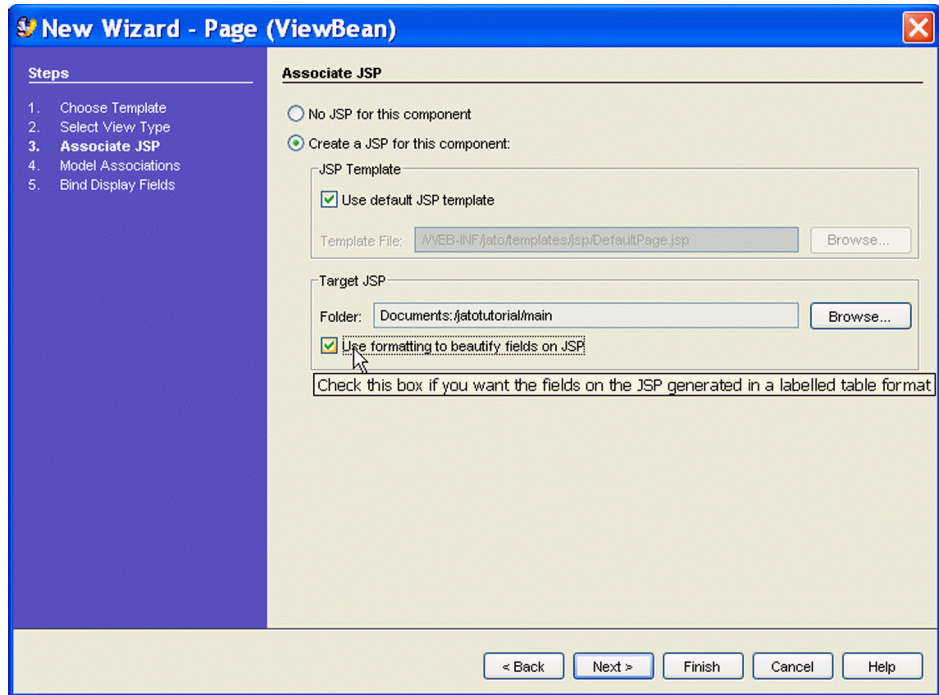


3. Enter LoginPage in the Name field (to replace <default>).

4. In the base components list, select *Basic ViewBean*.

5. Click Next.

The Associate JSP panel displays.



6. Accept all defaults but check the box *Use formatting to beautify fields on JSP* option.

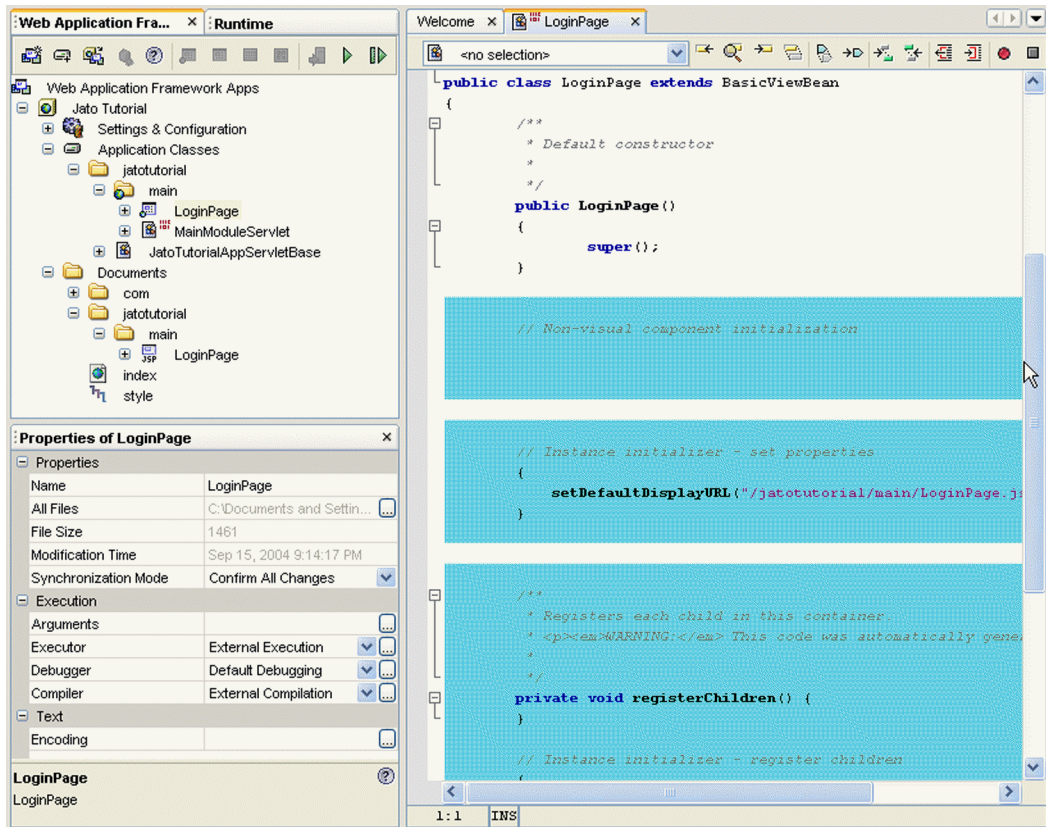
7. Click Finish.

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.

---



## 8. Double-click LoginPage.

The generated source code displays in the IDE's source editor.

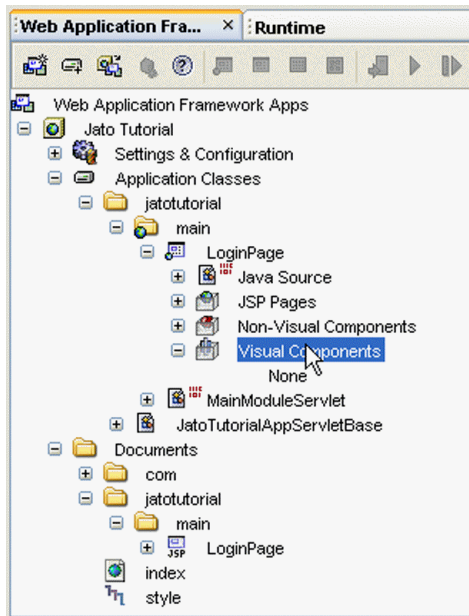
---

**Note** – Because you elected to create 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 (/jatutorial/main). For convenience, a link to the JSPs that use the LoginPage are placed in the JSP Pages node, which is under the LoginPage node.

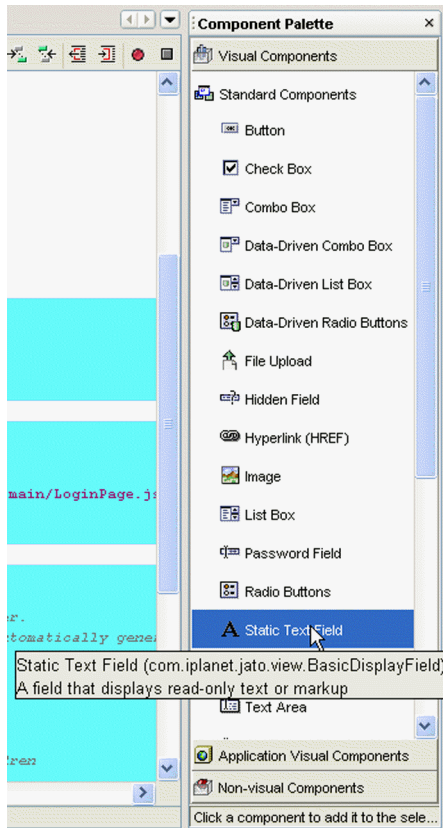
---

# Add Display Fields to the Login Page

1. Expand the `LoginPage` node.
2. Select the `Visual Components` node under the `LoginPage` node.

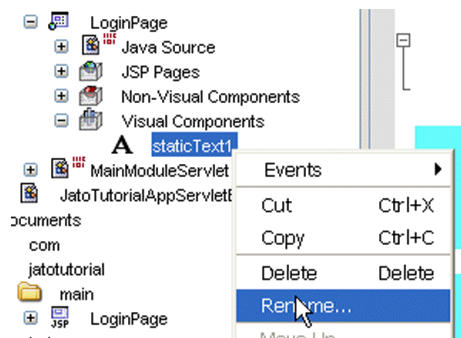


3. In the Web Application Framework Component Palette, click the Static Text Field option.



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

The default name is *staticText1*.



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

5. Select Rename.



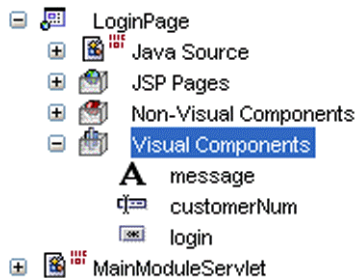
6. Rename the field to *message*.

7. 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.



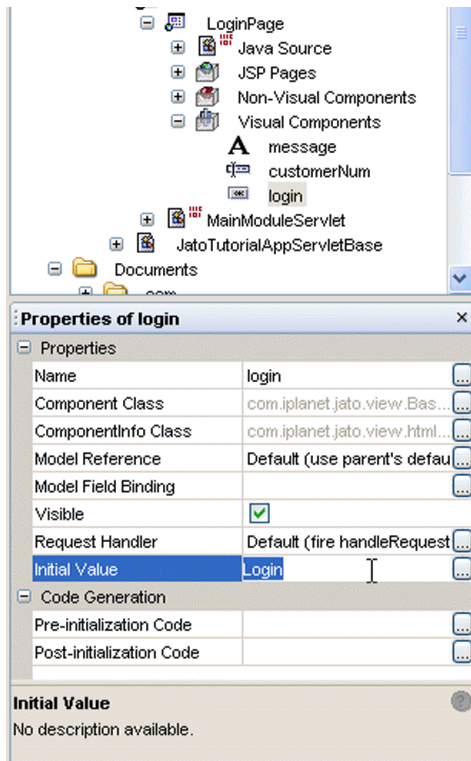
Adding display fields to the Page also adds the appropriate JSP tags for the display fields to the JSPs that are using this Page.

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

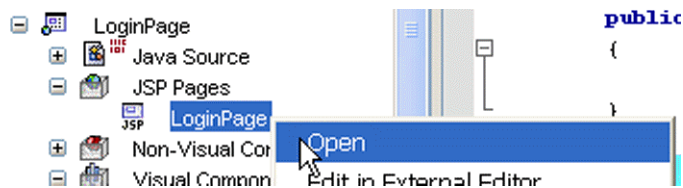
9. Click in the Initial Value property value entry area.

10. Enter the string *Login*.

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



11. 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 IDE's source editor.



12. 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 HTML formatting may have been applied to get you started. Exactly what additional formatting is provided automatically depends on what kinds and conditions of child view components are added to the parent view component. In this case, you will probably want to modify things a bit more because no additional formatting was provided as we had added this simple visual component manually.

---

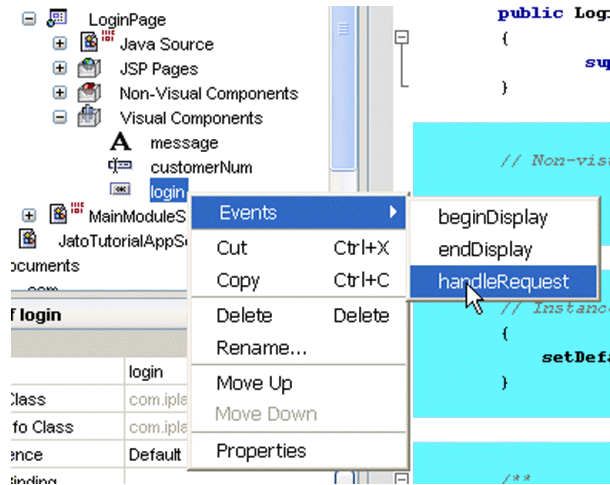
You can edit it directly in the IDE's source editor, or you can use your favorite WYSIWIG HTML editor.

