The NetBeans E-commerce Tutorial - Introduction

Tutorial Contents

Welcome to the NetBeans E-commerce Tutorial. In this multi-part tutorial, you learn how to create a simple yet effective e-commerce application that demonstrates various important features of Java web and EE development. In doing so, you'll familiarize yourself with the NetBeans IDE and become capable of applying it to your own development purposes.

Taking the time to master the IDE will ultimately lead you to become more efficient and versatile as a developer. While you work through the tutorial units, you'll learn how to make best use of the IDE's facilities and tools. These include:

- **Editor support for different languages:** syntax highlighting, code completion, API documentation support, keyboard shortcuts, refactoring capabilities, and code templates
- **Window system:** Projects, Files and Services windows, the Tasks window,

Javadoc window, HTTP Monitor, Navigator and Palette
- **Integration with other services:** automatic deployment to a registered server, database connectivity, browser interoperability
- **Development tools:** Debugger, Profiler, HTTP Server Monitor, Local History support, and a graphical Diff Viewer

The tutorial is modular in fashion, with each unit focusing on specific concepts, technologies, and features of the IDE. You can successfully follow a tutorial unit on its own using the provided setup instructions and application snapshots (from Unit 5 onward). However, you'll get the most benefit by working through all units consecutively, from beginning to end. This will also help to illustrate the development process.

Unit 3, Setting up the Development Environment introduces you to the NetBeans IDE. In it, you create a Java web project which is the basis for the work you undertake in later tutorial units. In Unit 4, Designing the Data Model, you primarily work with MySQL WorkBench, a visual database design tool, to create a data model for the application. Each successive tutorial unit provides you with a *project snapshot* that corresponds to the end state of the previous unit. This enables you to work through a single tutorial unit outside of the E-commerce Tutorial's larger context. To use these snapshots, download them to your computer and open them in the IDE using the Open Project wizard (Ctrl-Shift-O; ⌘-Shift-O on Mac).

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

The remainder of this unit covers some information relevant to the tutorial, as well as basic concepts necessary for Java EE development. Make sure you understand the concepts outlined below before proceeding with development.

About this Tutorial

Who this Tutorial is for

The content of this tutorial caters to four demographics:

- Java developers interested in expanding their skill set to include Java EE technologies
- Newcomers to the NetBeans IDE wanting to try out its development environment
- Web developers wanting to see how Java compares to other web-based technologies
- Students wanting to understand the nuts and bolts a simple e-commerce application, and how its development could apply to a real-world use-case

If you fall into any of these categories, this tutorial will be helpful to you. Depending on your background, you may find that certain tutorial units are more difficult to grasp than others.

Understanding how technologies work is key to leveraging the IDE for your purposes. Therefore, if you are really interested in learning the technologies involved, you may find that this tutorial works best as a companion to the Java EE Tutorial. For each tutorial unit, make best use of the provided links to relevant areas in the Java EE Tutorial, as well as to other useful resources.

What this Tutorial Covers

The application that you develop in this tutorial involves numerous concepts, technologies, and tooling components:

- **Concepts**
  - Front-end development
  - Web application project structure
  - Data modeling
  - Database connectivity
  - Object-relational mapping
  - Session management
  - Transactional business logic
  - Client and server-side validation
  - Localization
  - Web application security
  - Design patterns, including Model-View-Controller (MVC) and Session Facade
- **Technologies**
  - HTML, CSS, and JavaScript technologies
  - Servlet and JavaServer Pages (JSP) technologies
  - Enterprise JavaBeans (EJB) technology
  - Java Persistence API (JPA)
  - The JavaServer Pages Standard Tag Library (JSTL)
  - Java Database Connectivity (JDBC)
- **Development Tools**
  - NetBeans IDE
  - GlassFish, a Java EE application server
  - MySQL, a relational database management server (RDBMS)
  - MySQL WorkBench, a visual database design tool

What is an E-commerce Application?

The term *e-commerce*, as we think of it today, refers to the buying and selling of goods or services over the Internet. For example, you may think of Amazon, which provides online shopping for various product categories, such as books, music, and electronics. This form of e-commerce is known as electronic retailing, or *e-tailing*, and usually involves the transportation of physical items. It is also referred to as *business-to-customer*, or B2C. Other well-known forms include:

- **Consumer-to-consumer (C2C):** Transactions taking place between individuals, usually through a third-party site such as an online auction. A typical example of C2C commerce is eBay.
- **Business-to-business (B2B):** Trade occurring between businesses, e.g., between a retailer and wholesaler, or between a wholesaler and manufacturer.
- **Business-to-government (B2G):** Trade occurring between businesses and government agencies.

This tutorial focuses on business-to-customer (B2C) e-commerce, and applies the typical scenario of a small retail store seeking to create a website enabling customers to shop online. Software that accommodates a B2C scenario generally consists of two components:

1. **Store Front:** The website that is accessed by customers, enabling them to purchase goods over the Internet. Data from the store catalog is typically maintained in a database, and pages requiring this data are generated dynamically.
2. **Administration Console:** A password-protected area that is accessed over a secure connection by store staff for purposes of online management. This typically involves CRUD (create read update delete) access to the store catalog, management of discounts, shipping and payment options, and review of customer orders.

What is Java?

In the computer software industry, the term "Java" refers to the *Java Platform* as well as the *Java Programming Language*.

Duke, the Java mascot

Java as a Programming Language

The Java language was conceptualized by James Gosling, who began work on the project in 1991. The language was created with the following 5 design principles[1] in mind:

1. **Simple, Object-Oriented, and Familiar:** Java contains a small, consistent core of fundamental concepts that can be grasped quickly. It was originally modeled after the

then popular C++ language, so that programmers could easily migrate to Java. Also, it adheres to an *object-oriented* paradigm; systems are comprised of encapsulated objects that communicate by passing messages to one another.
2. **Robust and Secure:** The language includes compile-time and run-time checking to ensure that errors are identified quickly. It also contains network and file-access security features so that distributed applications are not compromised by intrusion or corruption.
3. **Architecture Neutral and Portable:** One of Java's primary advantages is its *portability*. Applications can be easily transferred from one platform to another with minimum or no modifications. The slogan "Write once, run anywhere" accompanied the Java 1.0 release in 1995, and refers to the cross-platform benefits of the language.
4. **High Performance:** Applications run quickly and efficiently due to various low-level features, such as enabling the Java interpreter to run independently from the run-time environment, and applying an automatic garbage collector to free unused memory.
5. **Interpreted, Threaded, and Dynamic:** With Java, a developer's source code is compiled into an intermediate, interpreted form known as *bytecode*. The bytecode instructional set refers to the machine language used by the Java Virtual Machine (JVM). With a suitable interpreter, this language can then be translated into *native code* for the platform it is run on. Multithreading capabilities are supported primarily by means of the `Thread` class, enabling numerous tasks to occur simultaneously. The language and run-time system are dynamic in that applications can adapt to environment changes during execution.

If you'd like to learn more about the Java language, see the Java Tutorials.

Java as a Platform

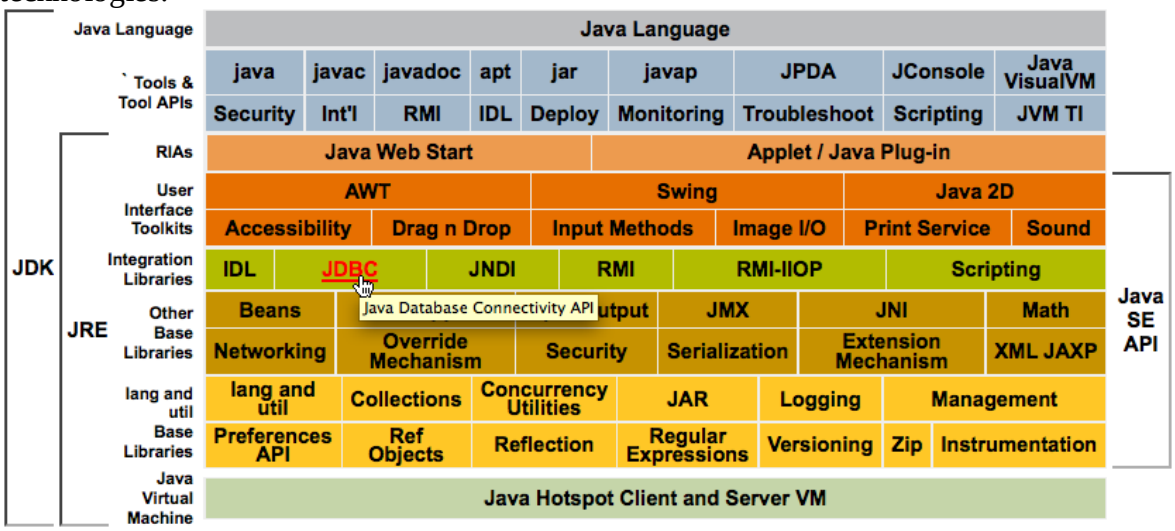
The Java Platform signifies a software-based platform that is comprised of two parts:

- **The Java Virtual Machine (JVM)**: The JVM is an engine that executes instructions generated by the Java compiler. The JVM can be thought of as an instance of the Java Runtime Environment, or JRE, and is embedded in various products, such as web browsers, servers, and operating systems.
- **The Java Application Programming Interface (API)**: Prewritten code, organized into packages of similar topics. For instance, the Applet and AWT packages include classes for creating fonts, menus, and buttons.

The Java Development Kit, or JDK, refers to the Java SE Edition, while other kits are referred to as "SDK", a generic term for "software development kit." For example, the Java EE SDK.[2]

You can see a visual representation of the Java platform by viewing the conceptual diagram of component technologies provided in the JDK Documentation. As shown below, the diagram is interactive, enabling you click on components to learn more about individual

technologies.



As the diagram indicates, the JDK includes the Java Runtime Environment (JRE). You require the JRE to run software, and you require the JDK to develop software. Both can be acquired from Java SE Downloads.

The Java platform comes in several *editions*, such as Java SE (Standard Edition), Java ME (Micro Edition), and Java EE (Enterprise Edition).

Java EE

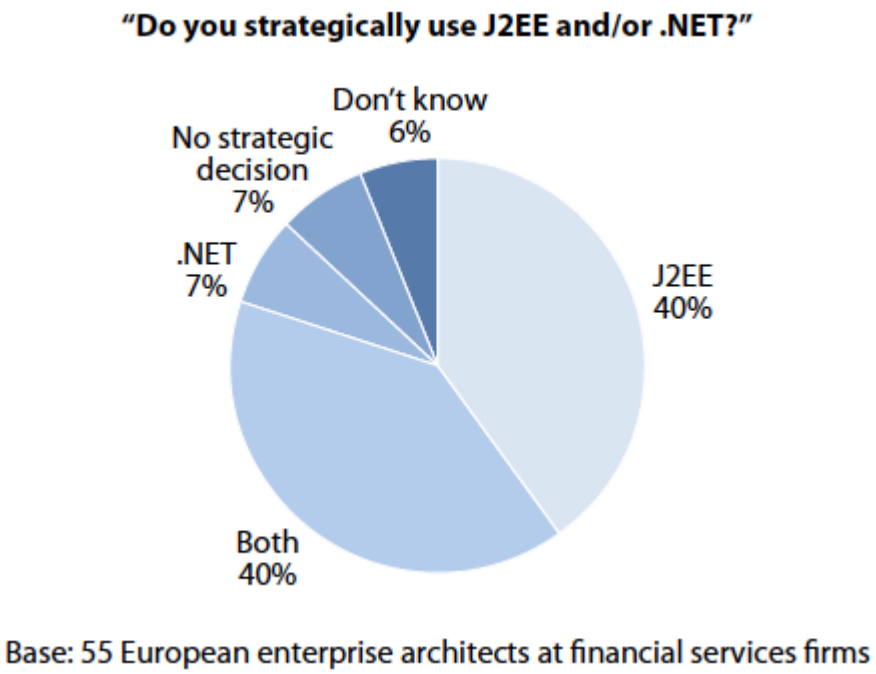
The Java Platform, Enterprise Edition (Java EE) builds upon the Java SE platform and provides a set of technologies for developing and running portable, robust, scalable, reliable and secure server-side applications.