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

1. Right-click the login button.
2. Select Events -> handleRequest



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

3. 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();
}

```



# Tutorial—Section 1.3

## Test Run the Login Page

---

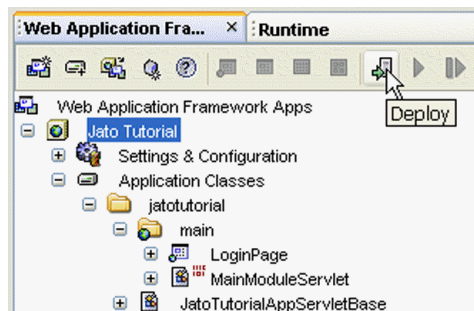
This chapter describes how to run your Web Application Framework application.

---

### Task 3: Test Run the Login Page

#### Compile the Web Application

1. Select the Application Name folder.



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

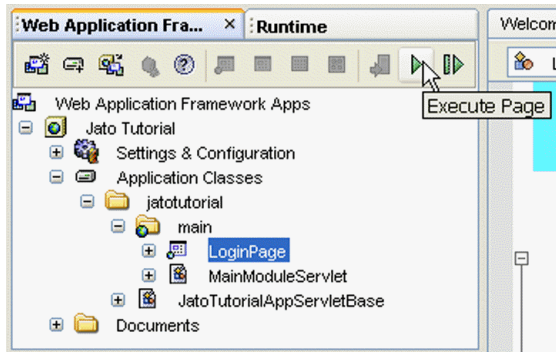
This compiles the entire Web application (those classes that need to be compiled) and deploys it to the Sun Java System Application Server in one step.

If you followed all of the tutorial instructions, the Web application compiles and deploys without error. See the IDE's 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 Java System Application Server (Application Server).

## Test Run the Login Page

1. Select **LoginPage**.



2. Click the **Execute Page** button located on the **Web 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 **Web Application Framework** toolbar) forces the Sun Java System Application Server to reload all resources (for example, JSP pages, classes, and so on). 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.

---

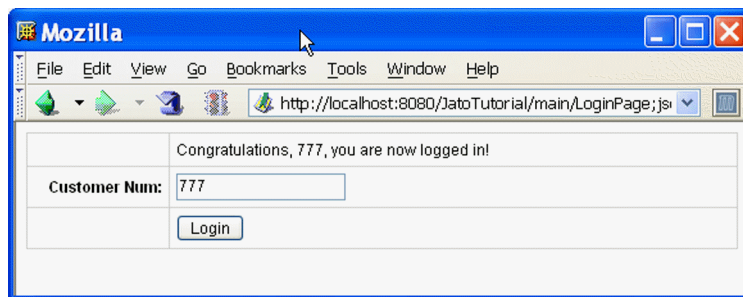
**Caution** – For some web browsers include certain version of Microsoft Internet Explorer, if you press the enter key while in the text field, the form is submitted for you. However, the server does not know which button to address from this submit action. The `<jato:form>` tag provides an attribute `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.

---

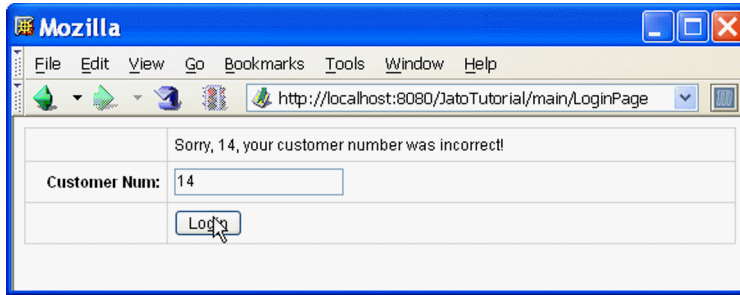
The login page should refresh displaying the success message.



## Test an Unsuccessful Login

1. Enter an invalid login name (for example, 100, 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 your application outside of the IDE, 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 IDE, as follows:

`<studio-install-dir>\AppServer7\pointbase\client_tools\lib\pbclient42RE.jar` for Windows.

`<studio-install-dir>/AppServer7/pointbase/client_tools/lib/pbclient42RE.jar` for Solaris/Linux.

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.

---

# Tutorial—Section 2.1

## Prepare Application to Access SQL Database

---

This chapter describes how to expand the application and prepare the Web 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

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

There is no requirement for an Web 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 steps that follow rely on the IDE's preconfigured Pointbase Database which is auto-started whenever the IDE tries to use the preconfigured JDBC connections to the samples database. Also, the automatically installed Sun Java System Application Server includes preconfigured database connection resources. Each new Web Application Framework application will be preconfigured with a datasource

matching the pre-existing Pointbase samples database connection resources. In this way the new developers can very quickly use Web Application Framework and the Pointbase samples database; no additional configuration is required.

## JDBC Datasources

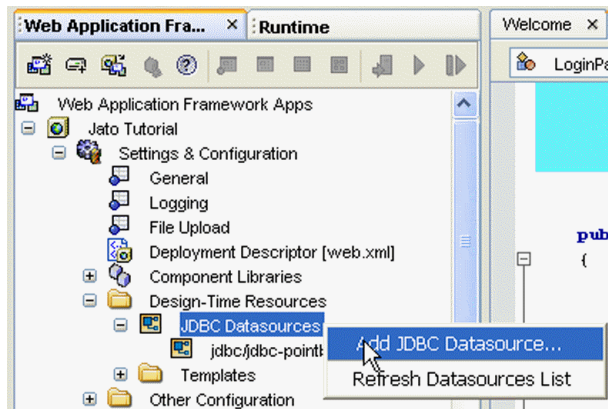
You can create a JDBC Datasource using the Web Application Framework JDBC Datasource wizard.

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

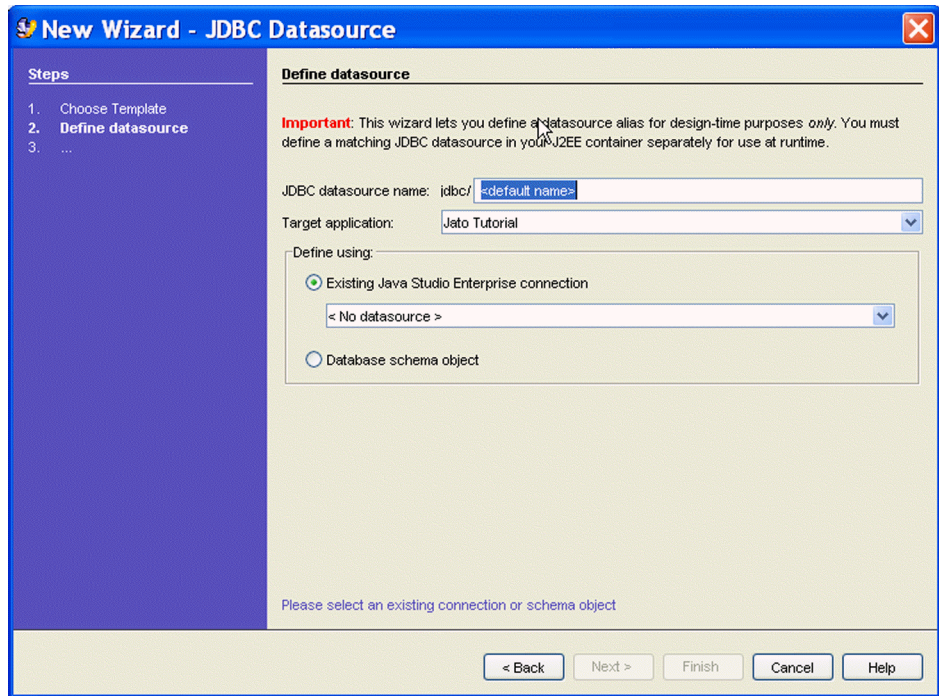
If you need to create additional JDBC Datasources for a different database other than the one used in the tutorial, use the following steps.

(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 Web 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.



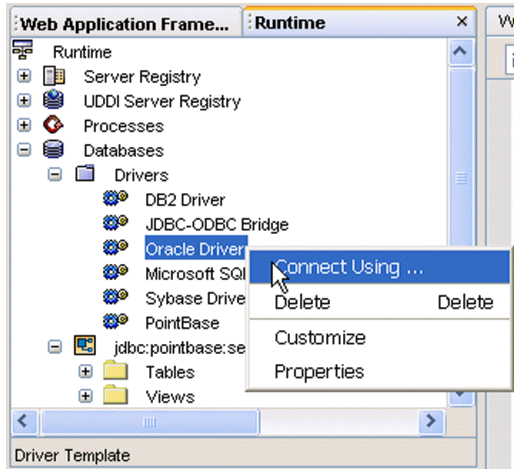
The Define JDBC datasource panel displays.



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

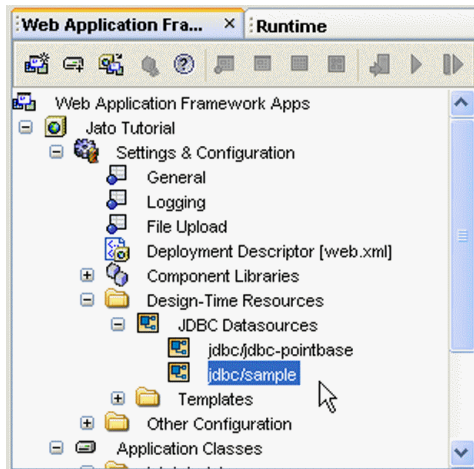
6. In the Select connection combo box, select the appropriate JDBC connection.

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 Web Application Framework tools. You need to select the Runtime window, expand the Databases node, expand the Drivers node, right-click the desired driver node, and select Connect Using. You might need to add a driver for your database before you can add a connection. See the IDE 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 Java System 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 Web 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 Java System Application Server, you need to copy the PointBase client library JAR file (`pbclient42RE.jar`) into your `WEB-INF/lib` directory.

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

`<studio-install-dir>/appserver7/pointbase/client_tools/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 Java System Application Server because the PointBase libraries are already included in its classpath.



## Tutorial—Section 2.2 Create the CustomerModel

---

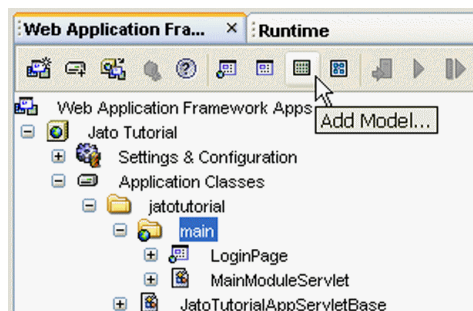
This chapter describes how to create a model to access the RDBMS in the Web Application Framework application.

---

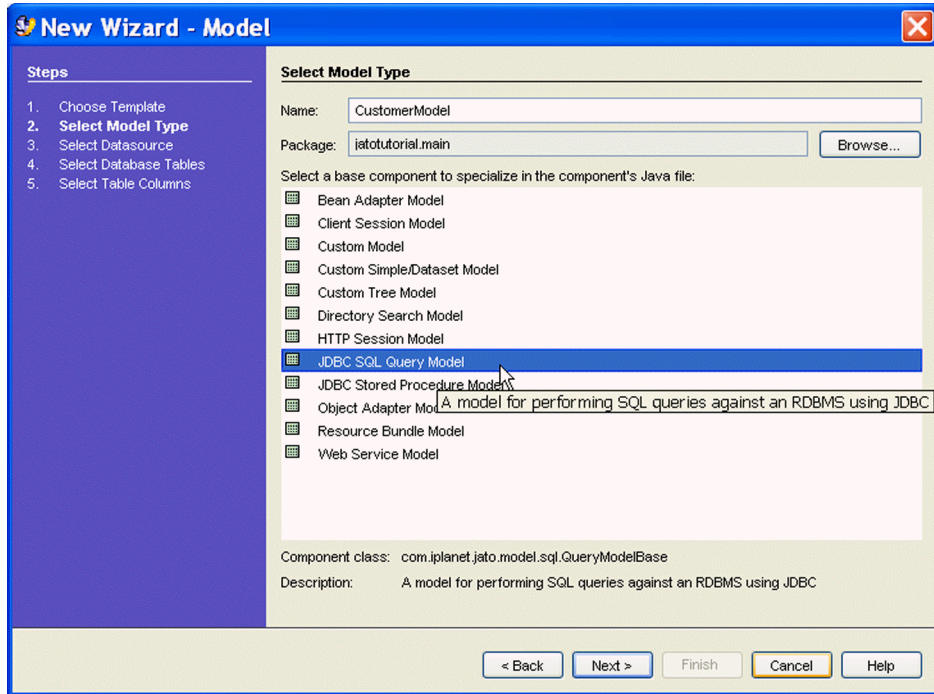
### Task 2: Create the CustomerModel

#### Create a JDBC™ SQL Model

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



The Select Model Type panel displays.



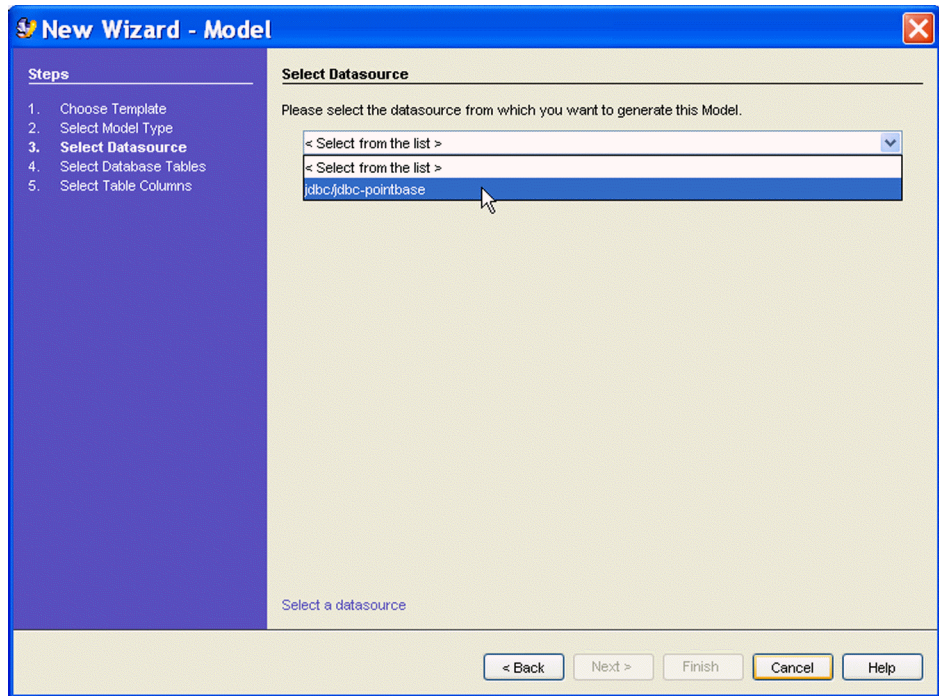
3. Enter *CustomerModel* in the Name field.

4. Select **JDBC SQL Query Model** from the model component list.

The list you see might vary depending on the Web 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.

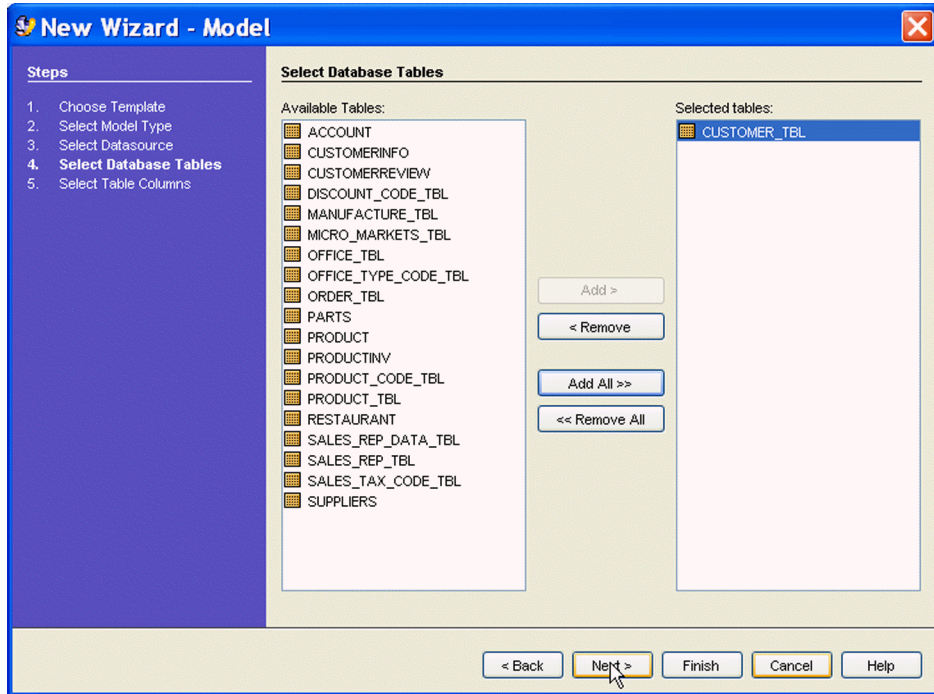
7. Click Next.

The Select Database Tables page displays.

---

**Note** – If a Connect dialog appears upon clicking Next, you may need to enter connection credentials (login - 'pbpublic', password - 'pbpublic' for the sample Pointbase database).

---

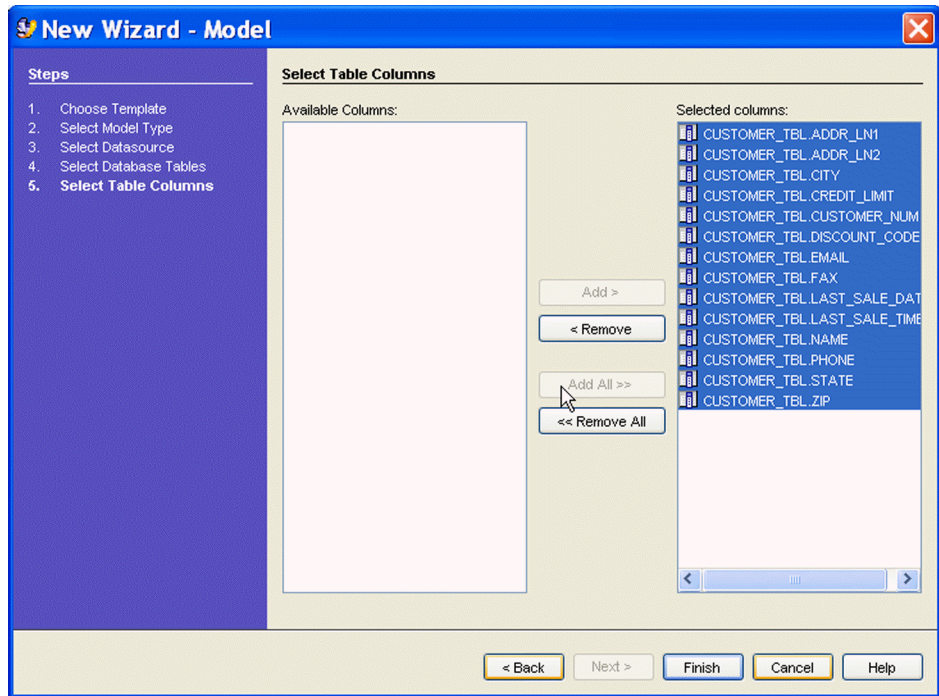


8. **Select** CUSTOMER\_TBL.

9. **Click Add.**

10. **Click Next.**

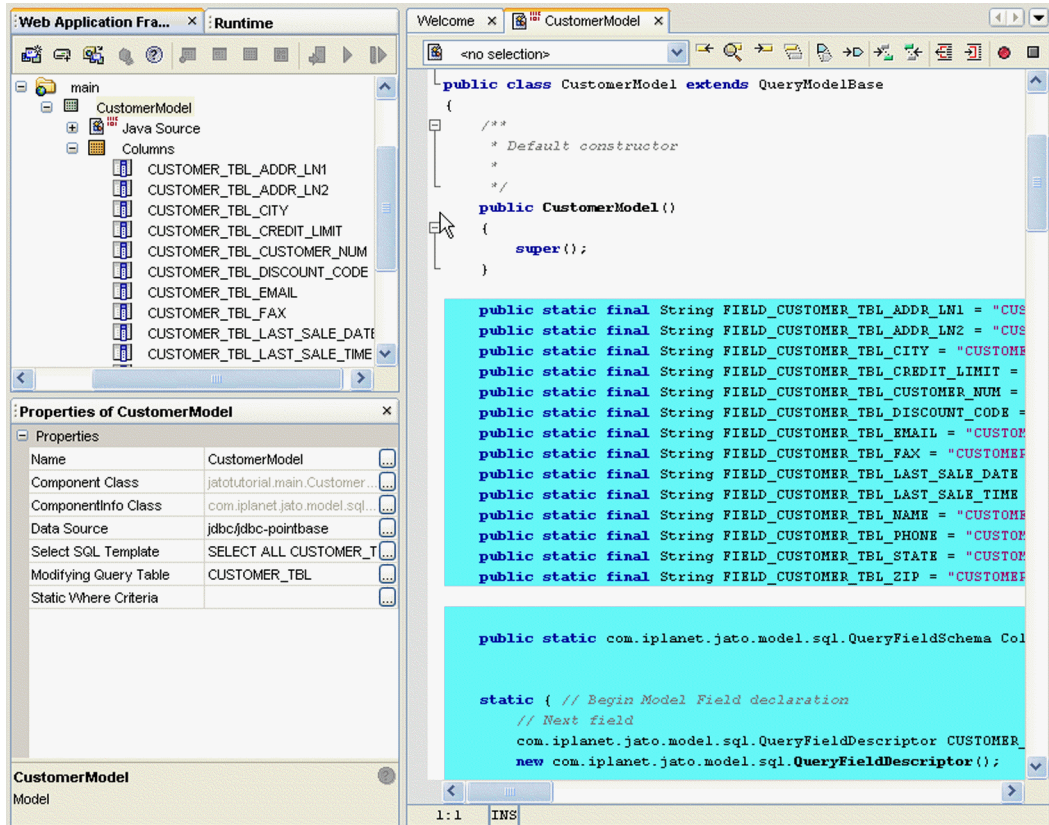
The Select Table Columns page displays.



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.



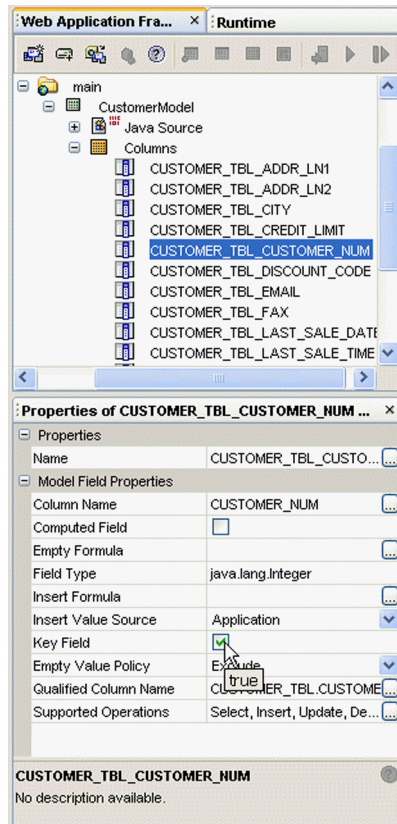
13. Expand the CustomerModel to see all of the columns.
14. Double-click the CustomerModel folder to view the code in the CustomerModel Java class.

## 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 Window -> 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*.



## 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 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 app servlet class (JatoTutorialAppServlet).

---

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.



## Tutorial—Section 2.3

# Create Customer Page

---

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

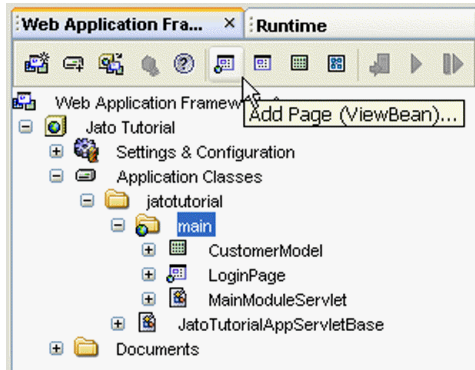
---

### Task 3: Create the Customer Page

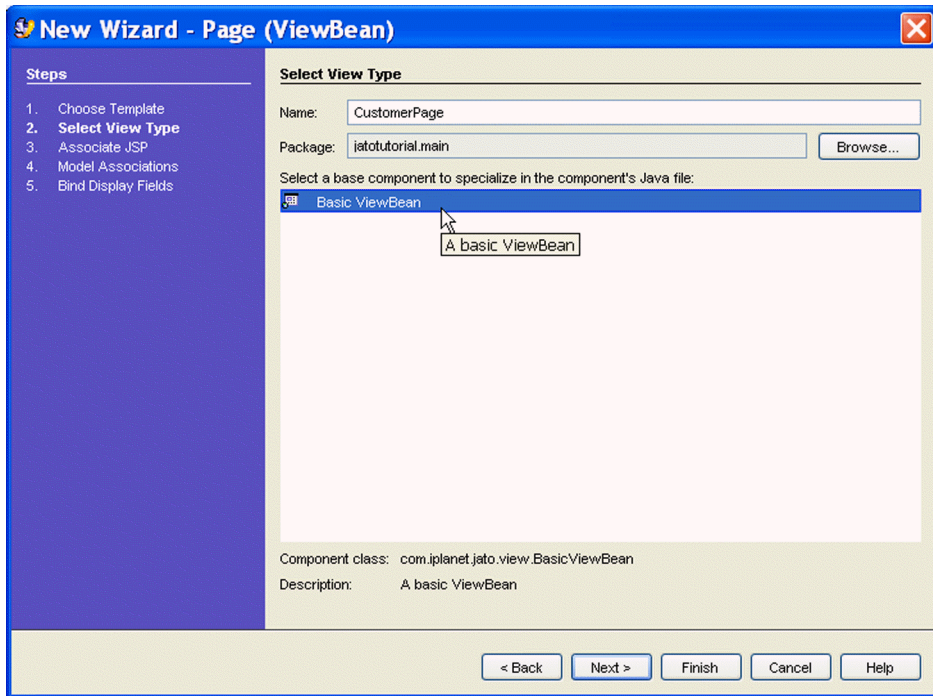
You will now create the second page of the application. However, this page will be bound to a model. This binding process automatically creates display fields on the page that display the data that is stored in the model's fields.

#### Add a ViewBean

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

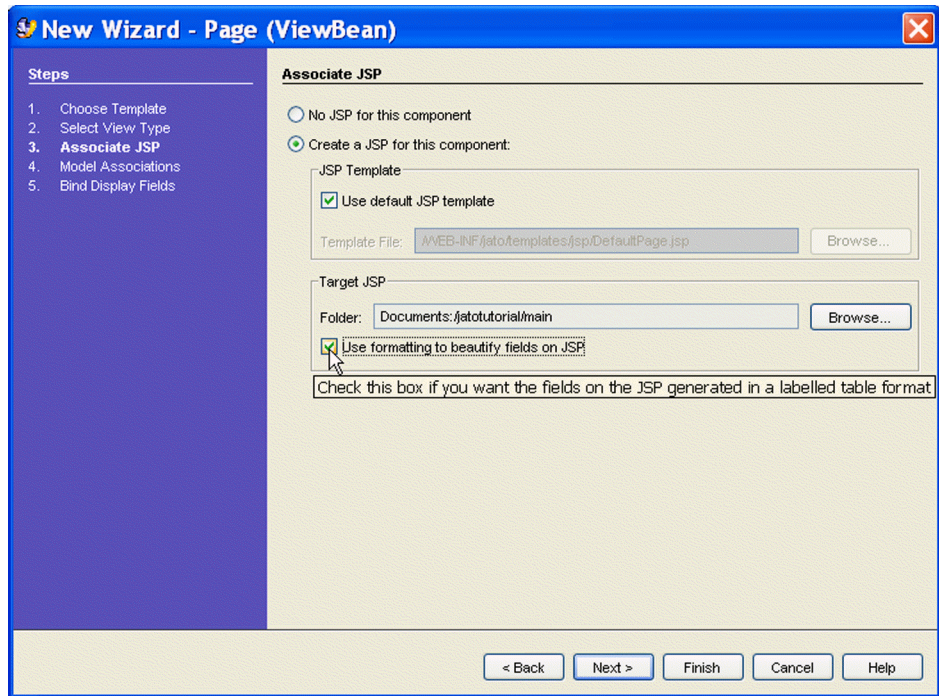


The Select View Type panel displays.



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

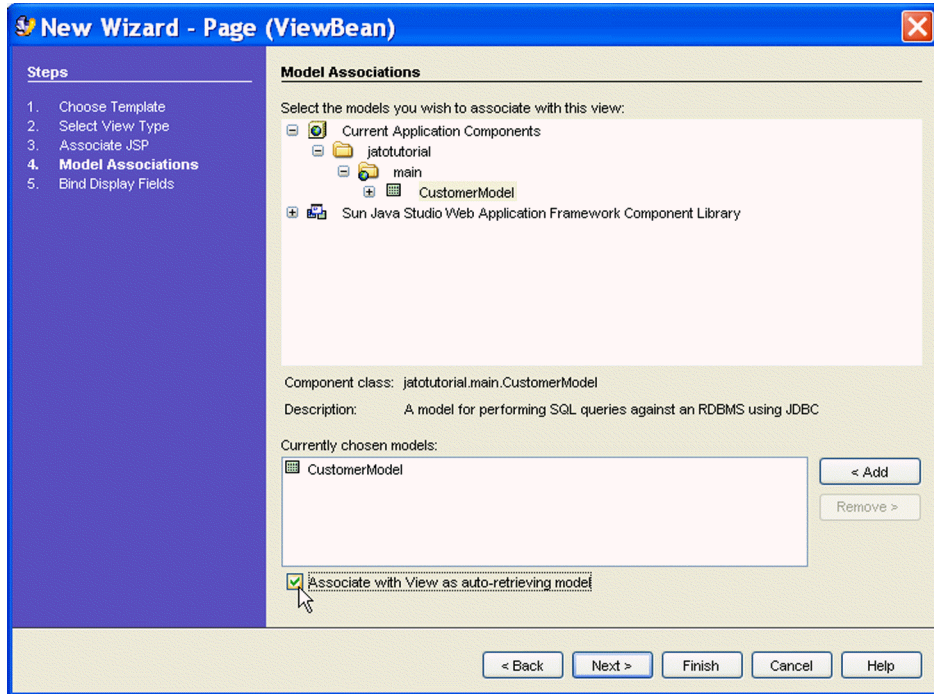
The Associate JSP panel displays.



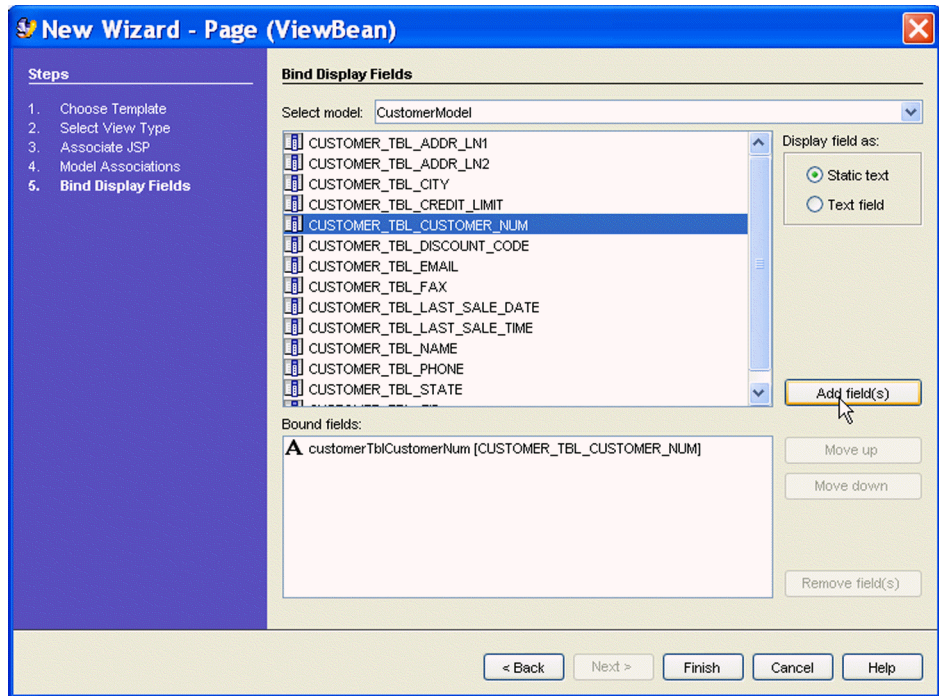
6. Click the *Use formatting to beautify fields on JSP* check box to apply some basic formatting.

7. Click Next.

The Model Associations panel displays.



8. Expand Current Application Components to expose jatutorial -> main.
9. Select Customer model.
10. Click Add.
11. Check the box *Associate with View as auto-retrieving model*.  
This will automatically configure this model to utilize its RetrievingModel features.
12. Click Next.  
The Bind Display Fields panel displays.



You only need to add three fields.

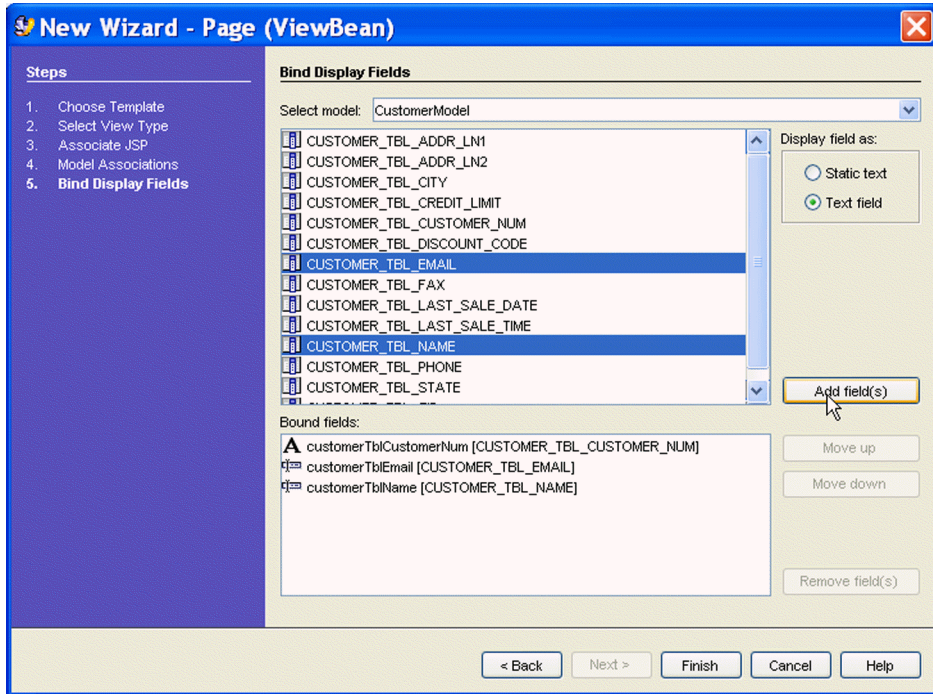
**13. Add the first field.**

**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.



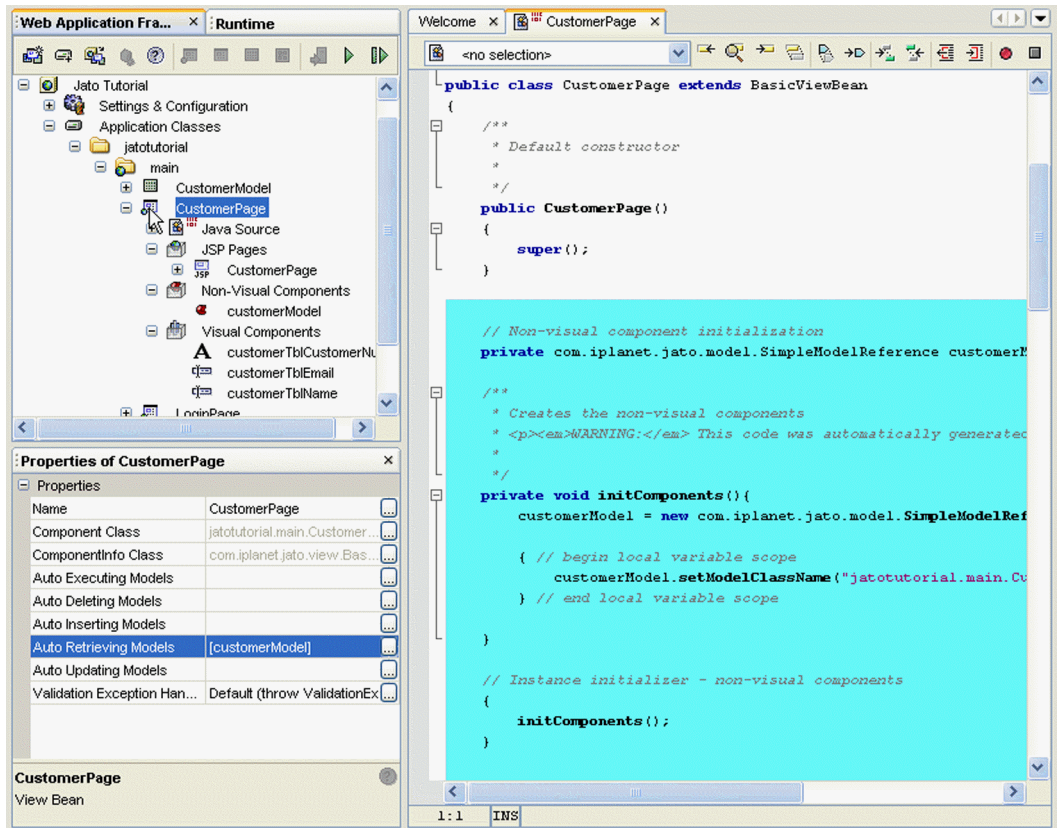
**14. Add the second and third fields simultaneously.**

- 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.

**15. Click Finish.**

You have created the ViewBean.



## 16. 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 Web 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 Web Application Framework, but it is most likely that the Web 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:

Window -> Web Application Framework -> Component Palette.

The following table lists the specifications for adding a button to the CustomerPage.

Type	Name	Initial Value
Button	update	Update

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

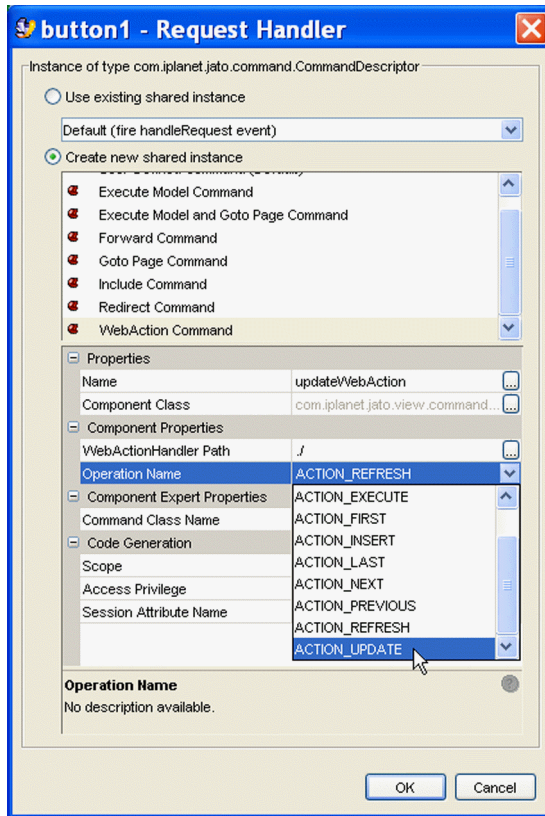
### a. Select the update button field.

### 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** section, 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**.

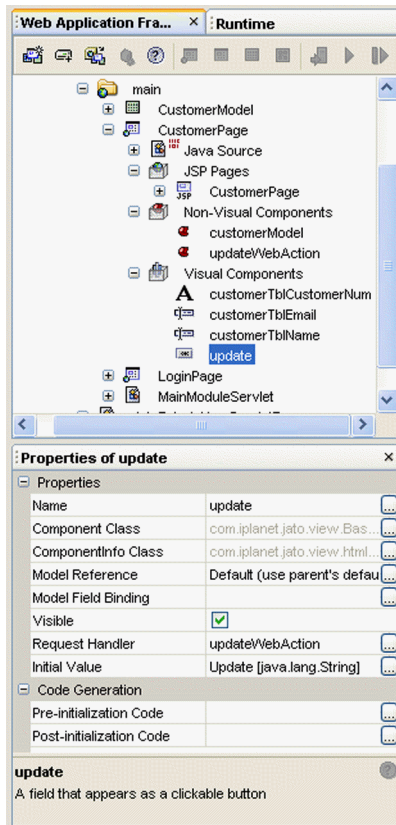
You have finished setting this property.

---

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

---

See the following image which shows the property sheet for the Model button. The Request Handler property now specifies the new command descriptor configured in the previous step.



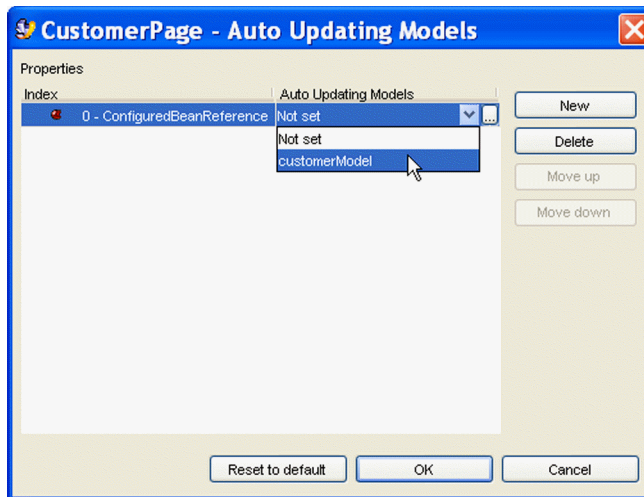
## Making a Model Auto Update

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.

---

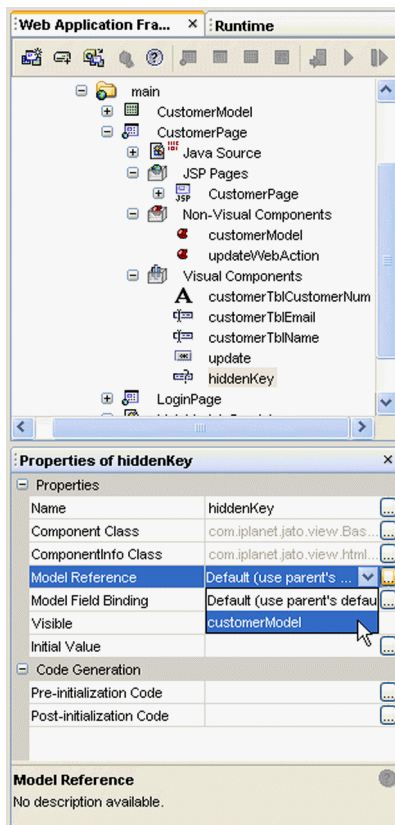
**Note** – The combination of the button's update Web action command descriptor and the auto retrieving or updating models configuration causes the CustomerModel to be executed when the Customer page is displayed or when the CustomerPage's Update button is clicked.

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 *handleUpdateRequest* event (similar to how the code was implemented for the Login button on the Login page).

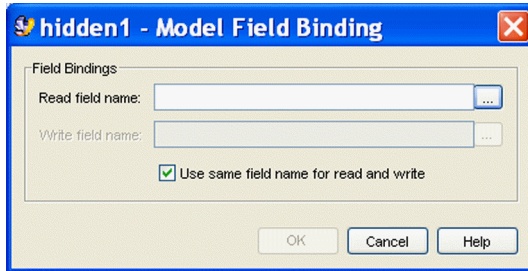
---

# Add a Hidden Field to the Customer Page

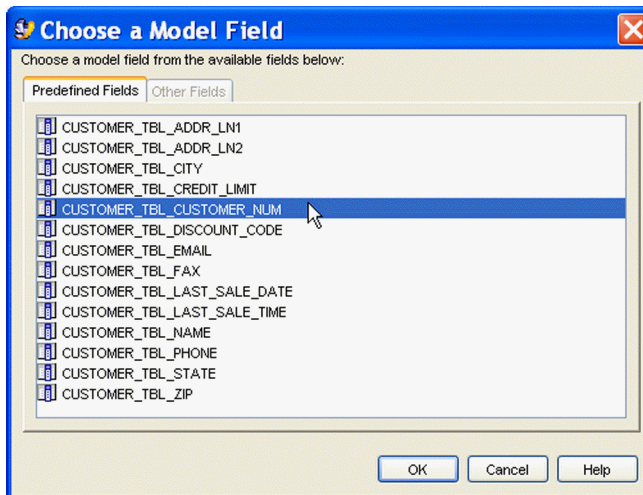
1. Expand the CustomerPage node.
2. Expand the Visual Components node.
3. Select the Visual Components node.
4. Select add a Hidden Field component using the Component Palette.  
A hidden field is added to the CustomerPage's Visual Components node.
5. Rename the hidden field as *hiddenKey*.
6. Bind the 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.

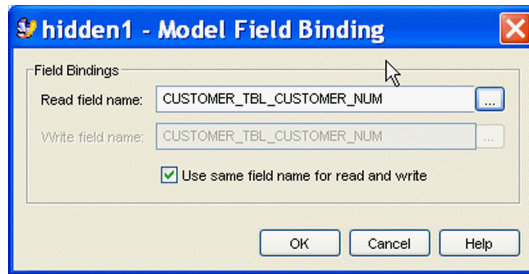


10. Click the ellipsis button of the *Read field name* property to launch the Model Field Chooser editor.



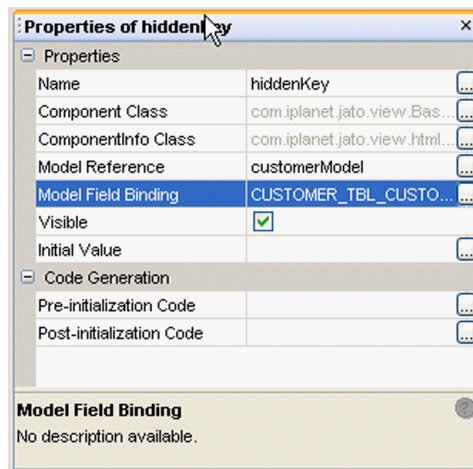
11. Select *CUSTOMER\_TBL\_CUSTOMER\_NUM*.
12. Click OK.

The read and write fields are populated with the *CUSTOMER\_TBL\_CUSTOMER\_NUM* model field.



13. Click OK.

You have now set the Model Field Binding property for the *hiddenKey* display field.



Because the static text field is not an HTML input field, its 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 Web 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 Web 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 *Web 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>
```