EE technologies are loosely divided into two categories:

- Web application technologies
- Enterprise application technologies

Depending on your needs, you may want to use certain technologies from either category. For example, this tutorial makes use of Servlet, JSP/EL, and JSTL "web" technologies, as well as EJB and JPA "enterprise" technologies.

Java EE currently dominates the market, especially in the financial sector. The following diagram is taken from an independent survey for European markets performed in 2007.



For a recent, informal comparison of Java EE to .NET, see the blog post Java EE or .NET - An Almost Unbiased Opinion by a well-known member of the Java EE community.

What's the Difference Between...?

There are many abbreviations and acronyms to parse. If you're new to all of this and find the above explanation somewhat confusing, the following resources can help explain what the differences are between some of the commonly used terminology.

- What's the Difference between the JRE and the JDK?
- What's the Difference between the JRE and the Java SE platform?
- What's the Difference between Java EE and J2EE?
- Unraveling Java Terminology

What is the Java Community Process?

The Java Community Process (JCP) is a program that manages the development of standard technical specifications for Java technology. The JCP catalogs Java Specification Requests (JSRs), which are formal proposals that document the technologies which are to be added to the Java platform. JSRs are run by an *Expert Group*, which typically comprises representatives of companies that are stakeholders in the industry. The JCP enables Java technology to grow and adapt according to the needs and trends of the community.

The JSRs of technologies used and referred to in this tutorial include the following:

- JSR 52: A Standard Tag Library for JavaServer Pages

- JSR 245: JavaServer Pages 2.1
- JSR 315: Java Servlet 3.0
- JSR 316: Java Platform, Enterprise Edition 6
- JSR 317: Java Persistence 2.0
- JSR 318: Enterprise JavaBeans 3.1

You can use the JCP website to search for individual JSRs. You can also view all current EE technologies (Java EE 6) at:

- http://java.sun.com/javaee/technologies/index.jsp

Java EE 5 technologies are listed at:

- http://java.sun.com/javaee/technologies/javaee5.jsp

A JSR's final release provides a *reference implementation*, which is a free implementation of the technology. In this tutorial, you utilize these implementations to develop the sample e-commerce application. For example, the GlassFish v3 application server, which is included in the standard Java download bundle for NetBeans 6.8, is the reference implementation of the Java EE 6 platform specification (JSR 316). As a reference implementation for the Java EE platform, it includes reference implementations for the technologies included in the platform, such as Servlet, EJB and JPA technologies.

Why use an IDE?

Firstly, the term *IDE* stands for *integrated development environment*. The purpose of an IDE has traditionally been to maximize a developer's productivity by providing tools and support such as:

- a source code editor
- a compiler and build automation tools
- a window system for viewing projects and project artifacts
- integration with other commonly-used services
- debugging support
- profiling support

Consider what would be necessary if you wanted to create a Java-based web application manually. After installing the Java Development Kit (JDK), you would need to set up your development environment by performing the following steps.[3]

1. Set your `PATH` environment variable to point to the JDK installation.
2. Download and configure a server that implements the technologies you plan to use.
3. Create a development directory where you plan to create and work on the web application(s). Furthermore, you are responsible for setting up the application directory structure so that it can be understood by the server. (For example, see Java BluePrints: Strategy for Web Applications for a recommended structure.)

4. Set your `CLASSPATH` environment variable to include the development directory, as well as any required JAR files.
5. Establish a deployment method, i.e., a way to copy resources from your development directory to the server's deployment area.
6. Bookmark or install relevant API documentation.

For educative purposes, it is worthwhile to create and run a Java web project manually so that you are aware the necessary steps involved. But eventually, you'll want to consider using tools that reduce or eliminate the need to perform tedious or repetitious tasks, thereby enabling you to focus on developing code that solves specific business needs. An IDE streamlines the process outlined above. As demonstrated in Unit 3, Setting up the Development Environment, you'll install NetBeans IDE with the GlassFish application server, and be able to set up a web application project with a conventional directory structure using a simple 3-step wizard. Furthermore, the IDE provides provides built-in API documentation which you can either call up as you code in the editor, or maintain open in an external window.

An IDE also typically handles project compilation and deployment in a way that is transparent to you as a developer. For example, the web project that you create in NetBeans includes an Ant build script that is used to compile, clean, package and deploy the project. This means that you can run your project from the IDE, and it will automatically be compiled and deployed, then open in your default browser. Taking this a step further, many IDEs support a Deploy on on Save feature. In other words, whenever you save changes to your project, the deployed version on your server is automatically updated. You can simply switch to the browser and refresh the page to view changes.

IDEs also provide templates for various file types, and often enable you to add them to your project by suggesting common locations and including default configuration information where necessary.

Aside from the "basic support" described above, IDEs typically provide interfaces to external tools and services (e.g., application and database servers, web services, debugging and profiling facilities, and collaboration tools) which are indispensable to your work if Java development is your profession.

Finally, IDEs usually provide enhanced editor support. The editor is where you likely spend most of your time working, and IDE editors typically include syntax highlighting, refactoring capabilites, keyboard shortcuts, code completion, hints and error messages, all aiming to help you work more efficiently and intelligently.

Why use NetBeans?

The NetBeans IDE is a free, open-source integrated development environment written entirely in Java. It offers a range of tools for create professional desktop, enterprise, web, and mobile applications with the Java language, C/C++, and even scripting languages such as PHP, JavaScript, Groovy, and Ruby.

People are saying great things about NetBeans. For a list of testimonials, see NetBeans IDE Testimonials. Many developers are migrating their applications to NetBeans from other IDEs. For reasons why, read Real Stories From People Switching to NetBeans IDE.

The IDE provides many features for web development, and several advantages over other IDEs. Here are several noteworthy points:

- **Works Out of the Box:** Simply download, install, and run the IDE. With its small download size, installation is a breeze. The IDE runs on many platforms including Windows, Linux, Mac OS X and Solaris. All IDE tools and features are fully integrated - no need to hunt for plug-ins - and they work together when you launch the IDE.
- **Free and Open Source:** When you use the NetBeans IDE, you join a vibrant, open source community with thousands of users ready to help and contribute. There are discussions on the NetBeans project mailing lists, blogs on Planet NetBeans, and helpful FAQs and tutorials on the community wiki.
- **Profiling and Debugging Tools:** With NetBeans IDE profiler, you get real time insight into memory usage and potential performance bottlenecks. Furthermore, you can instrument specific parts of code to avoid performance degradation during profiling. The HeapWalker tool helps you evaluate Java heap contents and find memory leaks.
- **Customizable Projects:** Through the NetBeans IDE build process, which relies on industry standards such as Apache Ant, make, Maven, and rake - rather than a proprietary build process - you can easily customize projects and add functionality. You can build, run, and deploy projects to servers outside of the IDE.
- **Collaboration Tools:** The IDE provides built-in support for version control systems such as CVS, Subversion, and Mercurial.
- **Extensive Documentation:** There's a wealth of tips and instructions contained in the IDE's built-in help set. Simply press F1 (fn-F1 on Mac) on a component in the IDE to invoke the help set. Also, the IDE's official knowledge base provides hundreds of online tutorials, articles and screencasts that are continuously being updated.

For a more extensive list of reasons why you should consider choosing NetBeans, see NetBeans IDE Connects Developers.

The NetBeans E-commerce Tutorial - Designing the Application

Tutorial Contents

3. Setting up the Development Environment
4. Designing the Data Model
5. Preparing the Page Views and Controller Servlet
6. Connecting the Application to the Database
7. Adding Entity Classes and Session Beans
8. Managing Sessions
9. Integrating Transactional Business Logic
10. Adding Language Support (Coming Soon)
11. Securing the Application (Coming Soon)
12. Load Testing the Application (Coming Soon)
13. Conclusion

The application that you design in this tutorial is based on a real-world scenario. After being introduced to the tutorial scenario, you consolidate a high-level list of customer requirements. You then prepare a diagram of the application's business process flow, and a series of *mockups* which help both you and your customer get a clearer picture of how the final application will look to an end-user. Finally, you break down the customer requirements into a set of implementation tasks, and structure your application so that the responsibilities and interactions among functional components are clearly defined.

This tutorial unit discusses the MVC (Model-View-Controller) design pattern. After investigating the benefits that this pattern offers, you set about mapping JSP, Servlet, and other technologies to the MVC architecture, and draft a diagram that illustrates the components of the application in terms of MVC.

This unit makes various references to Designing Enterprise Applications with the J2EE Platform, Second Edition. This book contains guidelines promoted by Java BluePrints.

Although this tutorial unit does not require use of the NetBeans IDE, it is essential because it lays the groundwork for tasks that will be covered in the following units.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

The Scenario

This tutorial is based on the following scenario. Although this is a fictitious scenario, it demonstrates how the software you are about to develop can be applied to real-world

business needs. It also serves as a platform from which you can derive customer requirements. Customer requirements should be established as clearly as possible before any design or implementation begins.

The Scenario

A small grocery store, the Affable Bean, collaborates with several local farms to supply a community with organic produce and foods. Due to a long-standing customer base and increasing affluence to the area, the store has decided to investigate the possibility of providing an online delivery service to customers. A recent survey has indicated that 90% of its regular clientele has continuous Internet access, and 65% percent would be interested in using this service.

The grocery store staff have asked you, the Java web developer, to create a website that will enable their customers to shop online. They have also asked that you create an administration console alongside the website, which will allow staff members to keep track of orders.

The store's location is in Prague, in the Czech Republic. Because regular clientele are both English and Czech-speaking, staff have requested that the website support both languages.

The grocery store has already purchased a domain, and have indicated that one web-oriented staff member is able to deploy the application to the domain once it is ready.

Gathering Customer Requirements

The initial phase of any project involves gathering information before making any design or implementation decisions. In its most common form, this involves direct and frequent communication with a customer. Based on the provided scenario, the Affable Bean staff have communicated to you that the application you are to create should fulfill the following requirements:

1. An online representation of the products that are sold in the physical store. There are four categories (dairy, meats, bakery, fruit & veg), and four products for each category, which online shoppers can browse. Details are provided for each product (i.e., name, image, description, price).
2. Shopping cart functionality, which includes the ability to:

- add items to a virtual shopping cart.
- remove items from the shopping cart.
- update item quantities in the shopping cart.
- view a summary of all items and quantities in the shopping cart.
- place an order and make payment through a secure checkout process.