## Tutorial—Section 2.4

# Test Run the Customer Page

---

This chapter describes how to run your Web Application Framework application.

---

### Task 4: Test Run the Customer Page

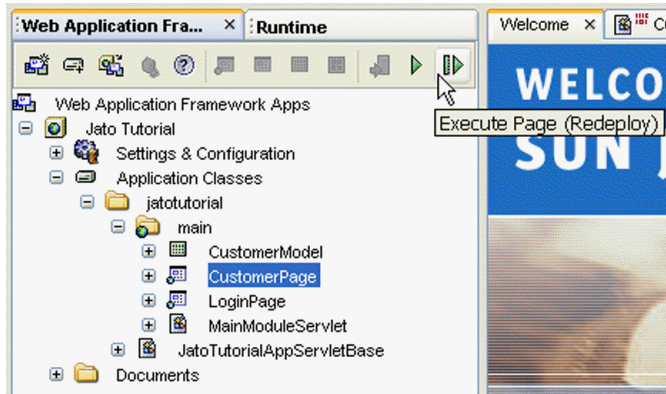
**Important:** Make sure the PointBase Network Server is running. If it is not, you can start it by doing the following:

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 Java System Application Server, you must Deploy the application when changes are made.

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

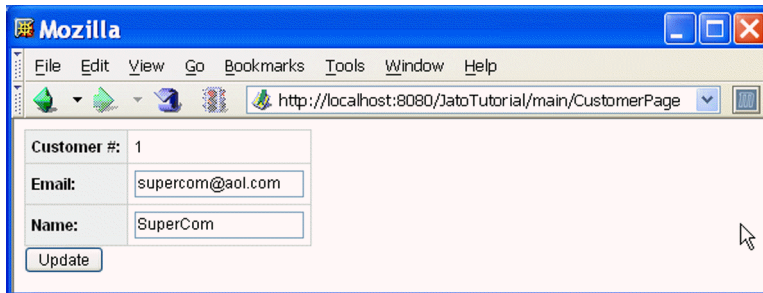
Using this execute and redeploy option restarts the server to ensure the server picks up all changes and does not use any cached resources.



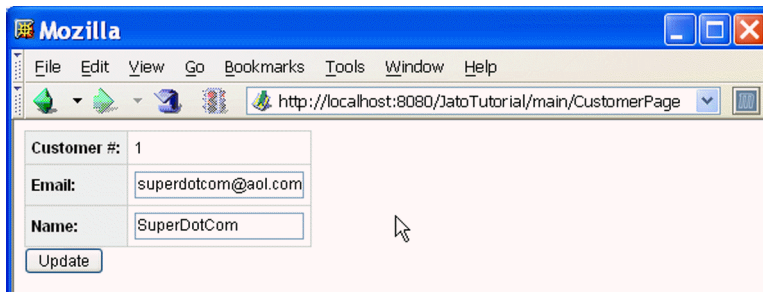
A default browser starts the application.

## Test a Customer Update

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



2. Click Update.



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

## Tutorial—Section 2.5

# Link Login Page to Customer Page

---

This chapter describes how to link the LoginPage to the CustomerPage in the Web 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();
    }
}

```





## Tutorial—Section 2.6

# Run Application

---

This chapter describes how to run the Web 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 IDE 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.

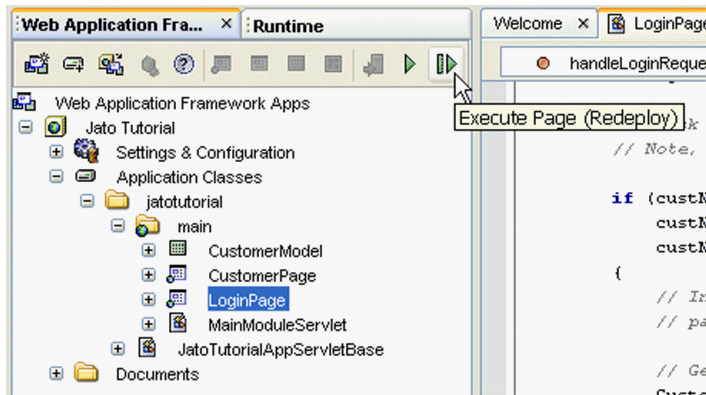
2. **Right-click the Application Classes node, and select the Compile All action.**

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

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

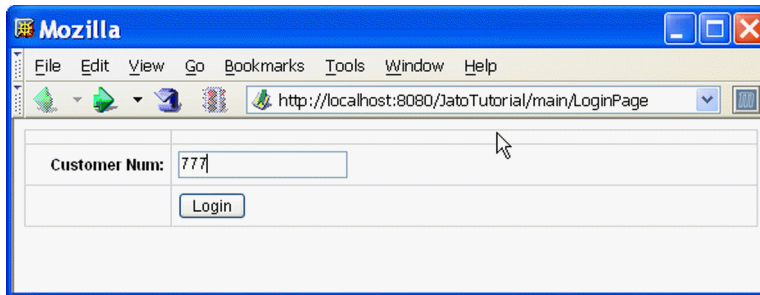
4. **Select the LoginPage node, and click the Execute Page (Redeploy) button**

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.



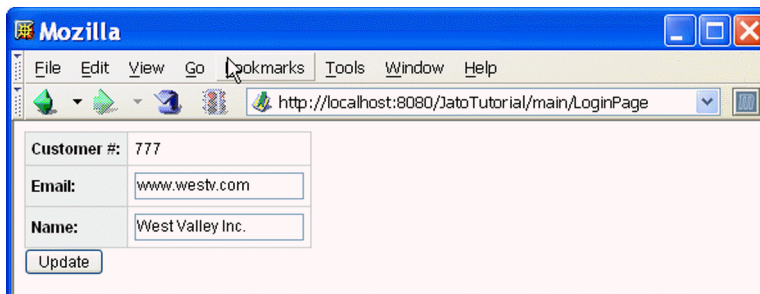
A default browser starts the application.

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.



## 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.ipplanet.jato.command.Command` interface. In this tutorial, you will create a new Command class using the Web 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.

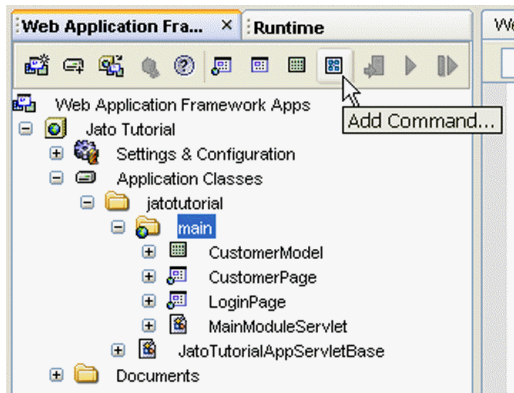
---

## Task 1: Create a Command Component

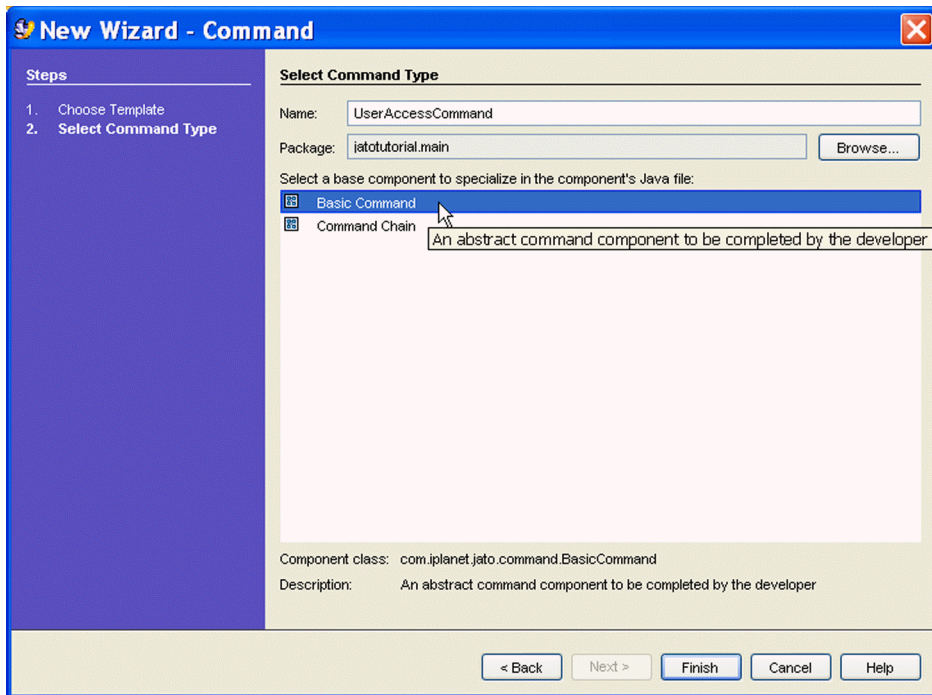
### Create the UserAccessCommand Component

Create a Command component using the Web Application Framework Command wizard.

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

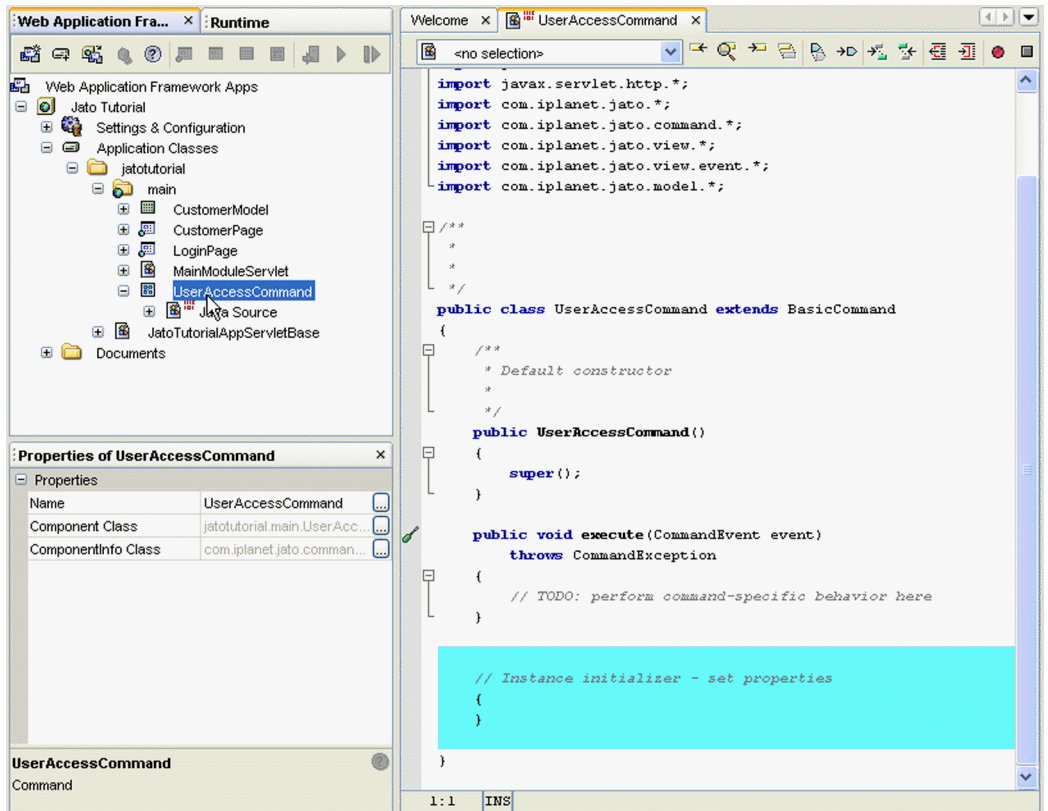


The Select Command Type panel displays.



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 CommandException
```

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 reimplementing 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);

// Display the Login page
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();

// Set the status message value
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);

// Display the Login page
loginVB.forwardTo(requestContext);
} // else if opName = logout