3. An administration console, enabling staff to view customer orders.
4. Security, in the form of protecting sensitive customer data while it is transferred over the Internet, and preventing unauthorized access to the administration console.
5. Language support for both English and Czech. (Website only)

The company staff are able to provide you with product and category images, descriptions and price details, as well as any website graphics that are to be used. The staff are also able to provide all text and language translations for the website.

There are many practices and methods devoted to software development management. Agile software development is one methodology that encourages frequent customer inspection, and places importance on adaptation during the development cycle. Although this is outside the scope of this tutorial, each tutorial unit concludes with a functional piece of software that could be presented to a customer for further communication and feedback.

Preparing Mockups

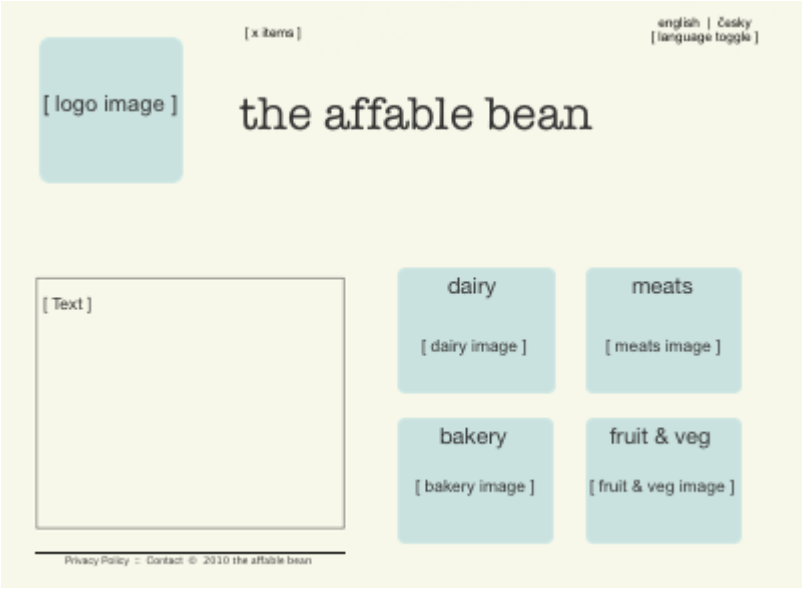
After gathering customer requirements, you work with the Affable Bean staff to gain a clearer picture of how they expect the website to look and behave. You create a use-case that describes how the application will be used and encapsulates its behavior:

Use-Case

Customer visits the welcome page and selects a product category. Customer browses products within the selected category page, then adds a product to his or her shopping cart. Customer continues shopping and selects a different category. Customer adds several products from this category to shopping cart. Customer selects 'view cart' option and updates quantities for cart products in the cart page. Customer verifies shopping cart contents and proceeds to checkout. In the checkout page, customer views the cost of the order and other information, fills in personal data, then submits his or her details. The order is processed and customer is taken to a confirmation page. The confirmation page provides a unique reference number for tracking the customer order, as well as a summary of the order.

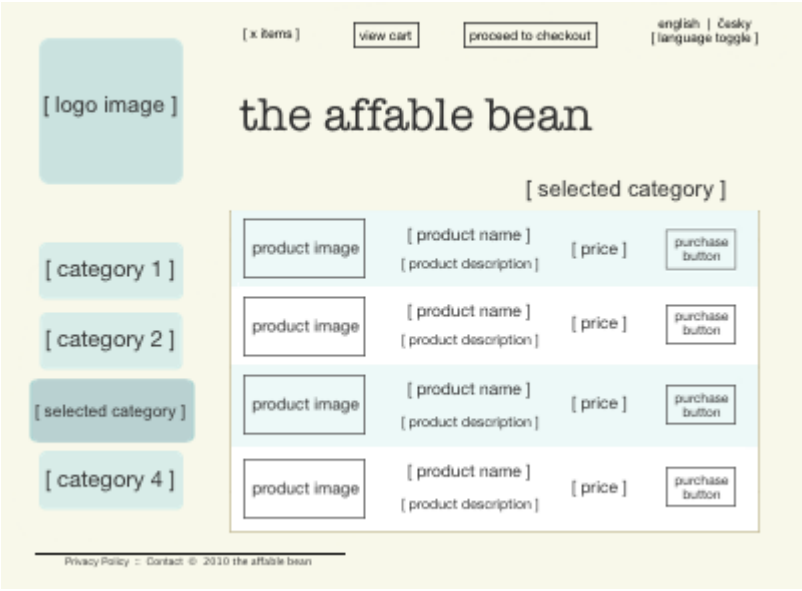
You also begin creating mockups. There are numerous ways to go about this task. For example, you could use storyboard software, or create a set of wireframes to relay the relationships between pages. Another common method is known as paper prototyping, where you collaborate with the customer by sketching ideas on paper.

In this scenario, we've produced *mockups* of the primary pages the user expects see when navigating through the website. When we later discuss the MVC design pattern, you'll note that these pages map to the *views* used by the application.



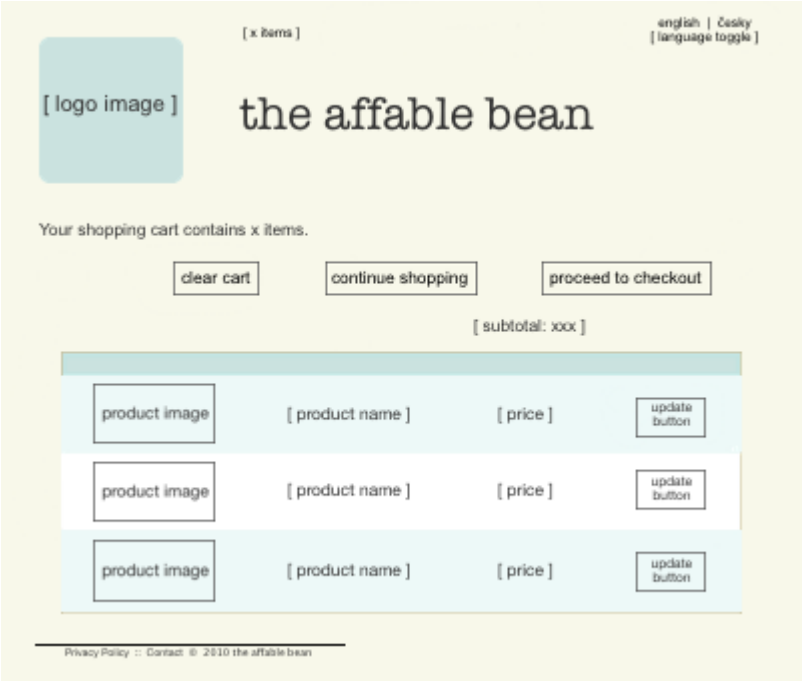
welcome page

The welcome page is the website's home page, and entry point for the application. It introduces the business and service to the user, and enables the user to navigate to any of the four product categories.



category page

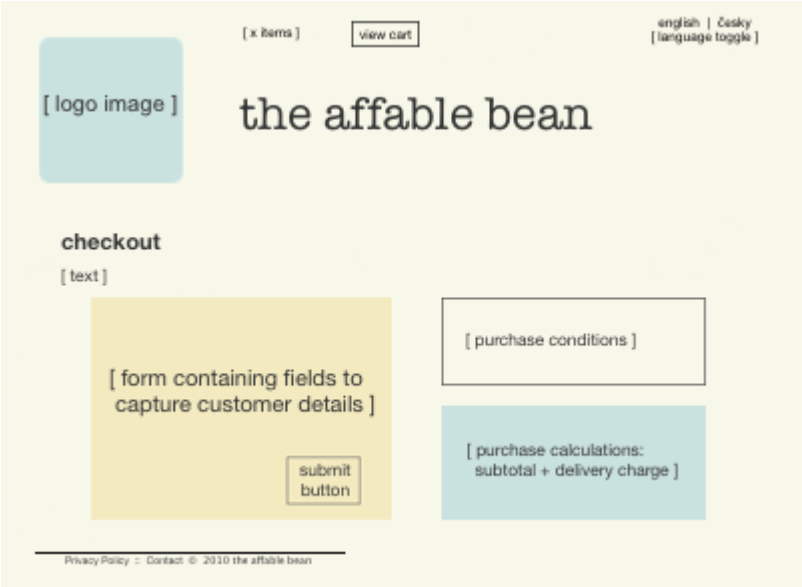
The category page provides a listing of all products within the selected category. From this page, a user is able to view all product information, and add any of the listed products to his or her shopping cart. A user can also navigate to any of the provided categories.



cart page

The cart page lists all items held in the user's shopping cart. It displays product details for each item, and tallies the subtotal for the items in the cart. From this page, a user can:

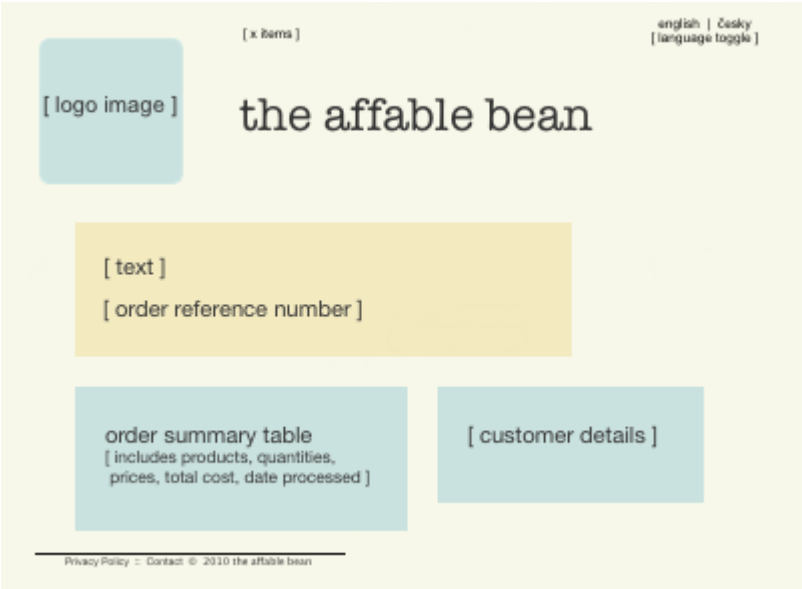
- Clear all items in his or her cart (Clicking 'clear cart' causes the 'proceed to checkout' buttons and shopping cart table to disappear.)
- Update the quantity for any listed item (The price and quantity are updated; the subtotal is recalculated. If user sets quantity to '0', the product table row is removed.)
- Return to the previous category by clicking 'continue shopping'
- Proceed to checkout



checkout page

The checkout page collects information from the customer using a form. This page also displays purchase conditions, and summarizes the order by providing calculations for the total cost.

The user is able to send personal details over a secure channel.



confirmation page

The confirmation page returns a message to the customer confirming that the order was successfully recorded. An order reference number is provided to the customer, as well as a summary listing order details.