// unknown operation name
else {
throw new CommandException(

```

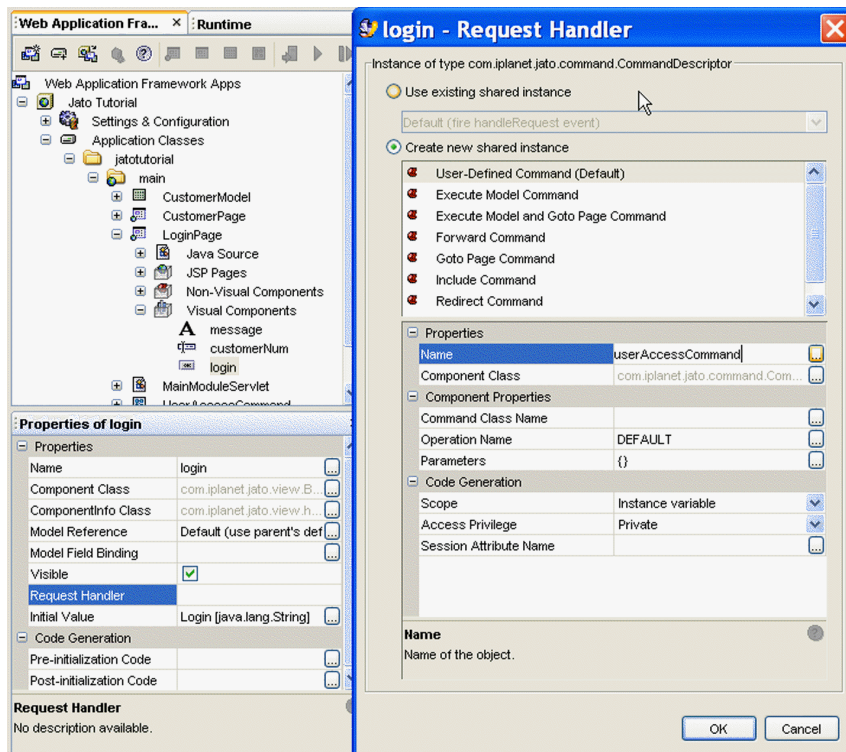
```
"Unknown UserAccessCommand operation name: " + opName);  
}  
}
```

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.

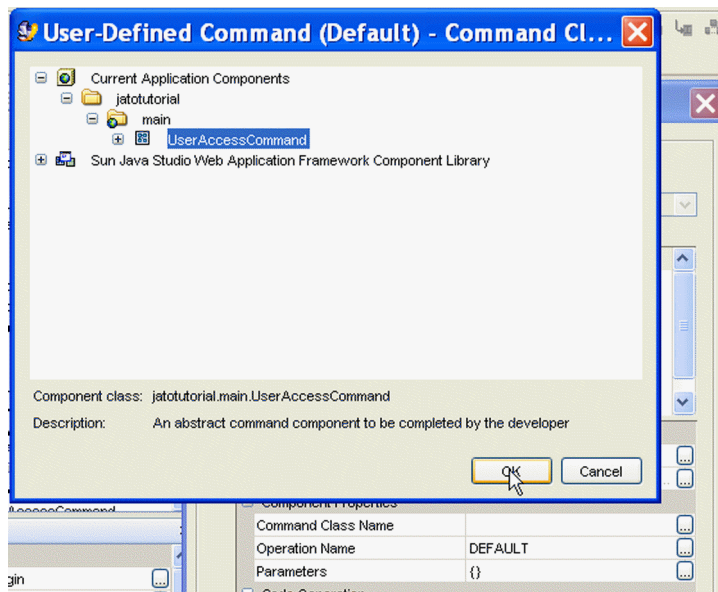
1. Expand the LoginPage node, and expand the Visual Components node.
2. Select the login button under Visual Components.



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.

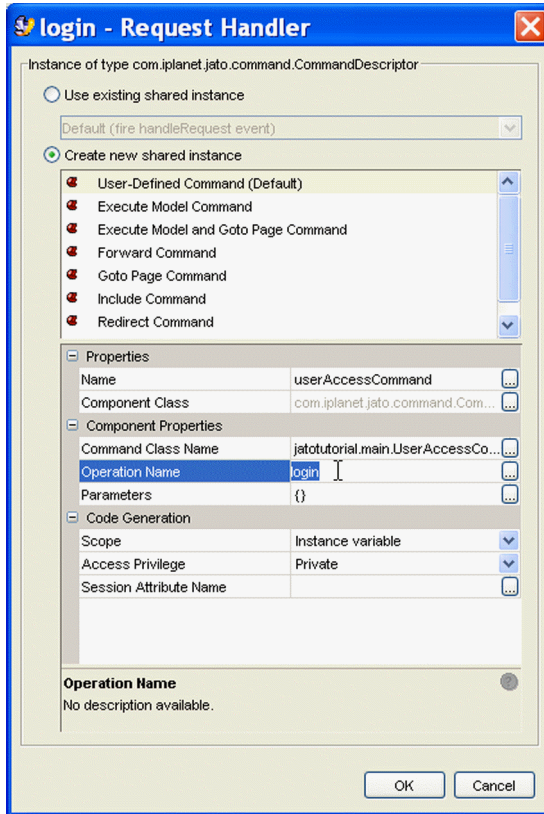


5. Change the Name property to `userAccessCommand`
6. Select the Component Properties tab at the bottom of the editor.
7. Click the ellipsis button for the *Command Class Name* property.  
This displays the Command Class Name chooser dialog.
8. Expand the Current Application Components node, then `jatutorial`, 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 Web 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` method. If the method is not implemented, you receive a *request handler not found* exception.

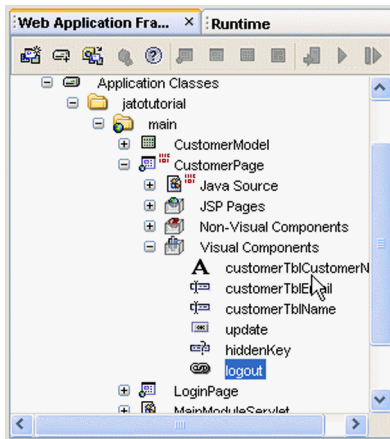
# Tutorial—Section 3.2

## 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*.

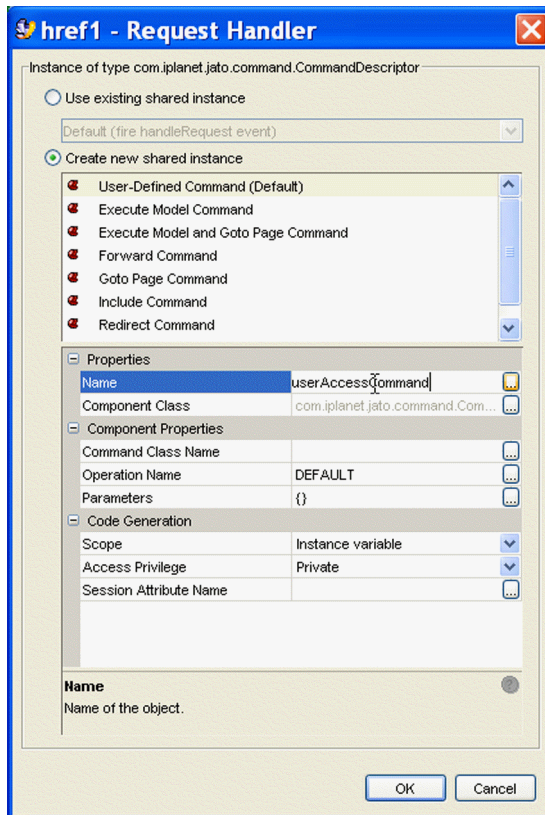


## Configure an HREF's Command Descriptor

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*.

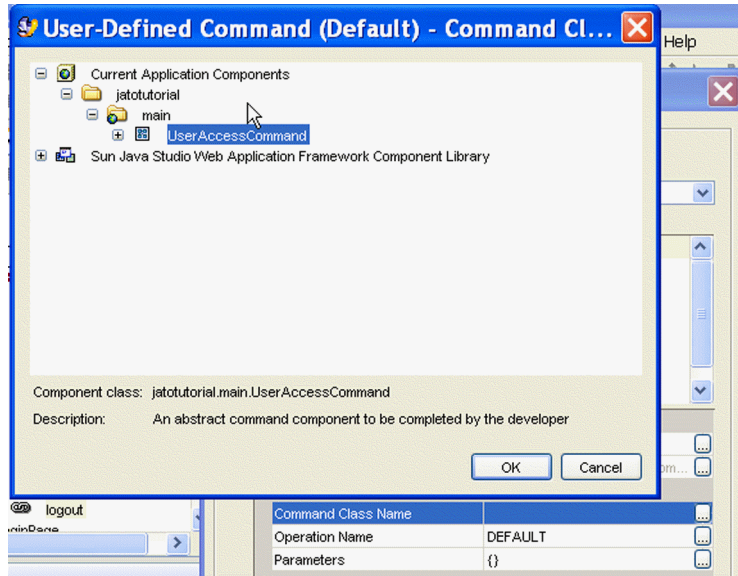
1. Select the logout HREF under `CustomerPage`'s Visual Components node.
2. Click the ellipsis button for the Request Handler property.

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.

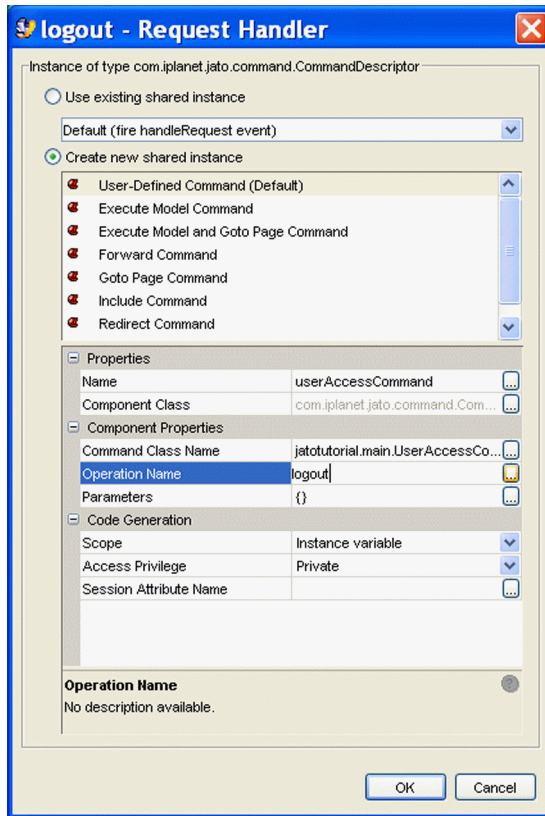
This displays the Command Class Name chooser dialog.



7. Expand the **Current Application Components** node, then expand **jatotutorial**, 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>*).





## Tutorial—Section 3.3

# Test Run the Login/Logout Command Component

---

This chapter describes how to run your Web 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 IDE 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 Java System Application Server, you must Deploy the application when changes are made.**

Select the Web Application Framework Application node (JatoTutorial), and click the Deploy button on the Web 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.

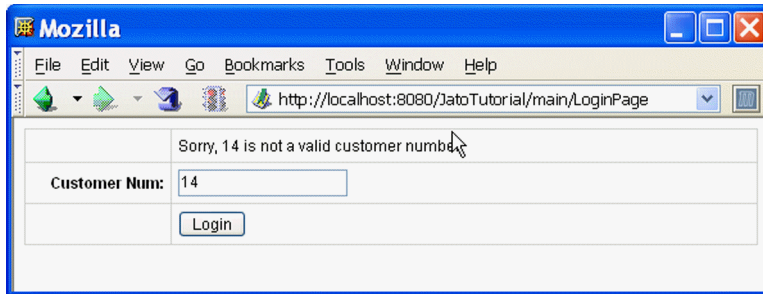
---

**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.

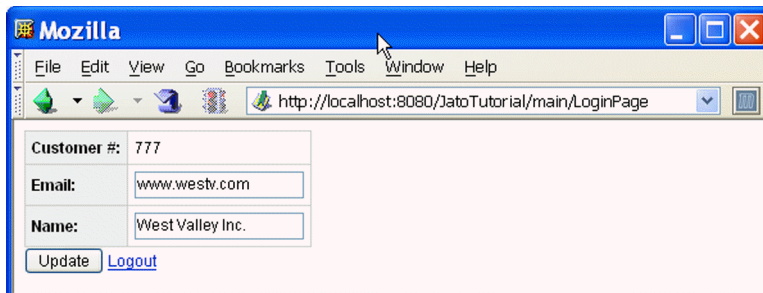
For your convenience, a list of the valid customer numbers from the sample PointBase database is as follows: 1, 2, 3, 25, 36, 106, 149, 409, 410, 722, 753, 777, 863

---

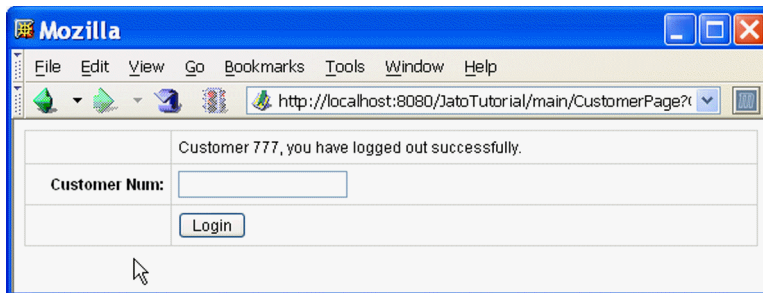
**5. Enter an invalid customer number first.**



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



**7. Try the Logout link.**



# 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 IDE 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 Web 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 IDE some time to refresh its state.

If it appears that the IDE 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.

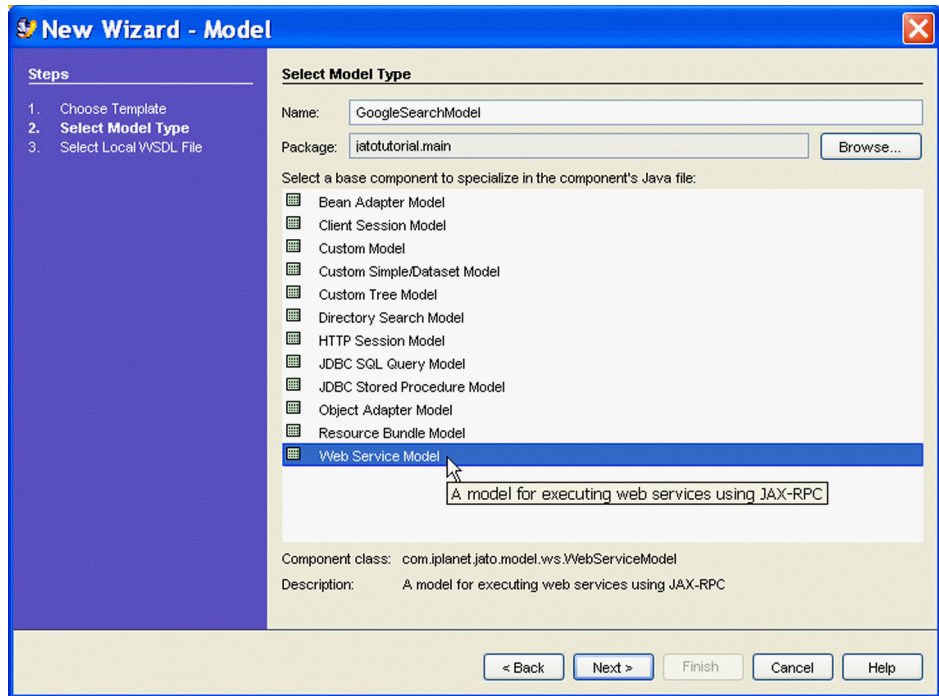
## Create the Web Service Model

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.**

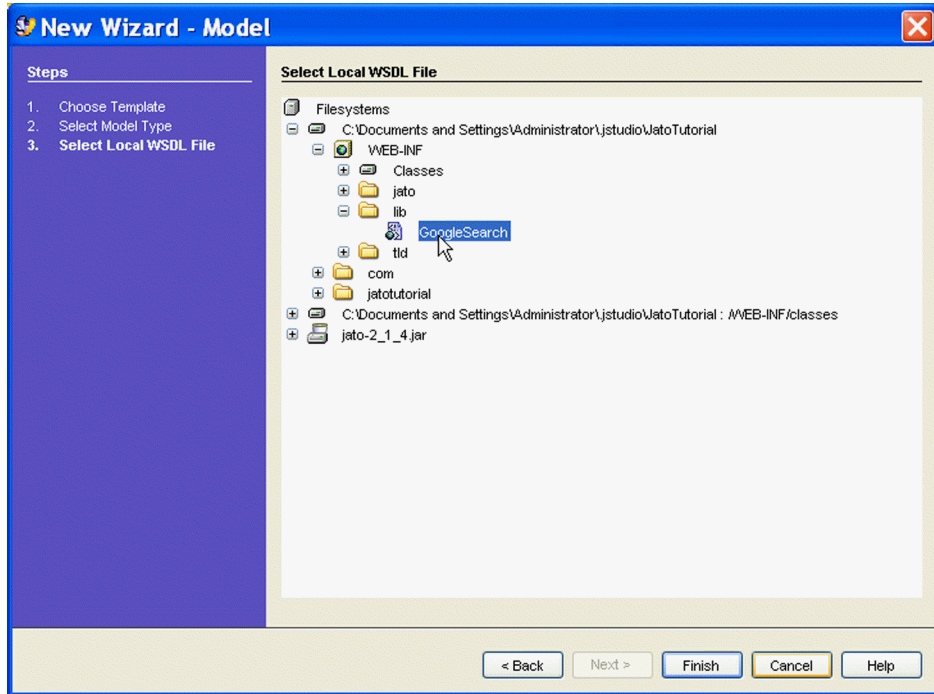
**2. Click the Add Model button on the Web Application Framework toolbar.**

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.

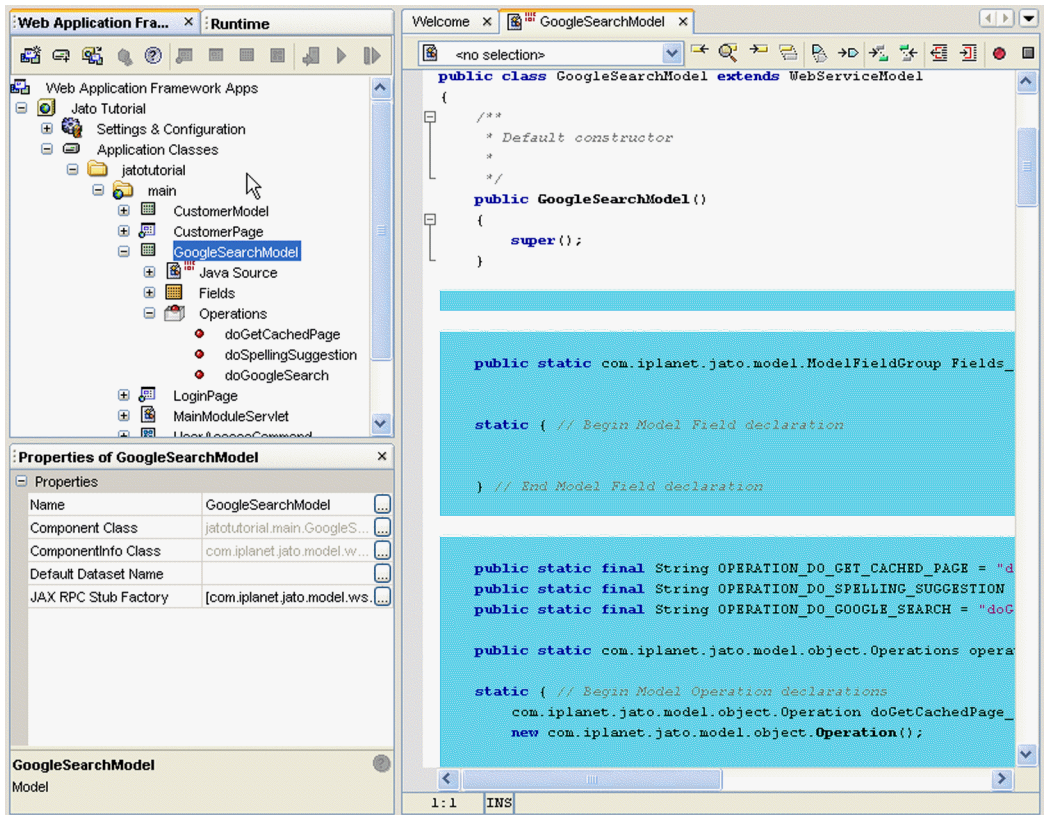


Navigate down the JatoTutorial application directory structure to the `lib` directory (`JatoTutorial/WEB-INF/lib`), and select `GoogleSearch` (the WSDL file).

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.

---





## Tutorial—Section 4.2

# Create the Google Search Page

---

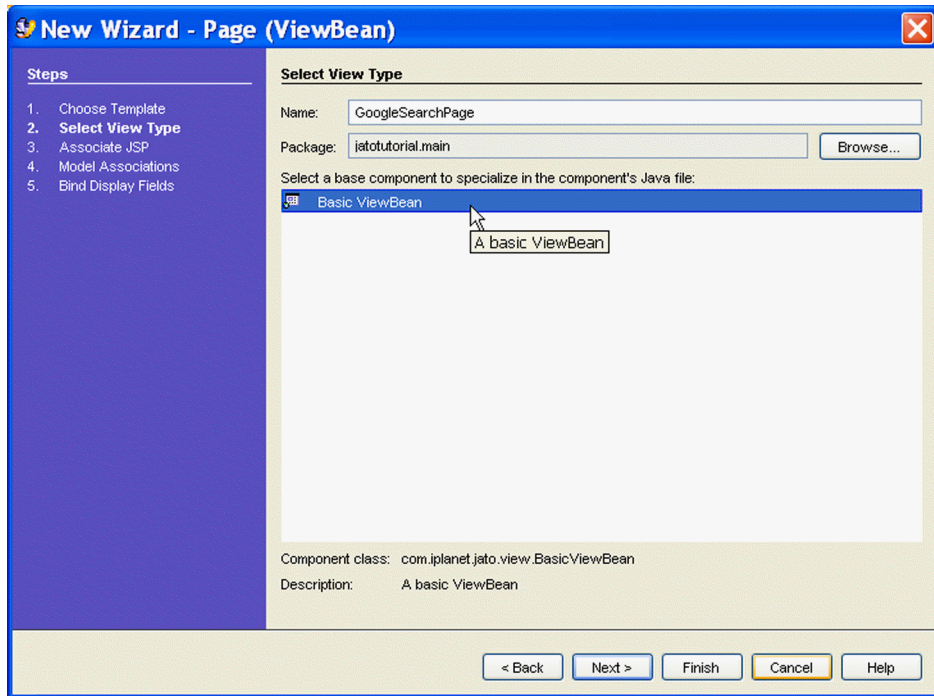