Order summary and customer personal details are returned over a secure channel.

Also, you agree with staff on the following rules, which apply to multiple pages:

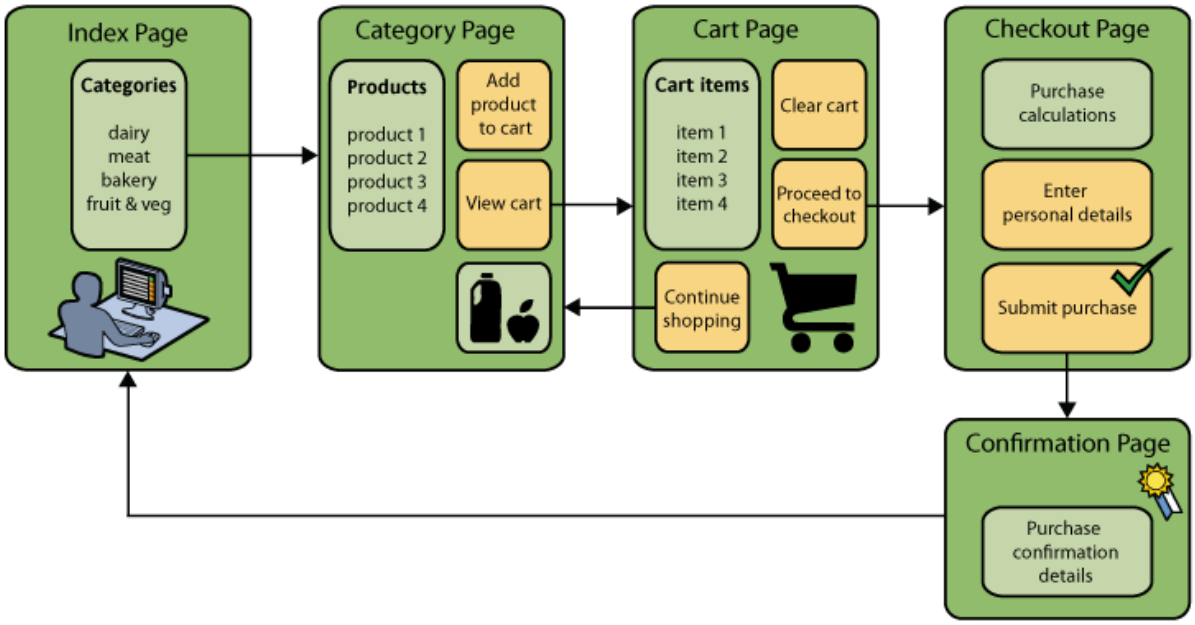
- The user is able to proceed to checkout from any page, provided that:
  - The shopping cart is not empty
  - The user is not already on the checkout page
  - The user has not already checked out (i.e., is on the confirmation page)
- From all pages, the user is able to:
  - Select the language (English or Czech) to view the page in
  - View the status of his or her shopping cart (if it is not empty)
  - Return to the welcome page by clicking the logo image

Note: Although not presented here, you would equally need to work with the client to produce use-cases and mockups, and establish rules for the administration console. The NetBeans E-commerce Tutorial focuses on developing the store front (i.e., the website). However, Unit 11, Securing the Application demonstrates how to create a login mechanism to access the administration console. Also, you can examine the provided implementation of the administration console by downloading the completed application.

The Business Process Flow

To help consolidate the relationships between the proposed mockups and better illustrate the functionality that each page should provide, you prepare a diagram that demonstrates the process flow of the application.

The diagram displays the visual and functional components of each page, and highlights the primary actions available to the user in order to navigate through the site to complete a purchase.



Determining the Architecture

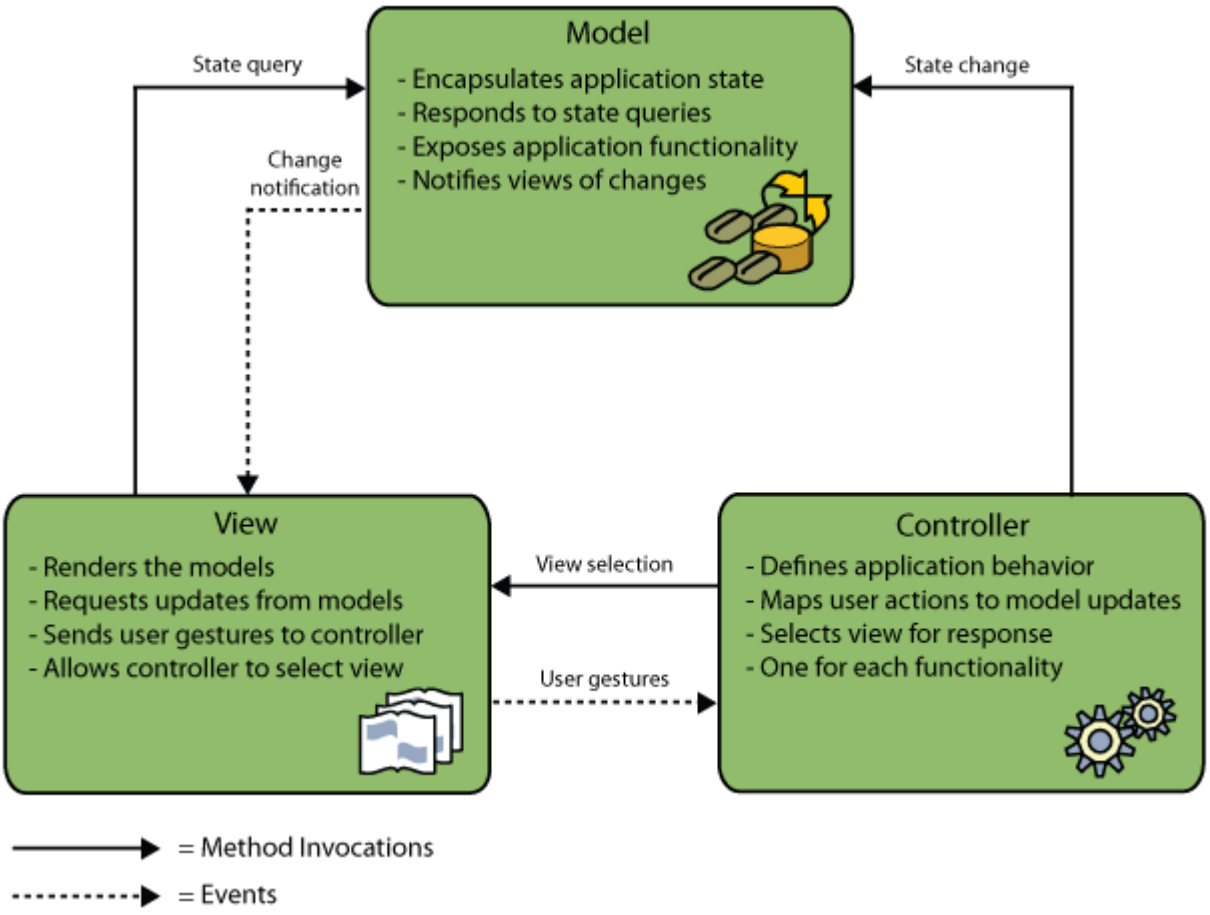
Before you start coding, let's examine the ways in which you can architect the project. Specifically, you need to outline the responsibilities among functional components, and determine how they will interact with each other.

When you work with JSP technologies, you can code all of your business logic into JSP pages using scriptlets. Scriptlets are snippets of Java code enclosed in `<% %>` tags. As you may already be aware, JSP pages are compiled into servlets before they are run, so Java code is perfectly valid in JSP pages. However, there are several reasons why this practice should be avoided, especially when working in large projects. Some reasons are outlined in Designing Enterprise Applications with the J2EE Platform, Second Edition as follows:[1]

- **Scriptlet code is not reusable:** Scriptlet code appears in exactly one place: the JSP page that defines it. If the same logic is needed elsewhere, it must be either included (decreasing readability) or copied and pasted into the new context.
- **Scriptlets mix logic with presentation:** Scriptlets are islands of program code in a sea of presentation code. Changing either requires some understanding of what the other is doing to avoid breaking the relationship between the two. Scriptlets can easily confuse the intent of a JSP page by expressing program logic within the presentation.
- **Scriptlets break developer role separation:** Because scriptlets mingle programming and Web content, Web page designers need to know either how to program or which parts of their pages to avoid modifying.
- **Scriptlets make JSP pages difficult to read and to maintain:** JSP pages with scriptlets mix structured tags with JSP page delimiters and Java language code.
- **Scriptlet code is difficult to test:** Unit testing of scriptlet code is virtually impossible. Because scriptlets are embedded in JSP pages, the only way to execute them is to execute the page and test the results.

There are various design patterns already in existence which provide considerable benefits when applied. One such pattern is the MVC (Model-View-Controller) paradigm, which divides your application into three interoperable components:[2]

- **Model:** Represents the business data and any business logic that govern access to and modification of the data. The model notifies views when it changes and lets the view query the model about its state. It also lets the controller access application functionality encapsulated by the model.
- **View:** The view renders the contents of a model. It gets data from the model and specifies how that data should be presented. It updates data presentation when the model changes. A view also forwards user input to a controller.
- **Controller:** The controller defines application behavior. It dispatches user requests and selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a web application, user inputs are HTTP GET and POST requests. A controller selects the next view to display based on the user interactions and the outcome of the model operations.

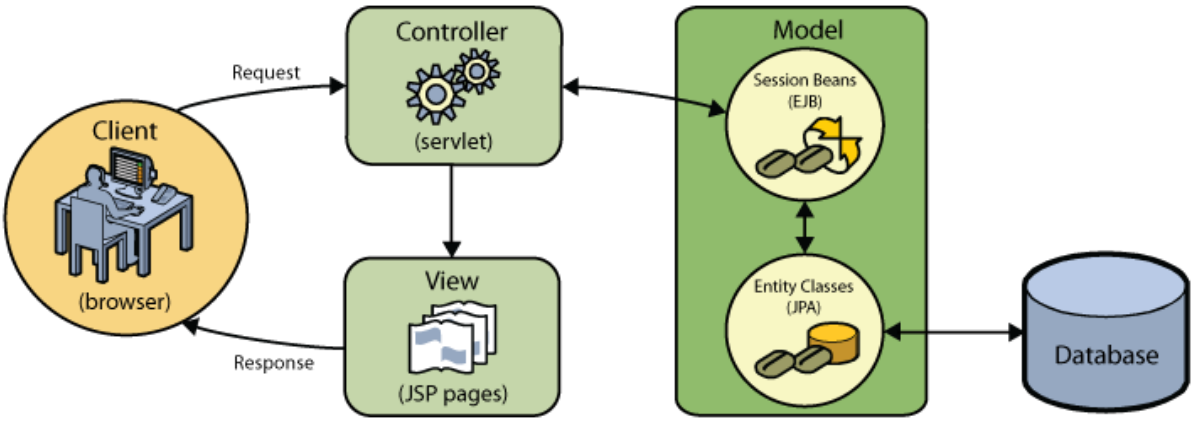


Adhering to the MVC design pattern provides you with numerous benefits:

- **Separation of design concerns:** Because of the decoupling of presentation, control, and data persistence and behavior, the application becomes more flexible; modifications to one component have minimal impact on other components. You can, for example, create new views without needing to rewrite the model.
- **More easily maintainable and extensible:** Good structure can reduce code complexity. As such, code duplication is minimized.
- **Promotes division of labor:** Developers with different skill sets are able to focus on their core skills and collaborate through clearly defined interfaces.