This chapter describes how to create a page in the Web Application Framework that displays data it gets from a model that accesses data from a Web service.

---

## Task 2: Create the Google Search Page

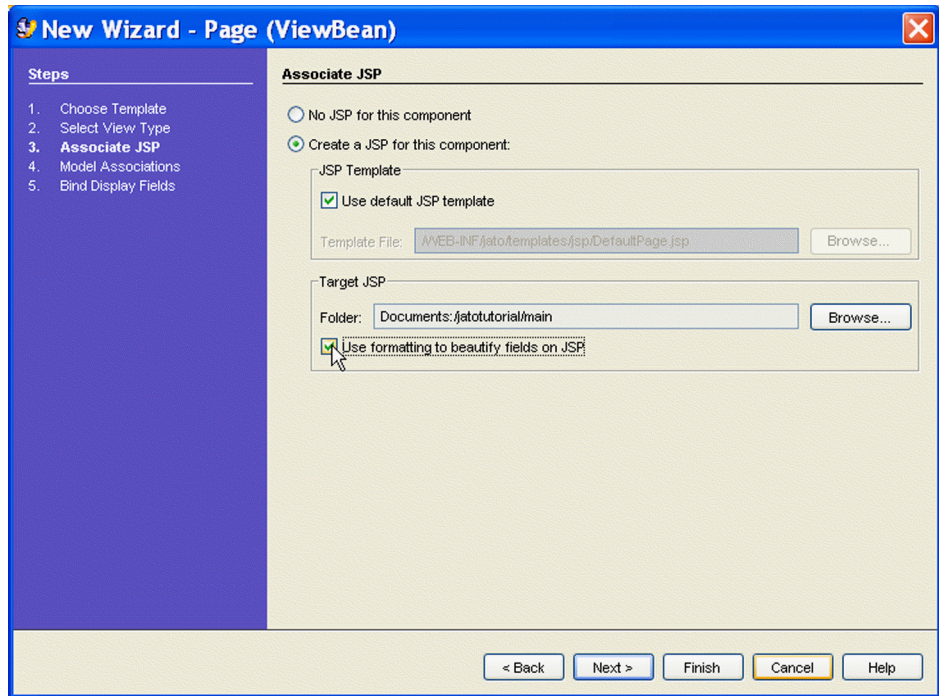
### Add a Page Component

1. **Select the main module folder.**
2. **Click the Add Page button on the Web 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`.

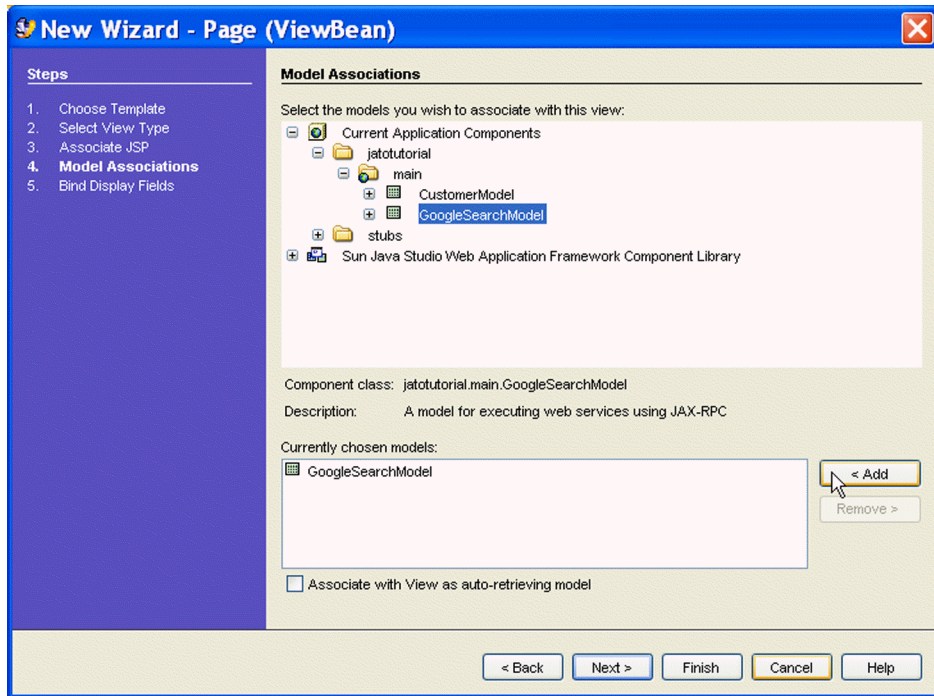
The Associate JSP panel displays.



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

7. Click Next.

The Model Associations panel displays.



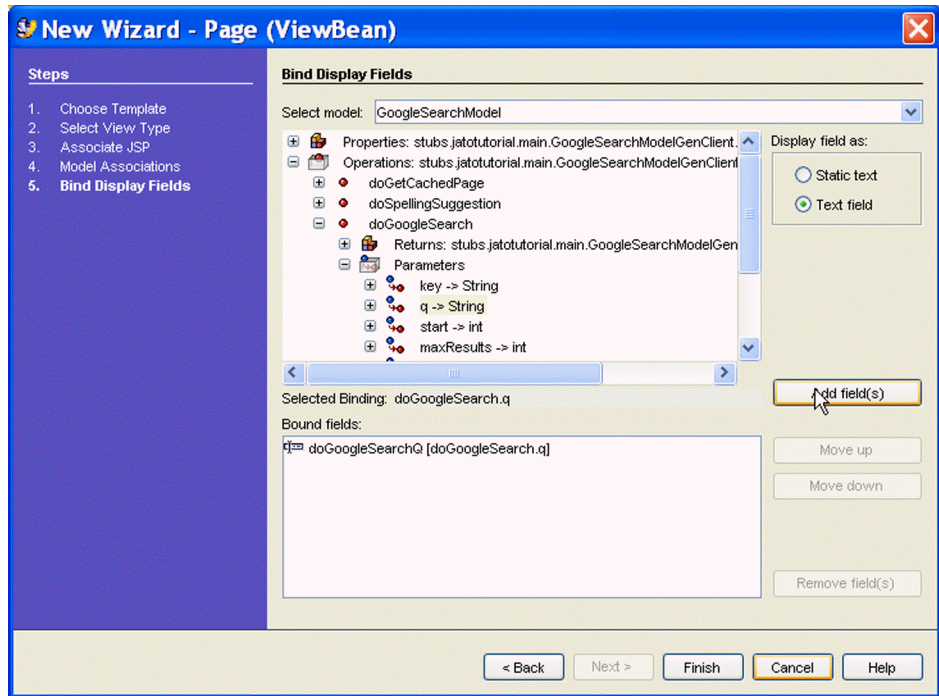
8. Expand Current Application Components to expose `jatutorial` -> `main`.

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

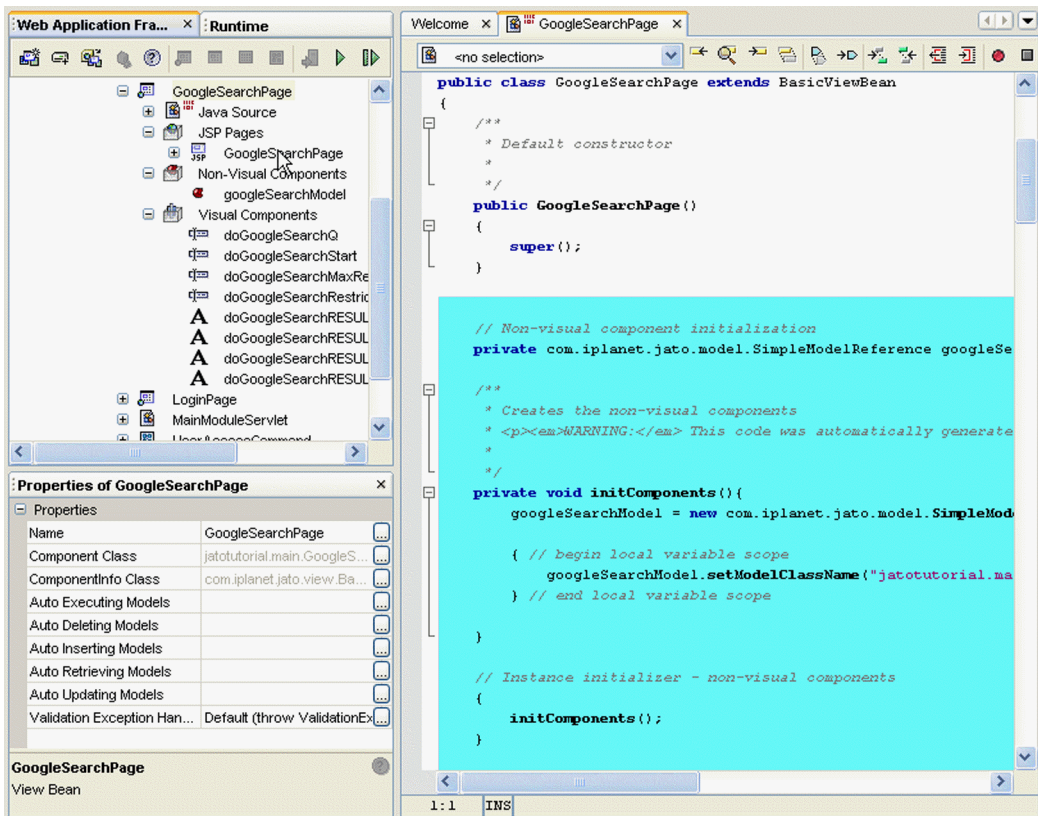
14. Add the following fields as Static text fields (not Text fields):

(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. 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 <b>Q</b>	to queryString
doGoogleSearch <b>Start</b>	to start
doGoogleSearch <b>MaxResults</b>	to max
doGoogleSearch <b>Restrict</b>	to restrict
doGoogleSearchRESULT <b>S</b> SearchTime	to searchTime
doGoogleSearchRESULT <b>S</b> StartIndex	to startIndex
doGoogleSearchRESULT <b>S</b> EndIndex	to endIndex
doGoogleSearchRESULT <b>S</b> 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

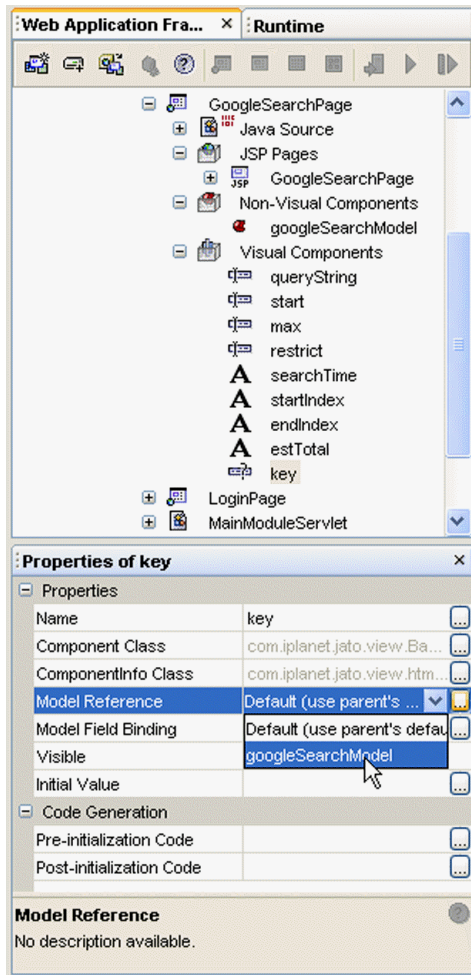
---

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*.

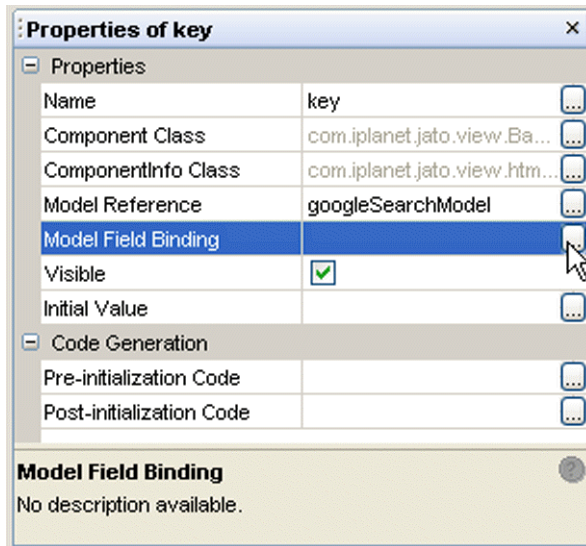
4. Set the Model Reference property for the *key* field.



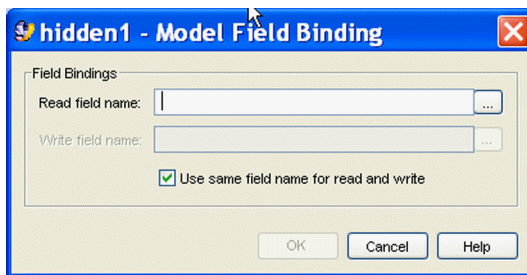
5. Select `googleSearchModel` from the drop down list.

6. Set the Model Field Binding property.

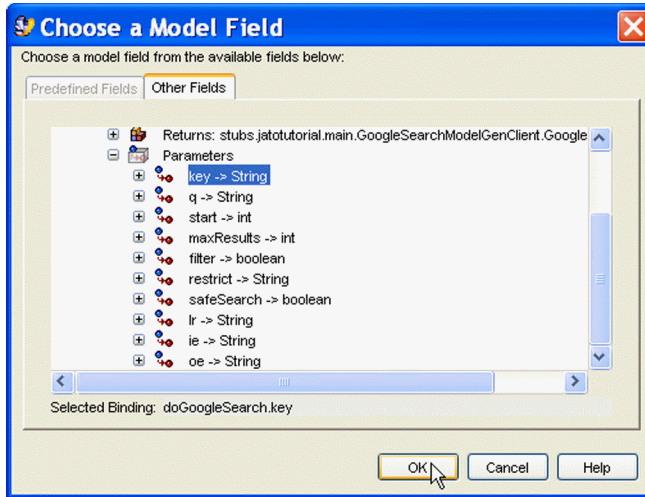




7. Click the ellipsis button to launch the Model Field Binding editor.

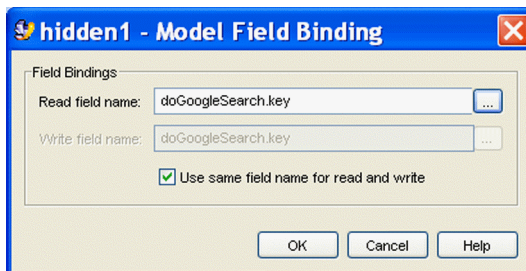


8. Click the ellipsis button for the Read field name property in this editor.



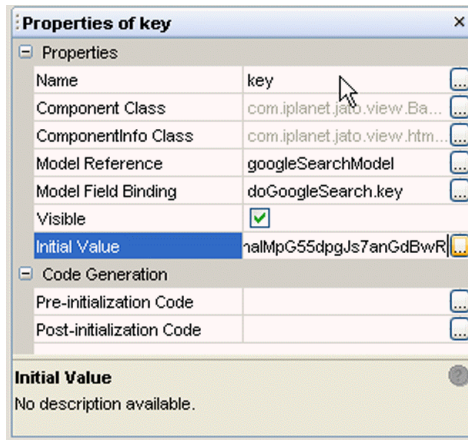
9. Expand the doGoogleSearch operation node, then expand the Parameters node.
10. Select key -> String.
11. Click OK.

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.



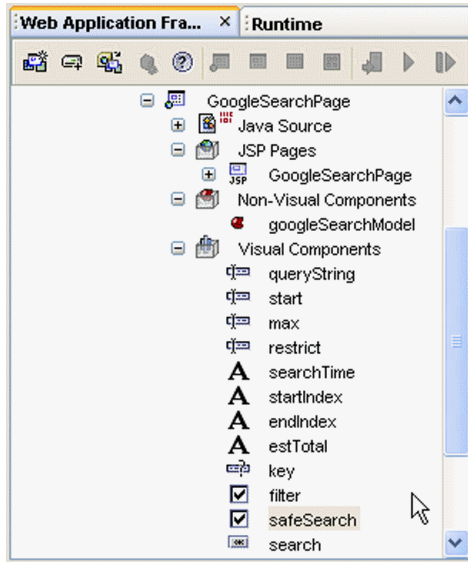
The key field's properties should look something as shown in the figure above, however, your key's initial value will be different.

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

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

Type	Name	Initial Value	Model Reference	Model Field Binding
Check Box	filter		googleSearchModel	doGoogleSearch/Parameters/filter
Check Box	safeSearch		googleSearchModel	doGoogleSearch/Parameters/safeSearch
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);

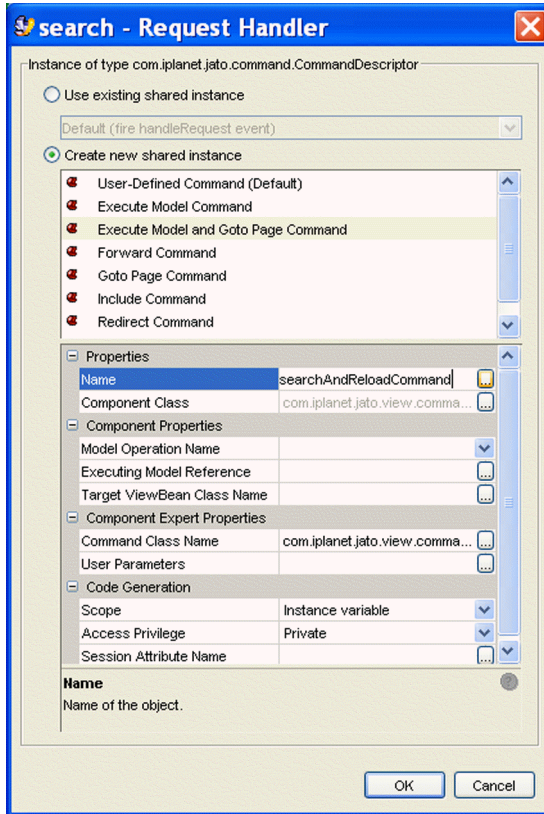
    // execute the model using the doGoogleSearch operation
    // (the model execution context)
    model.execute(new ModelExecutionContextBase("doGoogleSearch"));

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

## Point & Click Technique (code-free)

1. **Select the search button.**

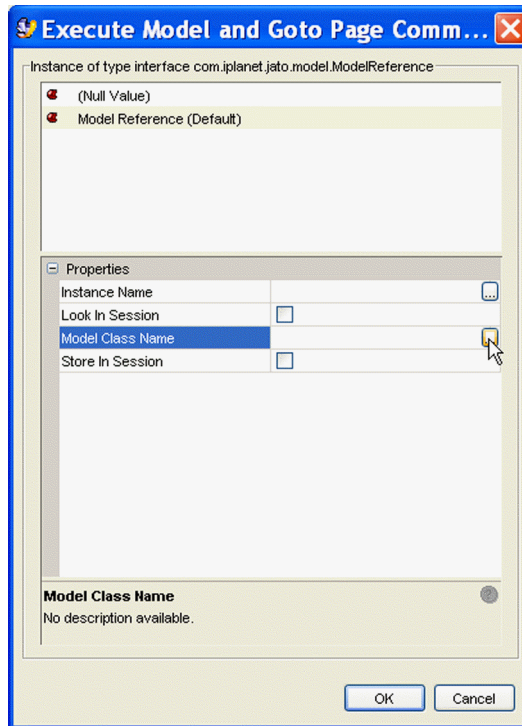
2. **Launch the editor for its Request Handler property by clicking the ellipsis button.**



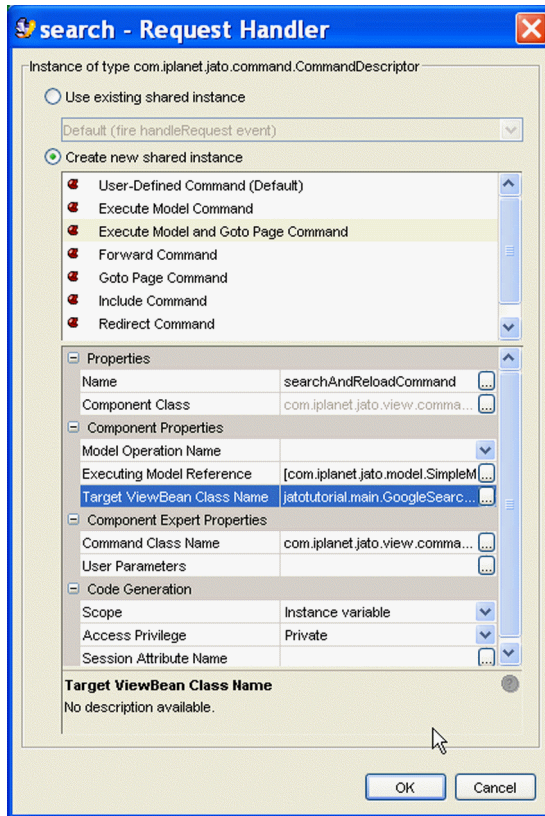
3. Select the *Execute Model and Goto Page Command* option.

4. Set the Name to *searchAndReloadCommand*.

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



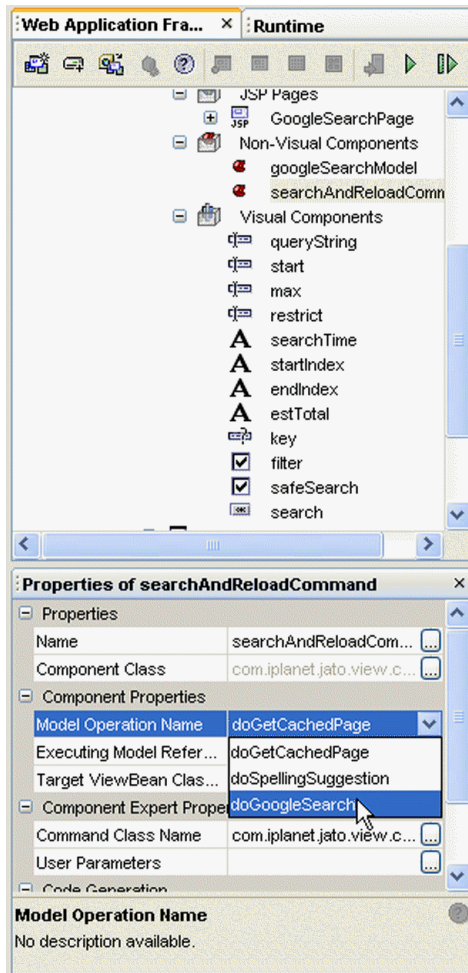
5. For the Executing Model Reference property, launch its editor by clicking the ellipsis button.
6. Select the Model Reference (Default) option.



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*.
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 IDE's source 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** next:

```

<jato:form name="GoogleSearchPage" method="post">
<b>Google Search</b>
<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>

```



## Tutorial—Section 4.3

# Test Run the Google Search Page

---

This chapter describes how to run your Web 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 Java System Application Server, you must Deploy the application when changes are made.
2. **Select the Web Application Framework Application node (JatoTutorial), and click the Deploy button on the Web 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.ipplanet.jato.NavigationException: Exception encountered
during forward
Root cause = [com.ipplanet.jato.model.ModelControlException: no
current dataset assigned yet]
```

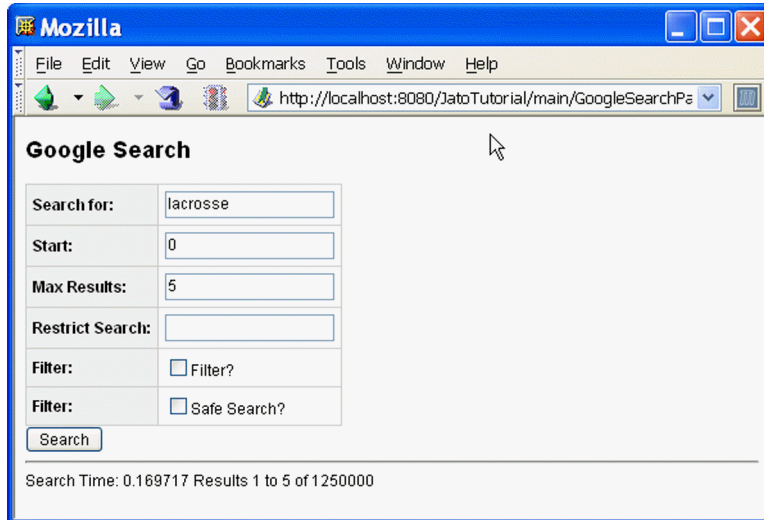
---

# 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.**

# Tutorial—Section 4.4

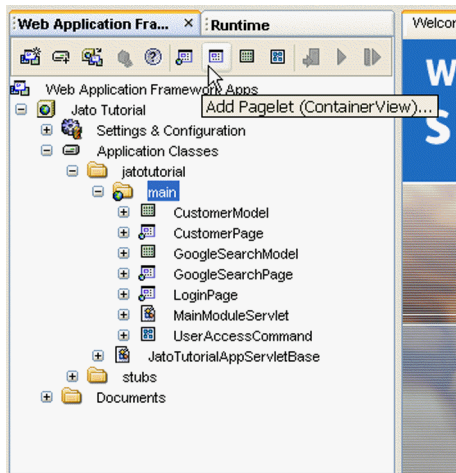
## Add Results Listing to the Google Search Page

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.

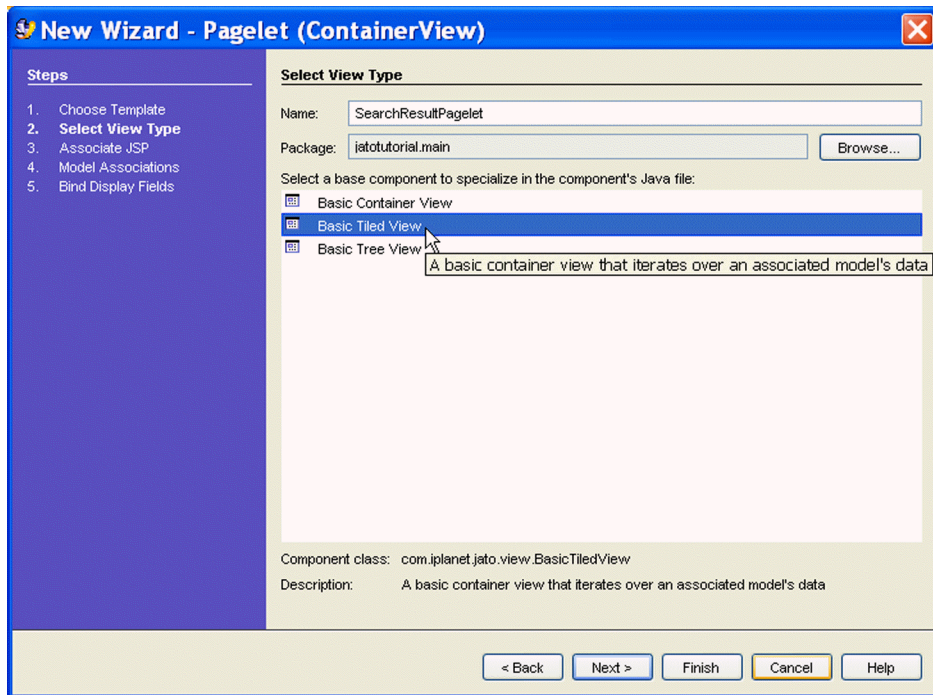
### Task 4: Create a TiledView Pagelet

#### Add a TiledView

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



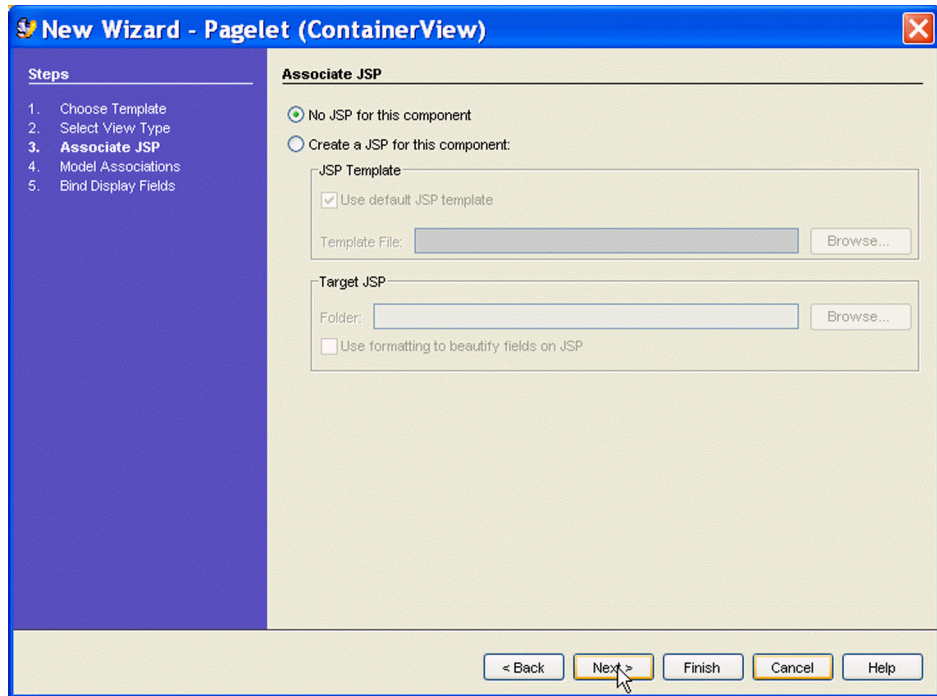
The Select View Type panel displays.



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.

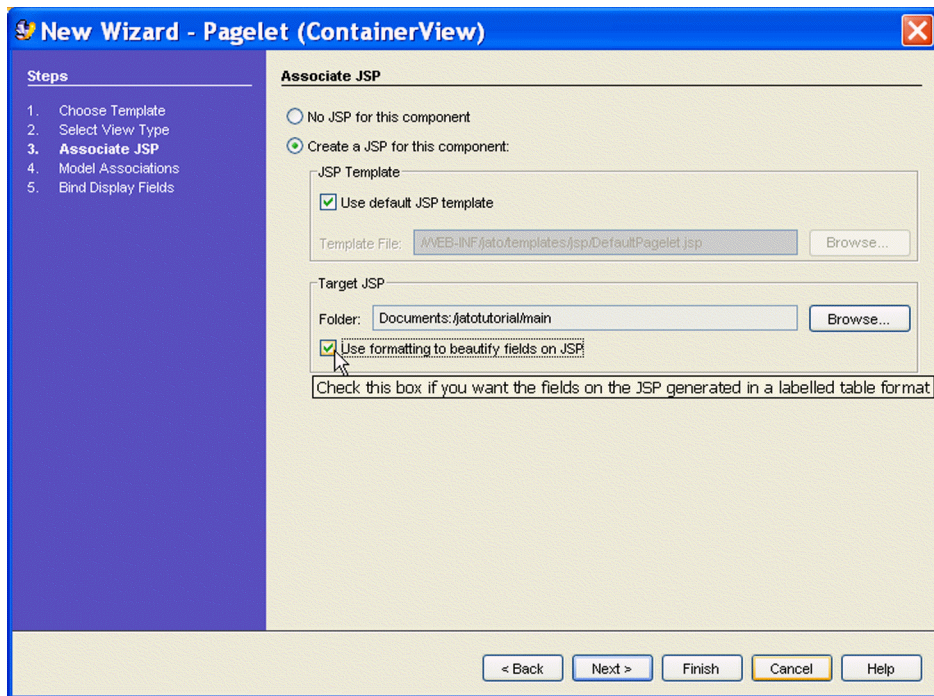
The question is whether to create a JSP for a pagelet component or not. There are pros and cons to both possibilities. The deciding factor is how the pagelet component will be reused on the JSP side. If the pagelet is meant to be rendered the same regardless of what page (or another pagelet) parents 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.

---

**Note** – In the figure below we show the alternative pagelet technique of creating a JSP file for the new pagelet. Here we select the ‘Create a JSP for the component’ radio button and we also check the box ‘Use formatting to beautify fields on JSP’. Again, the difference between this technique and that shown above is that the new pagelet will have an associated JSP fragment. When this pagelet is added as a child view component of another page or pagelet, if there is an associated JSP then the parent component will use an include directive to render the JSP fragment. At this point in the tutorial you may use either technique.

---

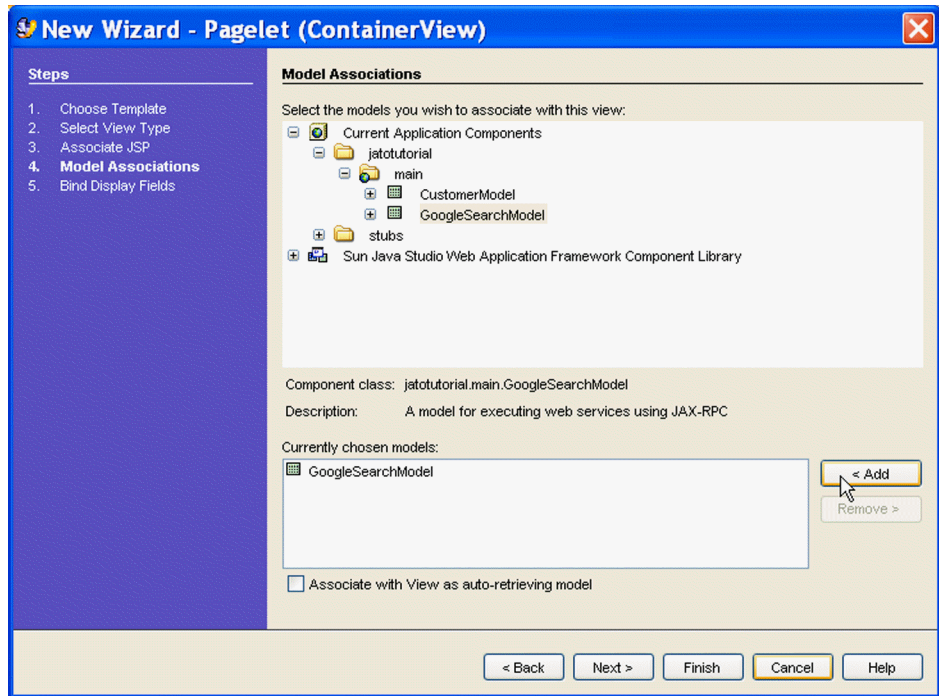


The nuances of these techniques might not be clear to you right away, but do not be concerned. As you become more skilled with JSP pages and the Web Application Framework, you will begin to fully understand the flexibility and power of the reusability of the Web Application Framework page and pagelet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

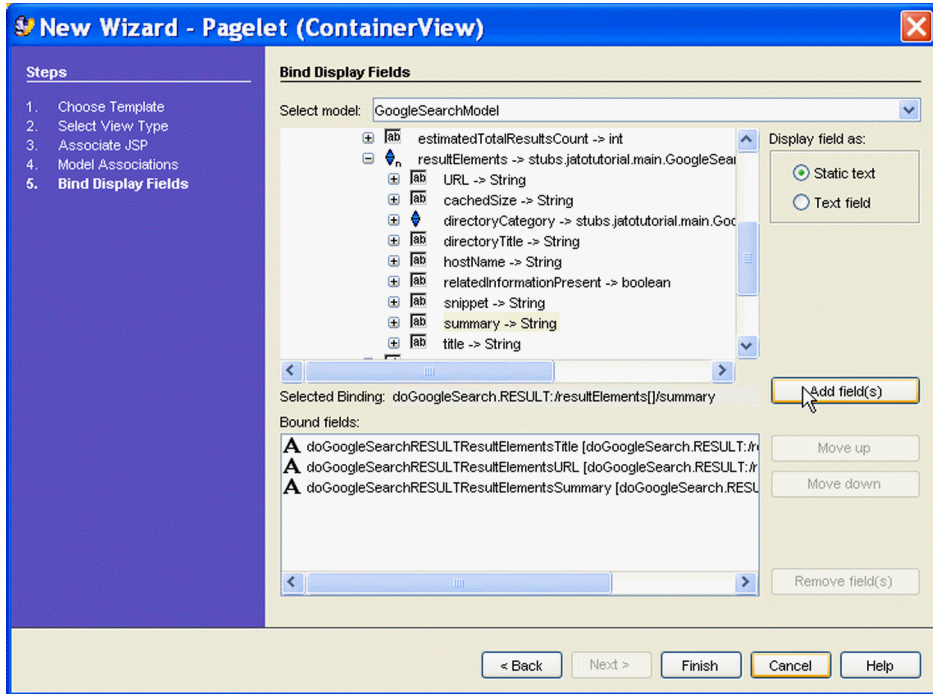
For more information, read the notes that are included on this wizard panel, and refer to the *Web Application Framework Developer's Guide* and the *Web Application Framework Component Author's Guide*.

##### 5. Click Next.

The Model Associations panel displays.

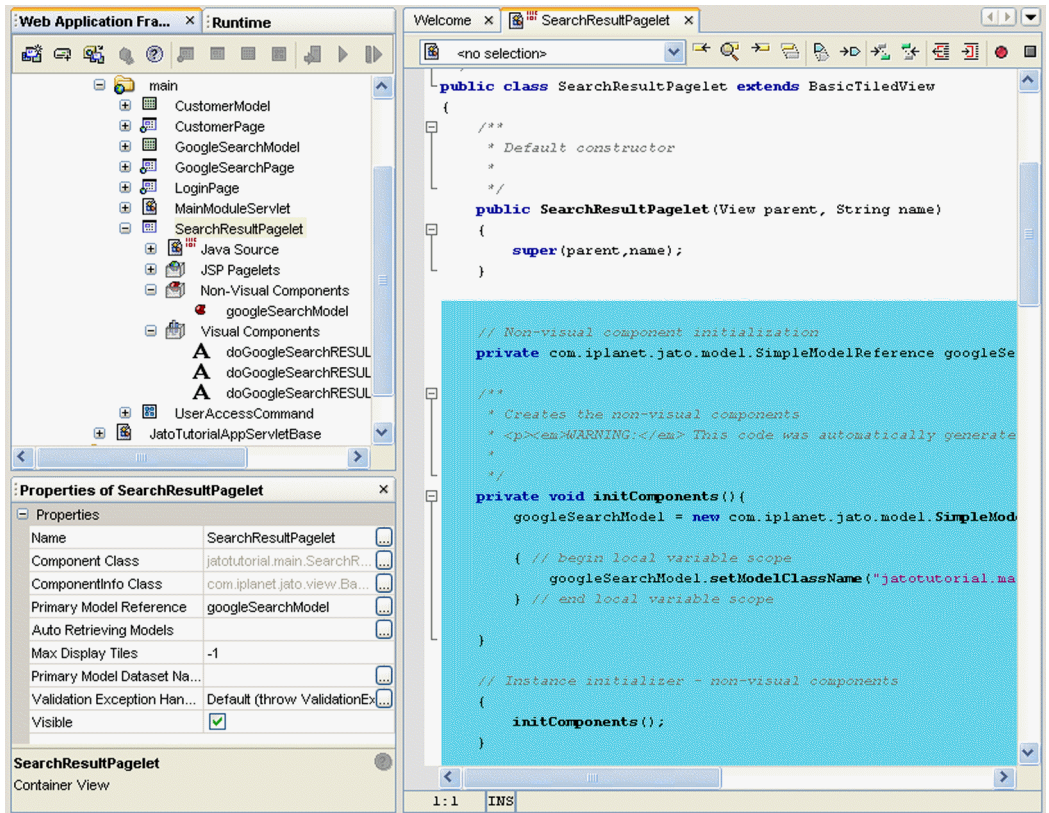


6. Expand **Current Application Components** to expose `jatotutorial/main`.
  7. Select *GoogleSearchModel*.
  8. Click **Add**.
  9. Click **Next**.
- The Bind Display Fields panel displays.



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.

---

doGoogleSearchRESULTSTitle	to title
doGoogleSearchRESULTSURL	to url
doGoogleSearchRESULTSSummary	to summary

---

# Configure the TiledView Pagelet Component

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

**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.

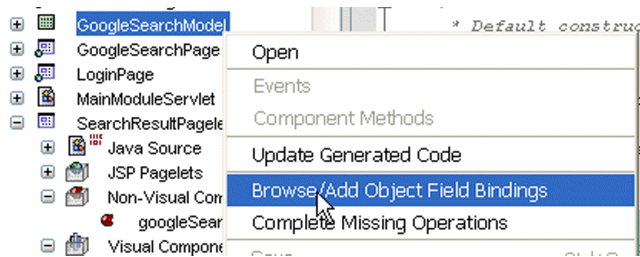
It is not imperative that you fully understand this concept, yet. To put it into simpler terms, a Web service model can have more than one result set. The Primary Model Dataset Name property just lets you specify which one to use, by default, for a particular TiledView.