Note: When JSP technology was first introduced in 1999, the early specifications included a description of two model architectures: Model 1 and Model 2. Model 1 involves implementing business logic directly within JSP pages, whereas Model 2 applies the MVC pattern. For more information on Model 1 and Model 2 architectures, see Designing Enterprise Applications with the J2EE Platform, section 4.4.1: Structuring the Web Tier.

You can apply the MVC pattern to the application that you develop for the Affable Bean client. You can use a servlet as a *controller* to handle incoming requests. The pages from the business process flow diagram can be mapped to *views*. Finally, the business data, which will be maintained in a database, can be accessed and modified in the application using EJB session beans with JPA entity classes. These components represent the *model*.



Planning the Project

In order to plan the project, you need to extrapolate functional tasks from the customer requirements. The tasks that we produce will structure the implementation plan for the project, and form the outline for tutorial units that follow. In practice, the more capable you are of identifying tasks and the work they entail, the better you'll be able to stick to the schedule that you and your customer agree upon. Therefore, begin with a high-level task list, then try to drill down from these tasks dividing each task into multiple sub-tasks, and possibly dividing sub-tasks further until each list item represents a single unit of work.

- ± Set up the development environment
- ± Prepare the data model for the application
- ± Create front-end project files
- ± Organize the application front-end
- ± Create a controller servlet
- ± Connect the application to the database
- ± Develop the business logic
- ± Add language support
- ± Create administration console
- ± Secure the application

The NetBeans E-commerce Tutorial - Setting up the Development Environment

Tutorial Contents

- See Also
  4. Designing the Data Model
  5. Preparing the Page Views and Controller Servlet
  6. Connecting the Application to the Database
  7. Adding Entity Classes and Session Beans
  8. Managing Sessions
  9. Integrating Transactional Business Logic
  10. Adding Language Support (Coming Soon)
  11. Securing the Application (Coming Soon)
  12. Load Testing the Application (Coming Soon)
  13. Conclusion

The following steps describe how to set up your development environment. In the process, you'll learn about some of the primary windows of the IDE and understand how the IDE uses an Ant build script to perform common actions on your project. By the end of this tutorial unit, you'll have created a web application project, and confirmed that you can successfully build the project, deploy it to your development server, and run it from the IDE.

You also learn how to connect the IDE to a MySQL database server, create database instances, and connect to database instances from the IDE's Services window. In this unit, you create a new database named `affablebean`, which you will use throughout the tutorial.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
GlassFish server	v3 or Open Source Edition 3.0.1
MySQL database server	version 5.1

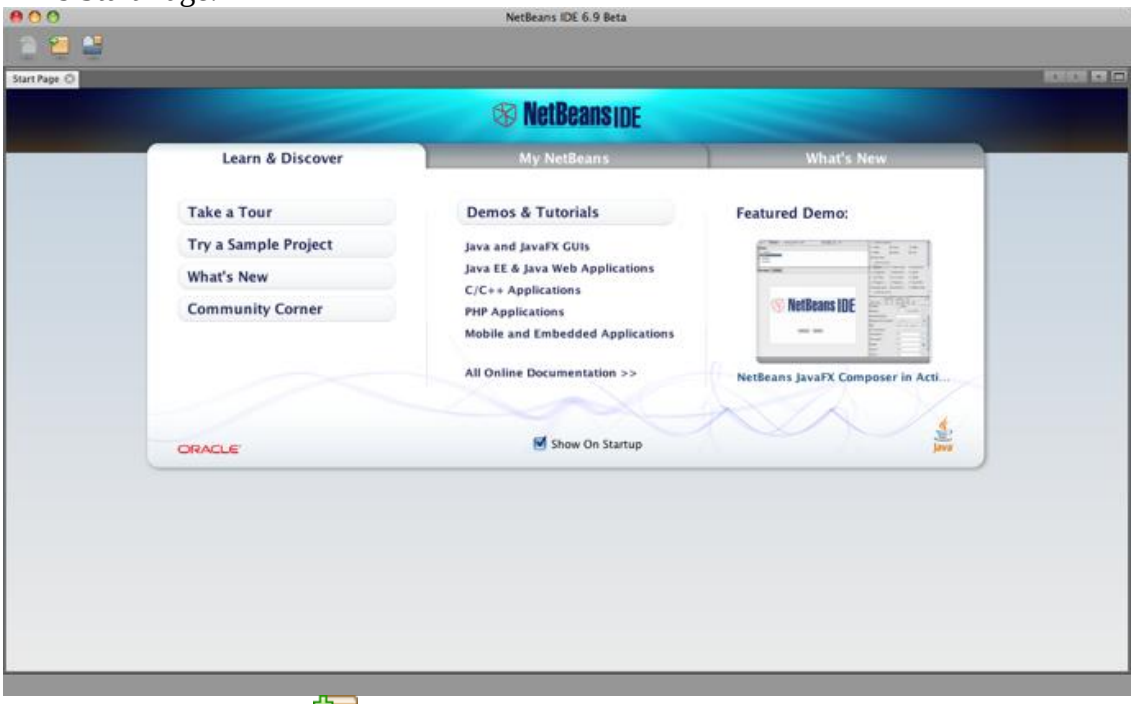
Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.
- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could download the GlassFish server independently, but the

version provided with the NetBeans download has the added benefit of being automatically registered with the IDE.

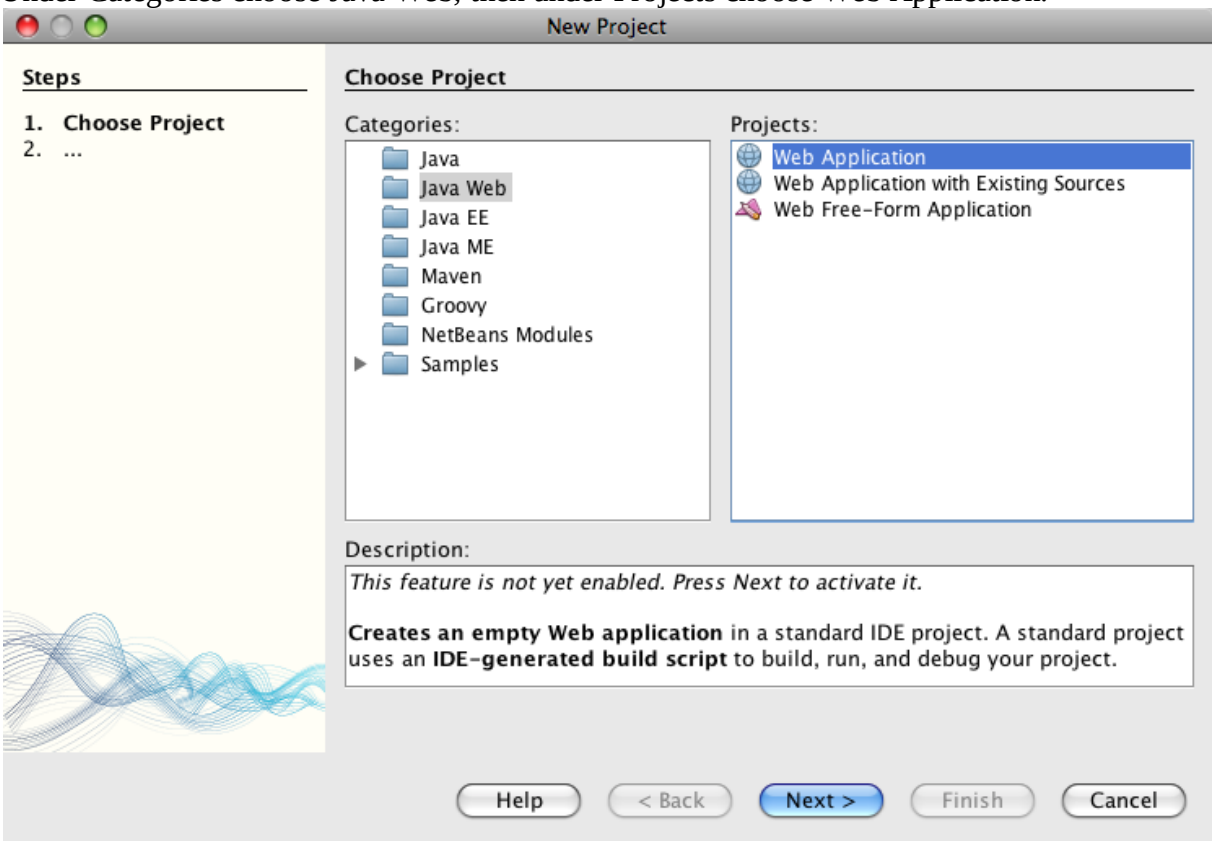
Creating a Web Project

1. Start the NetBeans IDE. If you are running the IDE for the first time, you will see the IDE's Start Page.



2. Click the New Project ( ) button (Ctrl-Shift-N; ⌘-Shift-N on Mac) to create a new Java web project. The New Project wizard opens to guide you through the process.

Under Categories choose Java Web, then under Projects choose Web Application.



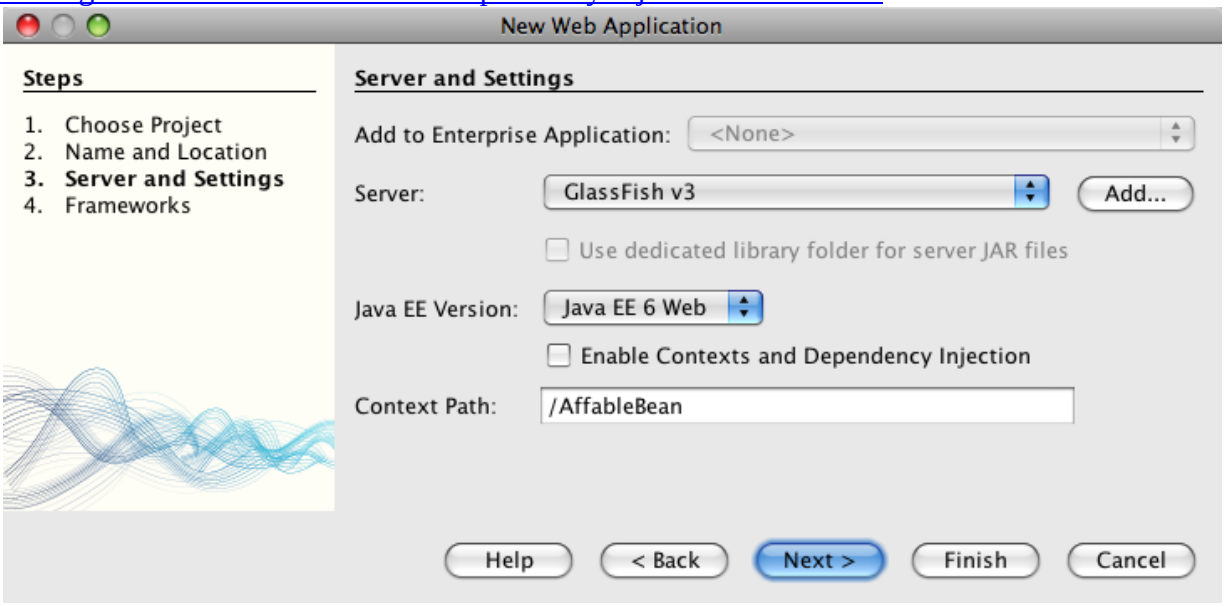
3. Click Next.
4. In Step 2: Name and Location, name the project `AffableBean`. In this step, you can also designate the location on your computer where the project will reside. By default, the IDE creates a `NetBeansProjects` folder in your home directory. If you'd like to change the location, enter the path in the Project Location text field.
5. Click Next.
6. In Step 3: Server and Settings, specify GlassFish v3 as the server to which your project will be deployed during development. Since you've included GlassFish v3 in your NetBeans installation, you'll see that GlassFish v3 is listed in the Server drop-down field.

   If you wanted to deploy to a server that isn't yet registered with the IDE, you would click the Add button, and step through the Add Server Instance wizard. You can view all servers registered with the IDE from the Servers window (Choose Tools > Servers from the main menu).

7. For Java EE Version, select Java EE 6 Web.

   The application that you create makes use of various Java EE 6 features, namely servlet annotations (new in the Servlet 3.0 specification), and EJBs used directly in servlet containers (new in the EJB 3.1 specification). Both Servlet 3.0 and EJB 3.1 are part of the Java EE 6 platform, therefore you require an EE-6 compliant server such as GlassFish v3 to work through this tutorial. For more information, see About Specifications and Implementations.
8. Make sure that the 'Enable Contexts and Dependency Injection' option is deselected. This option is specific to the Contexts and Dependency Injection (CDI) technology,

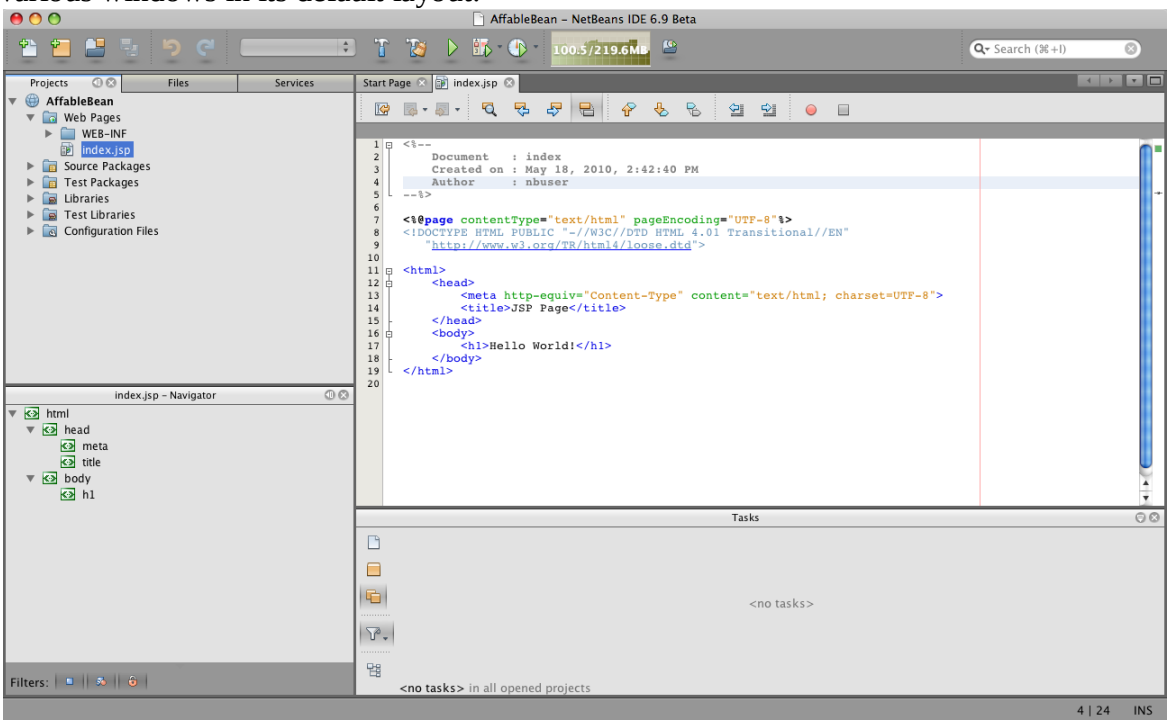
specified by JSR-299, and is not used in this tutorial. For more information, see Getting Started with Contexts and Dependency Injection and JSF 2.0.



Note that by default the context path for the application is the name of the project. This is the path at which your application can be accessed after it is deployed to the server. For example, GlassFish uses 8080 as its default port number, so during development you'll be able to access the project in a browser window from:

```
http://localhost:8080/AffableBean/
```

9. Click Finish. The IDE generates a skeleton project named `AffableBean` that adheres to the J2EE Blueprints conventions for web application structure. The IDE displays various windows in its default layout.



10. Examine the IDE's default layout. Here's a brief overview of the displayed windows and tabs:

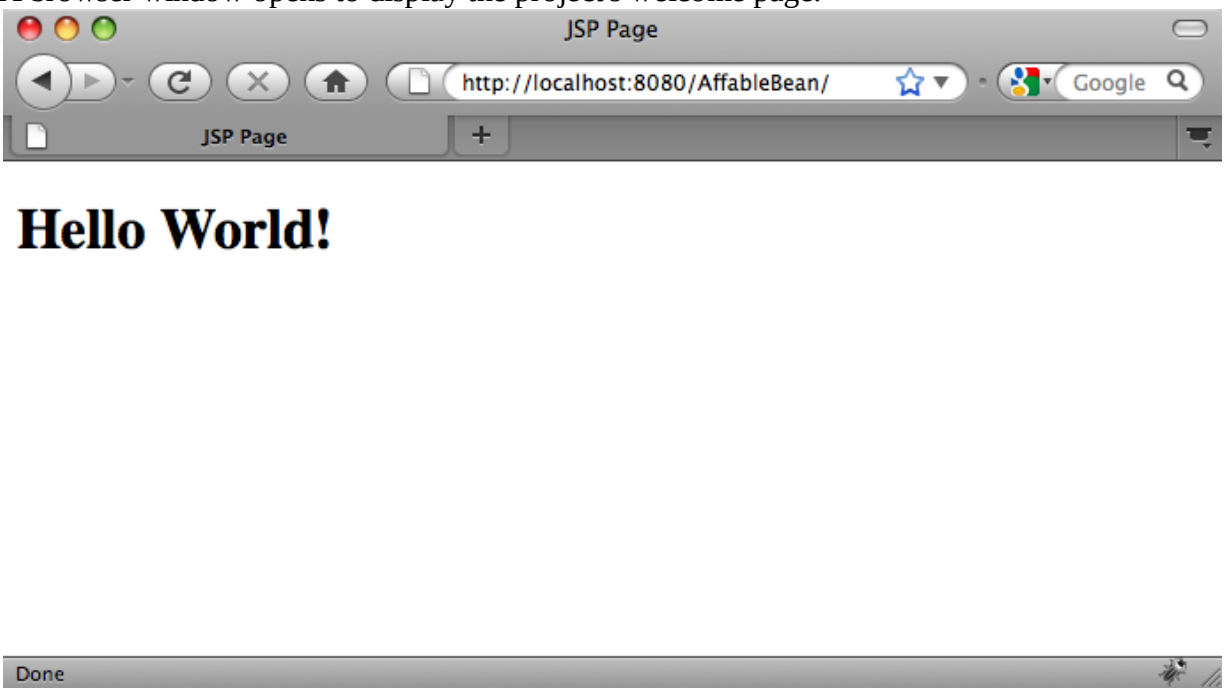
- **The Editor:** The editor (Ctrl-0; ⌘-0 on Mac) is the central component of the IDE, and is likely where you'll spend most of your time. The editor automatically adapts to the language you are working in, providing documentation support, code-completion, hints and error messages specific to the technology you are coding in.
- **Projects window:** The Projects window (Ctrl-1; ⌘-1 on Mac) is the entry point to your project sources. It provides a *logical view* of important project contents, and groups files together based on their function (e.g., `Configuration Files`). When right-clicking file nodes within the Projects window, you can call actions common to your development tasks (i.e., `Build`, `Clean`, `Deploy`, `Run`).
- **Files window:** The Files window (Ctrl-2; ⌘-2 on Mac) provides a directory-based view of your project. That is, it enables you to view the structure of your project, as it exists in your computer's file system. From this window, you can view all files pertaining to your project, including the Ant build script, (`build.xml`), and files required by the IDE to handle the project (contained in the `nbproject` folder). If you've built your project, you can see the location of compiled Java files (`build` folder), and the project's distributable WAR file (contained in the `dist` folder).
- **Navigator:** The Navigator (Ctrl-7; ⌘-7 on Mac) provides a structural overview of the file opened in the editor. For example, if an HTML web page is displayed, the Navigator lists tag nodes in a way that corresponds to the page's Document Object Model (DOM). If a Java class is opened in the editor, the Navigator displays the properties and methods pertaining to that class. You can use the Navigator to navigate to items within the editor. For example, when you double-click a node in the Navigator, your cursor is taken directly to that element in the editor.
- **Tasks window:** The Tasks window (Ctrl-6; ⌘-6 on Mac) automatically scans your code and lists lines with compile errors, quick fixes, and style warnings. For Java classes, it also lists commented lines containing words such as '`TODO`' or '`FIXME`'.
- **Services window:** The Services window (Ctrl-5; ⌘-5 on Mac) provides an interface for managing servers, web services, databases and database connections, as well as other services relating to team development.
- **Output window:** *(Not displayed)* The Output window (Ctrl-4; ⌘-4 on Mac) automatically displays when you call an action that invokes a service, generally from an outside resource such as a server, and can mirror server log files. With web projects, it also enables you to view information related to Ant tasks (e.g., `Build`, `Clean and Build`, `Clean`).
- **Palette:** *(Not displayed)* The Palette (Ctrl-Shift-8; ⌘-Shift-8 on Mac) provides various handy code snippets that you can drag and drop into the editor. Many of the snippets included in the Palette are also accessible by invoking code completion in the editor, as will later be demonstrated.

Note: All of the IDE's windows can be accessed from the Window menu item.