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. Your worst enemy would be a typo causing some nasty runtime exceptions.

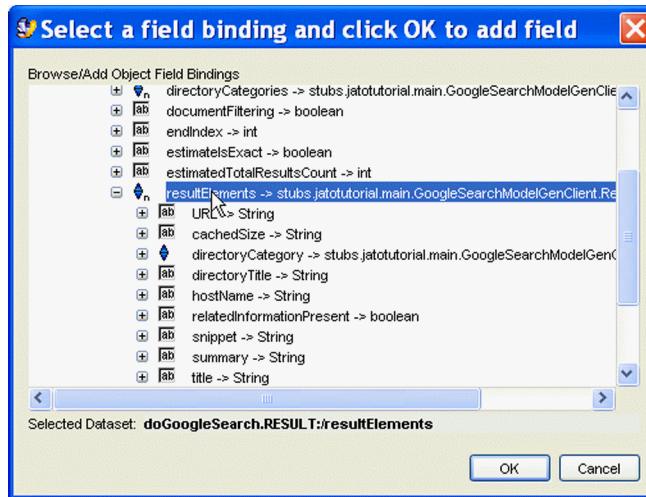
Currently, the Application Framework tools do not have a direct way to set this value by browsing the Web service, and selecting a *key path*. But there is a one-off Web service browsing technique for getting to this value so that you can copy it, then paste it into the property.

Start by selecting the GoogleSearchModel in your application.

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



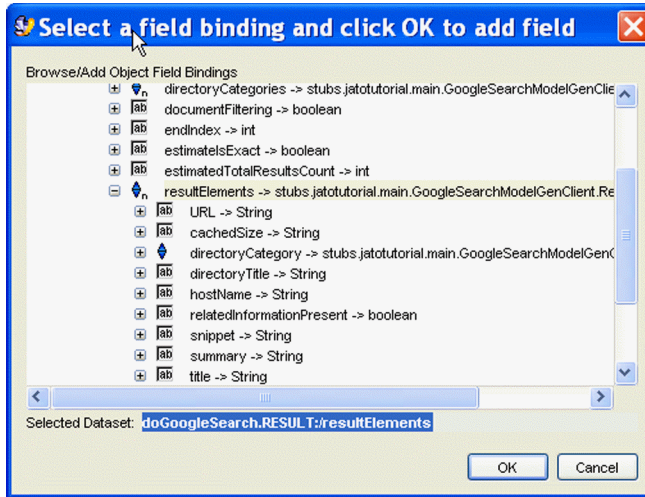
The Web Service Field Bindings editor is launched.



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 filesystems 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.

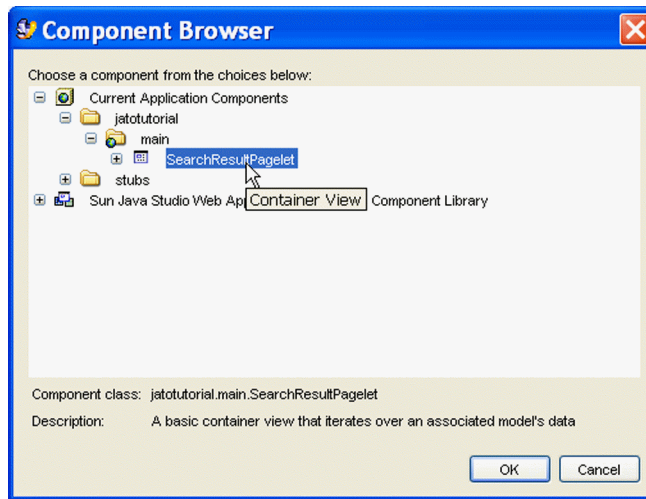
Directories (analogous to pagelets) must be contained under drives or other directories. These directories can be nested arbitrarily deep under a drive.

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.

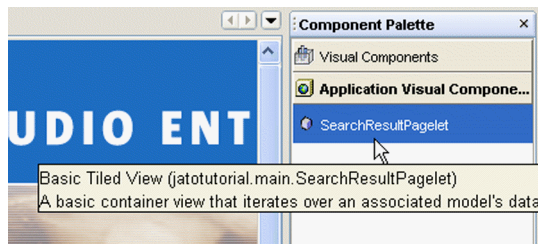


3. Expand the `Current Application Components` node.
4. Expand `jatutorial`.
5. Expand the `main` node.
6. Select the `SearchResultsPagelet TiledView` component.
7. Click `OK`.

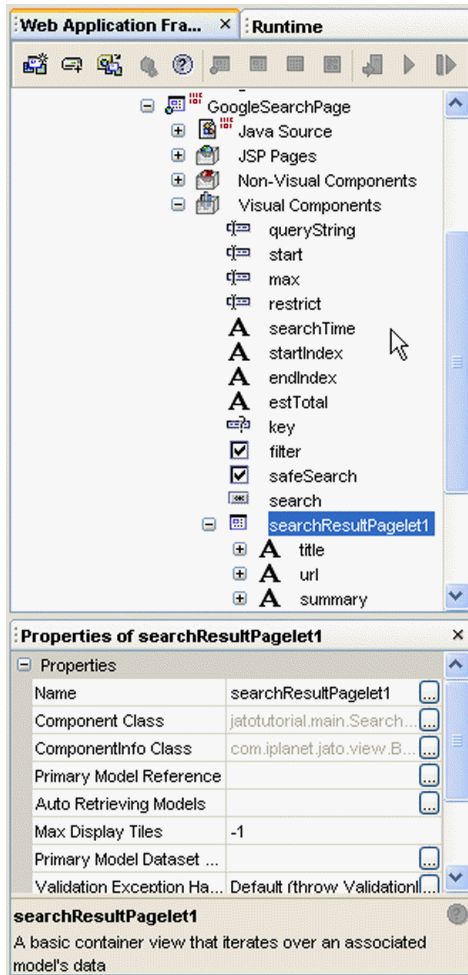
---

**Note** – The figure below shows an alternate way of adding pagelet component using the Component Palette.

---



If you select the Application Visual Components section of the Component Palette Window you will see all the Pagelet components which have been created so far in the application. For instance, SearchResultPagelet is available. Steps 2-7 may be replaced by simple clicking with your left mouse button on SearchResultPagelet in the Component Palette.



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.

---

**Note** – In “Add a TiledView” on page 135 an alternate pagelet creation option was shown. In that case we did create a new JSP fragment to be the associated JSP for the new pagelet. If you used this alternative technique then the parenting page GoogleSearchPage will not add new tags for the added pagelet but instead will only add a JSP include directive for the pagelet’s JSP fragment. Logically the addition of the pagelet is the same for both techniques but the changes to the GoogleSearchPage JSP will be different due to the existence of the pagelet’s associated JSP fragment.

---

## 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).

---

**Note** – The next two steps, code example and explanation which follows is pertinent to the case where the `SearchResultPagelet` was created without an associated JSP fragment. Because the `GoogleSearchPage.jsp` had tags automatically generated for the pagelet view child there is a lot of formatting which is needed. If you had followed the alternative to create an associated JSP fragment in [“Add a TiledView” on page 135](#), see [“Formatting the JSP and Pagelet JSP Fragment \(alternative\)” on page 149](#).

---

## Formatting the JSP and Pagelet JSP Fragment (alternative)

1. Open the page's JSP by double clicking the `GoogleSearchPage` JSP page node. Open the pagelets JSP fragment by double clicking the `SearchResultPagelet` JSP fragment node. The images of the JSP source is shown below.

```
Welcome x | JSP GoogleSearchPage x | JSP SearchResultPagelet x
</tr>
<tr>
<th align=right valign=middle>Filter:</th>
<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>

<table>
<tr>
<th>Results</th>
</tr>

<jato:tableView name="searchResultPagelet1">
<%@include file="/jatutorial/main/SearchResultPagelet.jsp"%>
</jato:tableView>
</table>

</body>
</html>

</jato:useViewBean>

69:17 INS
```

```
Welcome x | JSP GoogleSearchPage x | JSP SearchResultPagelet x
<%@taglib prefix="jato" uri="/WEB-INF/jato.tld"%>
<jato:pagelet>
<tr>
<td align=left valign=middle>
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"/>
</td>
</tr>
</jato:pagelet>
```

If you recall in [“Add a TiledView” on page 135](#), when we created the SearchResultPagelet we not only provided for the creation of a JSP fragment to be associated with the new pagelet but also selected automatically formatting checkbox. See image above. Notice that the GoogleSearchPage has been automatically updated with HTML table tags and table header tags and the searchResultPagelet1 tileview has automatically been added including the directive for the pagelets JSP fragment. You may accept the auto-formatted changes as is or you may alter both JSP files as shown to make the search result look even better.





## Tutorial—Section 4.5

# Test Run the Google Search Page

---

This chapter describes how to run your Web 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 Java System Application Server, you must Deploy the application when changes are made.
- 2. Select the Web Application Framework Application node (JatoTutorial), and click the Deploy button on the Web 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.

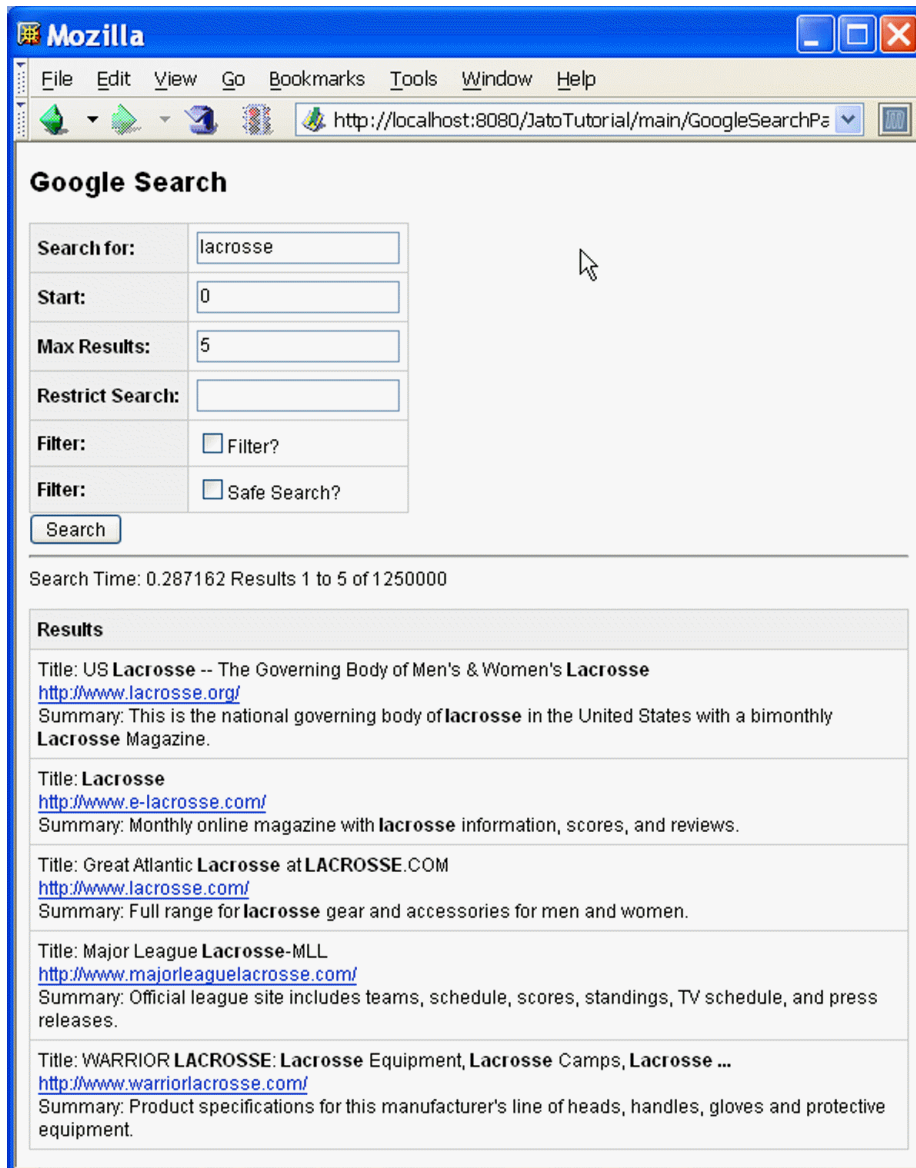
The results portion of the page initially have zeroes for values. The search returns values for those fields and a list of five links that satisfy the query string.

### 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, 35
- Add JDBC Datasource option, 52
- Add Model button, 57
- Add Page button, 65
- Advanced Tip - Modules, 34
- Alternative runtime environments, 50
- Application Framework applications, how organized, 20
- application layout, observe, 32
- Application Location panel, 28
- Application Name folder, select, 47
- application pages, link, 51
- Application Properties panel, 29
- Application Servlet, 33
- application servlet not required, 34
- application servlet, JatoTutorialAppServletBase, super class, 33
- Application, Run the, 89
- application, tutorial, about, 22
- Associate JSP panel, 37, 66
- assumptions for this tutorial, 17
- audience for this tutorial, 17
- Auto Retrieving Models combo box, 76
- Auto Retrieving Models custom editor, 76
- Auto Retrieving Models property, 76

## B

- base component list, Basic ViewBean option, 36
- base directory, default, 29

- Basic ViewBean option, select, 36
- basics covered in tutorial, 17
- Before You Start the Tutorial, 15 to 16
- Bind Display Fields panel, 68
- Bound fields list box, 69
- Button Command Descriptor, configure, 96
- Button Component, add, 73

## C

- Choose Template panel, 28
- Columns node of the CustomerModel, 63
- Command Component, create, 91
- Command Descriptor editor, 73
- Command Descriptor property, set, 74
- Component Palette, 73
- Create an Application Wizard, 27
- Current Application Components, 68
- Customer Page, add hidden field, 77
- Customer Page, create, 65
- Customer Page, test run, 83
- customer record, enable button to update, 73
- Customer Update, test, 84
- CustomerMode, Create, 57
- CustomerModel object, created in main module, 61
- CustomerPage, 72
- CustomerPage node, 75

## D

- deployment step required, 48
- Design-Time Resources folder, 52

directory location for new application, 27

Display Fields, add to Login page, 39

## E

events, module servlet, about, 33

execute Method, add code to, 94

Execute Page (Redeploy) button, 48

Execute Page button, 48

## F

Features, Web Application Framework, 15

first page of application, create, 35

front controller servlet, 34

## G

Getting Started, 17, 17 to 22

Google Search Page, create, 113

Google Search Page, test run, 133

Google Search Page, test run, with results, 153

Google Web service software SDK, download, 107

## H

handleLoginRequest Method in LoginPage, edit, 85

HREF Command Descriptor, configure, 100

HREF tag in Customer JSP, format, 102

HREF, add to a Customer page, 99

## I

infrastructure, need to create, 27

## J

J2EE application, description, 20

J2EE component, description, 19

J2EE module, description, 19

J2EE Web application, description, 19

JDBC datasource panel, 52

JDBC Datasources, 52

JDBC Datasources node, 52

JDBC Datasources, create additional, 52

JDBC SQL Model, create, 57

JDBC URL, makes connection to database, 56

JSP Content, format, 130

JSP, format the, 80

JSP, formatting, 147

## L

location, directory, for new application, 27

login name, invalid, 49

Login Page, create, 35

Login Page, link to Customer Page, 85

Login Page, test run, 47, 48

Login, test a successful, 49

Login/Logout Command, test run, 105

LoginPage, 48

LoginPage node, 39

## M

main module folder, 57

MainModuleServlet, 34

Manual Code Technique, 125

Model Associations panel, 67

Model Auto Retrieve, making, 75

Model Field Properties tab in property sheet, 63

Model's Key Field(s), mark, 62

Module Properties panel, 30

Module Servlet, 34

module servlet hierarchy, can be customized, 34

module, only one in this application, 34

## N

New App Directory field, 29

New datasource name textbox, 53

Non-JNDI Enabled Containers, add connection code for, 63

Non-Visual Components node, 74

## O

Operation Name property, 74

## P

Page Component, add, 113

Pagelet, add to a page, 144

Point & Click Technique (code-free), 125

PointBase driver, 50

Preface, 9 to 14

Primary Model Dataset Name, getting correct, 142

## R

RDBMS database, assumption, 51

Rename option, 40  
RequestHandler property, 73  
Runtime Environments, alternative, 50

## S

Sample Database, connect to, 51  
Search Button, enable, 124  
Search, try a, 134, 153  
Section 4.1, Prepare to Create a Web Service Model, 107 to 111  
Section 4.2, Create the Google Search Page, 113 to 131  
Section 4.3, Test Run the Google Search Page, 133 to 134  
Section 4.4, Add Results Listing to the Google Search Page, 135 to 149  
Section 4.5, Test Run the Google Search Page, 153 to 154  
Select connection combo box, 53  
Select Database Tables page, 59  
Select Datasource page, 58  
Select Model Type panel, 57  
Select Table Columns page, 60  
Select View Type panel, 36, 66  
servlet classes created, 32  
Settings & Configuration folder, 52  
SQL Database, accessing, 51  
SQL-based model, add, 51  
SQL-based model, add page to display data, 51  
Static Text Field option, 39  
success message, 49  
successful login, test, 49  
Sun Java Studio editor display, 38  
super class, application servlet,  
    JatoTutorialAppServletBase, 33

## T

TiledView Pagelet Component, configure, 142  
TiledView Pagelet, create, 135  
TiledView, add, 135  
Tomcat (and other non-JNDI containers) SQL Connection Preparation, 55  
tutorial basics, 17  
Tutorial Sections (Links to), 23 to 25

Tutorial, about, 22  
tutorial, goal of, 18  
Tutorial-Section 1.1, Application Infrastructure, 27 to 34  
Tutorial-Section 1.2, Create Login Page, 35 to 45  
Tutorial-Section 1.3, Test Run the Login Page, 47 to 50  
Tutorial-Section 2.1, Prepare Application to Access SQL Database, 51 to 56  
Tutorial-Section 2.2, Create the CustomerModel, 57 to 64  
Tutorial-Section 2.3, Create Customer Page, 65 to 80  
Tutorial-Section 2.4, Test Run the Customer Page, 83 to 84  
Tutorial-Section 2.5, Link Login Page to Customer Page, 85 to 87  
Tutorial-Section 2.6, Run Application, 89 to 90  
Tutorial-Section 3.1, Create a Command Component, 91 to 98  
Tutorial-Section 3.2, Add a Logout Link to the Customer Page, 99 to 103  
Tutorial-Section 3.3, Test Run the Login/Logout Command Component, 105 to 106

## U

Unsuccessful Login, test, 49  
Use formatting to beautify fields on JSP - option, 37  
Use formatting to beautify fields on JSP check box, 67  
UserAccessCommand Component, create, 91

## V

ViewBean - created, 70  
ViewBean, add, 65  
Visual Components node, 39  
Visual Components, add more to page, 119

## W

Web Application Framework  
    Primary Features, 15  
Web Application Framework Application, new, 27  
Web Application, compile, 47  
Web Service Model, create, 108  
Web Service SDK, download, 107

Web Service User Registration and  
Downloading, 107  
Web Service, register to use, 108