Running the Web Project

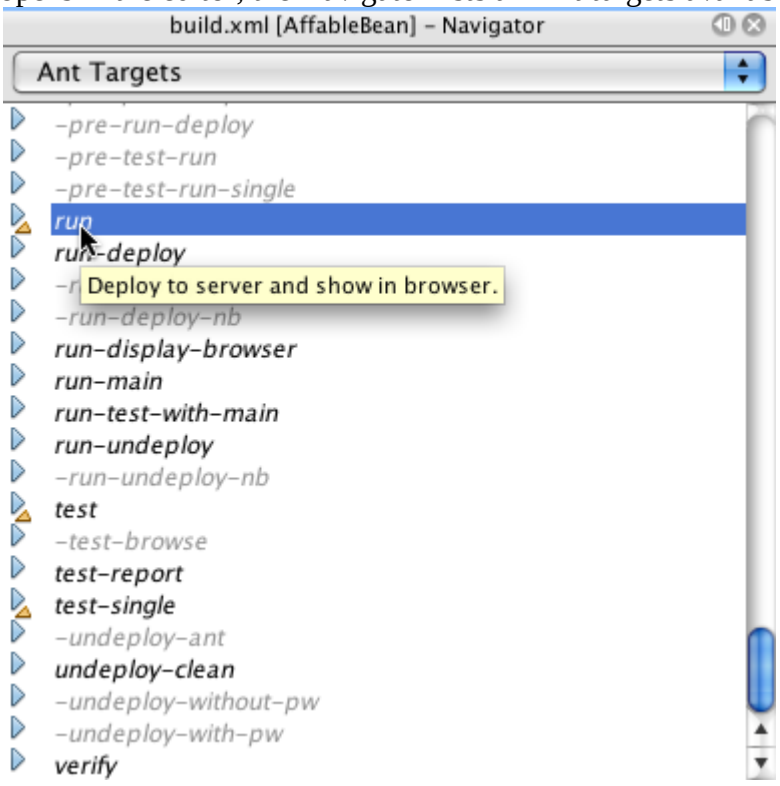
1. Run the new `AffableBean` project. In the Projects window, you can do this by right-clicking the project node and choosing Run, otherwise, click the Run Project () button (F6; fn-F6 on Mac) in the IDE's main toolbar.

A browser window opens to display the project's welcome page.



So what just happened? When you run a web project, the IDE invokes the `run` Ant target in your project's build script. You can investigate by opening your project's `build.xml` file in the editor.

2. Switch to the Files window (Ctrl-2; ⌘-2 on Mac), expand the project node and double-click the `build.xml` file contained in your project. When the `build.xml` file opens in the editor, the Navigator lists all Ant targets available to the script.



Normal Ant targets are displayed using the general target ( ) icon. The *emphasized* Ant target ( ) icon merely indicates that the target includes a description, which is displayed as a tooltip (as shown in the above image). For more information, see Creating, Importing, and Configuring Java Projects.

3. Double-click the `run` target. The `build-impl.xml` file opens in the editor and displays the target definition.

```
<target depends="run-deploy,run-display-browser" description="Deploy
to server and show in browser." name="run"/>
```

Why did the `build-impl.xml` file open when we clicked on a target from `build.xml`? If you switch back to `build.xml` (press Ctrl-Tab) and examine the file contents, you'll see the following line:

```
<import file="nbproject/build-impl.xml"/>
```

The project's build script is basically an empty file that imports NetBeans-defined targets from `nbproject/build-impl.xml`.

You can freely edit your project's standard `build.xml` script by adding new targets or overriding existing NetBeans-defined targets. However, you should not edit the `build-impl.xml` file.

You can see from the `run` target's definition that it depends on the following targets:

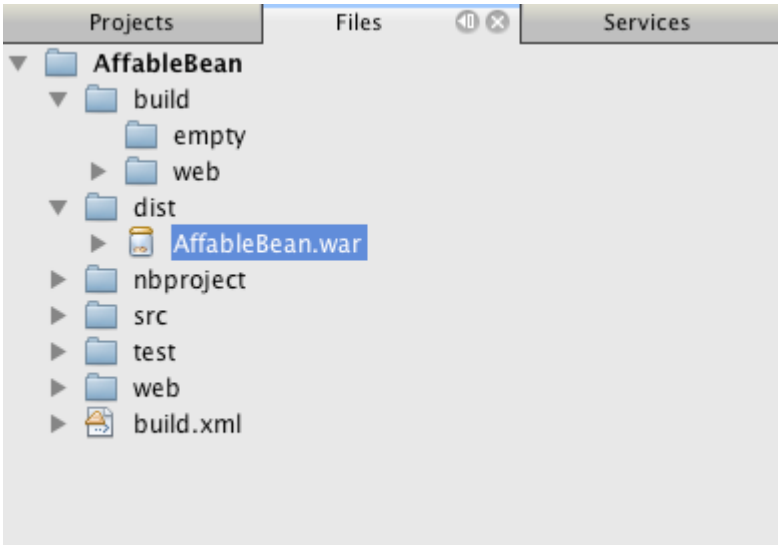
- `run-deploy`
- `run-display-browser`

Both of these targets in turn depend on other targets, which you can examine elsewhere in the `build-impl.xml` file. But essentially, the following actions take place when the `run` target is invoked:

3. The project gets compiled.
4. A WAR file is created.
5. The server starts (if it is not already running).
6. The WAR file gets deployed to the designated server.
7. The browser opens to display the server's URL and application's context path.

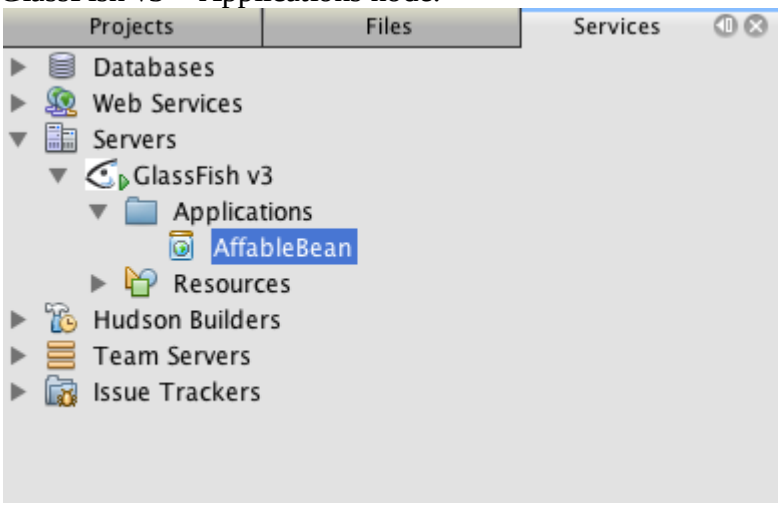
Consult the official Ant Manual for more information on using Ant.

In the Files window (Ctrl-2; ⌘-2 on Mac) expand the project node. The `build` folder contains your compiled project. The `dist` folder contains the project WAR file that was deployed to GlassFish.



Note: If you *clean* the project (In the Projects window, choose Clean from the project node's right-click menu), both of these folders are removed.

Switch to the Services window (Ctrl-5; ⌘-5 on Mac) and expand the Servers > GlassFish v3 > Applications node.



The green arrow icon on the server node indicates that the server is running. The Applications folder lists all deployed applications; you can see that the `AffableBean` application has been successfully deployed.

At this stage, you've created a Java web project in the IDE, and have confirmed that it can be successfully built and deployed to your development server, and opened in a browser when run.

Communicating with the Database Server

Once you've downloaded and installed the MySQL database server, you can connect to it from the IDE. A default installation uses '`root`' and '' (an empty string) as the user account and password to connect to the database server. However, due to connectivity issues with GlassFish, it is recommended that you use an account with a non-empty password.[1] The

following instructions demonstrate how to run the database server and change the password for the `root` account to '`nbuser`' from the MySQL command-line. The '`root`' / '`nbuser`' combination is used throughout the NetBeans E-commerce Tutorial. With the database server running and properly configured, you register it in the IDE and create a database instance.

Note: If you require assistance with installation, refer to the official MySQL documentation: General Installation Guidance.

Check if the MySQL Server is Running

Before connecting to the MySQL server from the IDE, you need to make sure the server is running. One way to do this is by using the `mysqladmin` client's `ping` command.

1. Open a command-line prompt and navigate to the `bin` directory within your MySQL installation.

   ```
   shell> cd <install-dir>/bin
   ```

   (Where *<install-dir>* is the path to your MySQL installation directory.)

2. Type in the following:

   ```
   shell> mysqladmin ping
   ```

   If the server is running, you will see output similar to the following:

   ```
   mysqld is alive
   ```

   If the server is not running, you'll see output similar to the following:

   ```
   mysqladmin: connect to server at 'localhost' failed
   error: 'Can't connect to local MySQL server through socket
   '/tmp/mysql.sock'
   Check that mysqld is running and that the socket: '/tmp/mysql.sock'
   exists!
   ```

Start the Database Server

In the event that your MySQL server is not running, you can start it from the command-line. See Starting and Stopping MySQL Automatically for a brief, cross-platform overview. The following steps provide general guidance depending on your operating system.

Unix-like systems:

For Unix-like systems, it is recommended to start the MySQL server by invoking `mysqld_safe`.

1. Open a command-line prompt and navigate to the `bin` directory within your MySQL installation.

   For example, if you have installed the MySQL database server at `/usr/local/mysql`, enter the following at your `shell>` prompt:

   ```
   shell> cd /usr/local/mysql/bin
   ```

2. Run the `mysqld_safe` command:

   ```
   shell> sudo ./mysqld_safe
   ```

   You will see output similar to the following:

   ```
   090906 02:14:37 mysqld_safe Starting mysqld daemon with databases
   from /usr/local/mysql/data
   ```

Windows:

The MySQL Windows installer enables you to install the database server as a Windows service, whereby MySQL starts and stops automatically with the operating system. If you need to start the database manually, run the `mysqld` command from the installation directory's `bin` folder.

1. Open a Windows console window (from the Start menu, choose Run and type `cmd` in the text field). A command-line window displays.
2. Enter this command (The indicated path assumes you have installed version 5.1 to the default install location):

   ```
   C:\> "C:\Program Files\MySQL\MySQL Server 5.1\bin\mysqld"
   ```

For more information, refer to the official reference manual: Starting MySQL from the Windows Command Line.

Change the Password

To set the `root` account's password to '`nbuser`', perform the following steps.

1. Open a command-line prompt and navigate to the `bin` directory within your MySQL installation.

   ```
   shell> cd <install-dir>/bin
   ```

   (Where *<install-dir>* is the path to your MySQL installation directory.)

2. Type in the following:

3. `shell> mysql -u root`

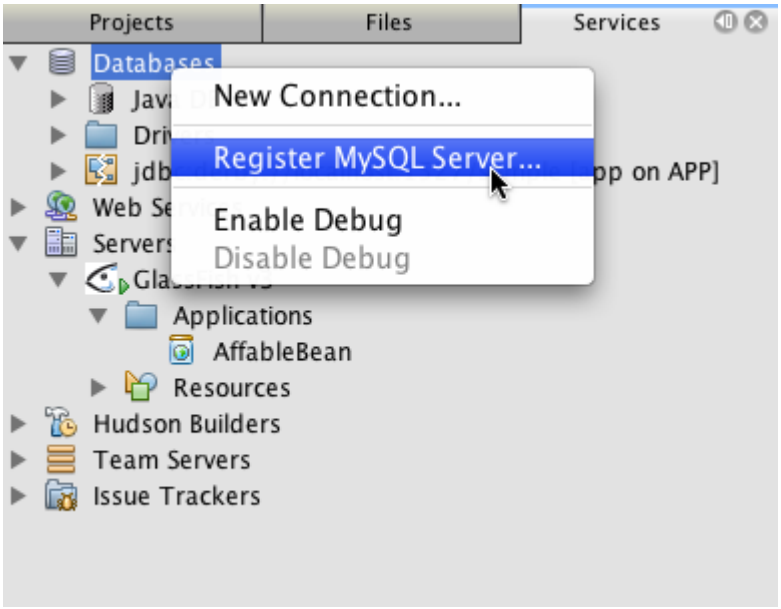
4.
```
mysql> UPDATE mysql.user SET Password = PASSWORD('nbuser') WHERE User
= 'root';
mysql> FLUSH PRIVILEGES;
```

For more information, see the official MySQL Reference Manual: Securing the Initial MySQL Accounts.

Register the Server in the IDE

The IDE's Services window enables you to connect to the server, start and stop the server, view database instances and the data they contain, as well as run an external administration tool on the server.

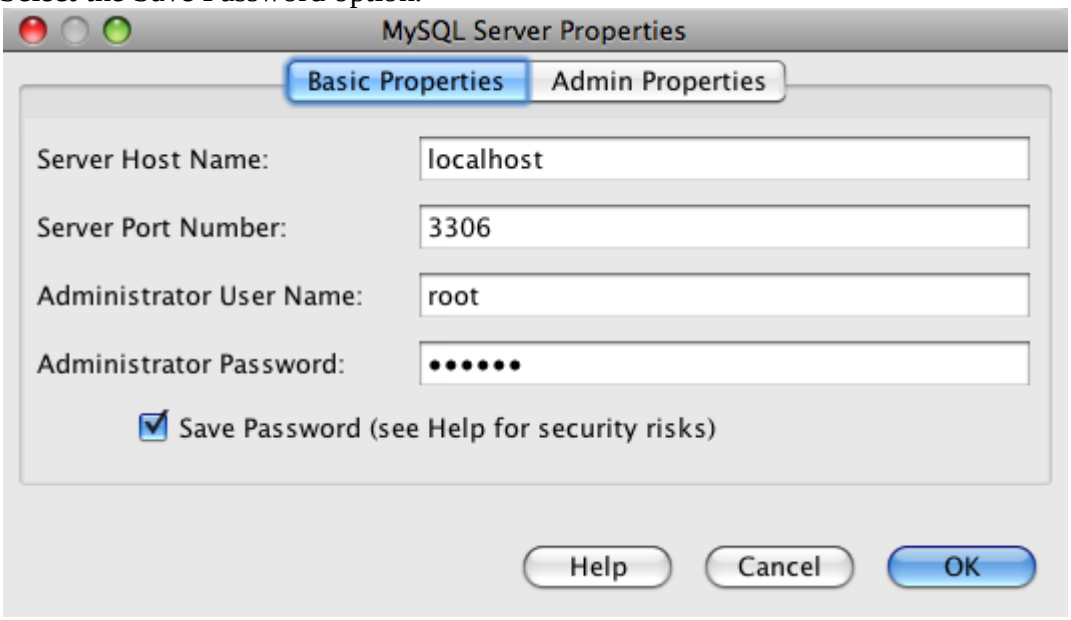
1. In the Services window, right-click the Databases node and choose Register MySQL Server.



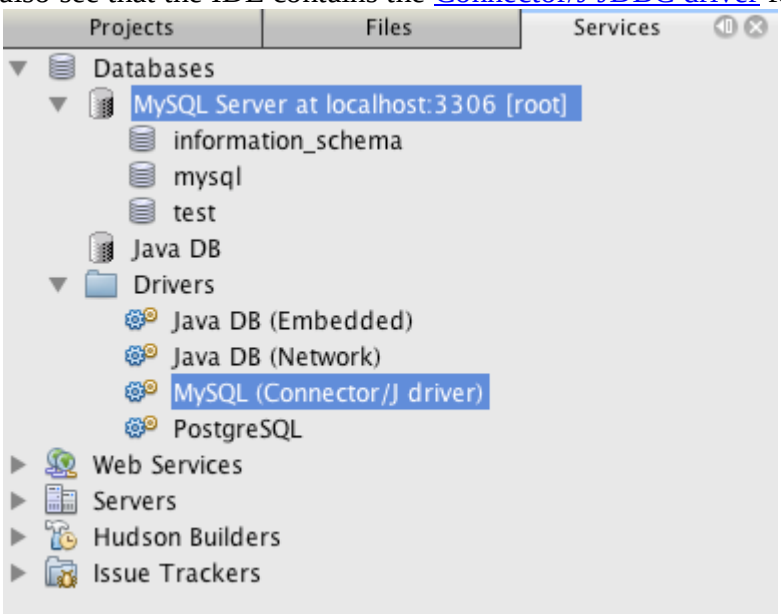
In the MySQL Server Properties dialog, under the Basic Properties tab, you can see the default settings for the MySQL server installation. These are:

- **Server Host Name:** `localhost`
- **Server Port Number:** `3306`
- **Administrator User Name:** `root`
- **Administrator Password:** `nbuser`

2. Select the Save Password option.



3. Click OK. The IDE connects to your MySQL database server, and lists database instances that are maintained by the server. If you expand the Drivers node, you can also see that the IDE contains the Connector/J JDBC driver for MySQL.



The application server (i.e., GlassFish) requires the driver to enable communication between your Java code and the the MySQL database. Because the IDE already contains the Connector/J driver, you do not need to download it. Furthermore, as will later be demonstrated, you can specify in your server settings to enable JDBC driver deployment so that the driver will be automatically deployed to GlassFish if it is missing on the server.

Steps 4-7 below are optional. You can configure the IDE to start and stop the MySQL server, as well as run an external administration tool on the server.

4. Right-click the MySQL server node and choose Properties. In the MySQL Server Properties dialog, select the Admin Properties tab.

5. In the 'Path/URL to admin tool' field, enter the path on your computer to the executable file of a database administration tool, such as MySQL Administrator. The MySQL Administrator is included in the MySQL GUI Tools bundle.
6. In the 'Path to start command' field, type in the path to the MySQL start command (i.e., `mysqld` or `mysqld_safe`, depending on your operating system. (See Start the Database Server above.)

   Note: For Unix-like systems, you may find that you can only invoke the start command with root or administrative privileges. To overcome this, you can create a script (using GKSu for Linux and Solaris, osascript for Mac) that will accomplish this task. For more information, see this blog post.

7. In the 'Path to stop command' field, enter the path to the MySQL stop command (i.e., `mysqladmin shutdown`). Because the command requires a user account with shutdown privileges, you must enter username/password credentials in the Arguments field. For example:

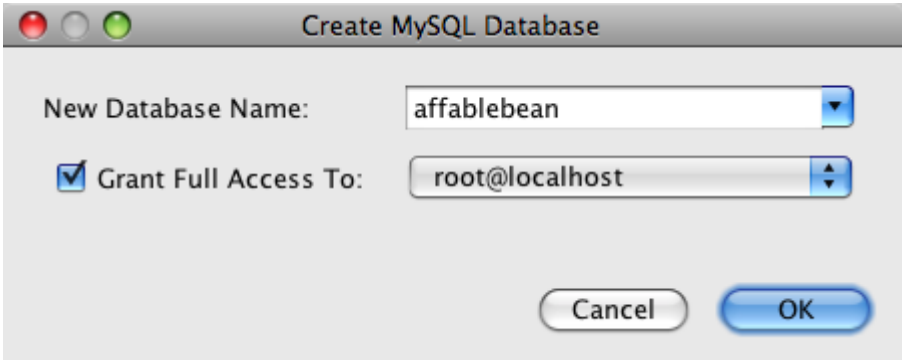
- **Arguments:** `-u root -pnbuser shutdown`

After you have set the fields listed under the Advanced Properties tab, you can:

- **Start the MySQL server:** Right-click the MySQL server node and choose Start.
- **Stop the MySQL server:** Right-click the MySQL server node and choose Stop.
- **Run the external administration tool:** Right-click the MySQL server node and choose Run Administration Tool.

Create a Database Instance

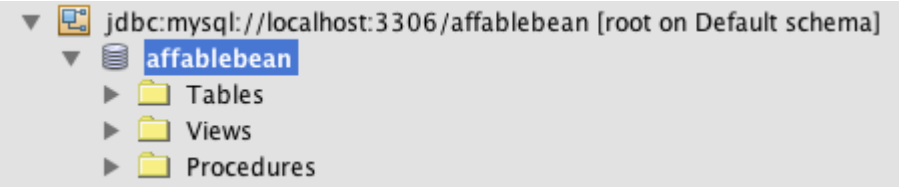
1. Create the database instance which you will use in this tutorial. To do so, right-click the MySQL Server node and choose Create Database.
2. In the dialog that displays, type in `affablebean`. Select the 'Grant Full Access to' option, then select `root@localhost root @ localhost` from the drop-down field. This enables the `root` account on the `localhost` host access to the database. Later, when you create a connection pool on the server, you'll need to provide the `root` account and `nbuser` password as username/password credentials in order to grant the server access to the database.



3. Click OK. When you do so, the database named `affablebean` is created, and a connection to the database is automatically established. Connections are displayed in the Services window using a connection node ( ).

Note: Connection nodes are persisted in the Services window. If you restart the IDE, the connection node displays with a jagged line ( ), indicating that the connection is broken. To reconnect to a database, make sure that the database server is running, then right-click the node and choose Connect.

4. Expand the connection node for the `affablebean` database. The connection contains the database's default schema (`affablebean`), and within that are nodes for tables, views, and procedures. Currently these are empty since we haven't created anything yet.



At this stage, you've connected to the MySQL server from the IDE and have created a new database named `affablebean` which you'll use throughout the tutorial. Also, you've created a Java web project in the IDE, and have confirmed that it can be successfully built and deployed to your development server, and opened in a browser when run. Now that your development environment is ready, you can begin drafting the application's data model.

The NetBeans E-commerce Tutorial - Designing the Data Model

Tutorial Contents

This tutorial unit focuses on data modeling, or the process of creating a conceptual model of your storage system by identifying and defining the entities that your system requires, and their relationships to one another. The data model should contain all the logical and physical design parameters required to generate a script using the Data Definition Language (DDL), which can then be used to create a database.[1]

In this unit, you work primarily with MySQL Workbench, a graphical tool that enables you to create data models, reverse-engineer SQL scripts into visual representations, forward-engineer data models into database schemata, and synchronize models with a running MySQL database server.

You begin by creating an entity-relationship diagram to represent the data model for the `AffableBean` application. When you have completed identifying and defining all entities and the relationships that bind them, you use Workbench to forward-engineer and run a DDL script that converts the data model into a database schema. Finally, you connect to the new schema from the NetBeans IDE.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
MySQL database server	version 5.1
MySQL Workbench	version 5.1 or 5.2

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.

Identifying Entities for the Data Model

In the real world, you may not have the luxury of designing the data model for your application. For example, your task may be to develop an application on top of an existing database system. Provided you do not have a data model to base your application on, creating one should be one of the first design steps you take before embarking on development. Creating a data model involves identifying the objects, or *entities*, required by your system and defining the relationships between them.

To begin identifying the entities we need for the data model, re-examine the use-case presented in Designing the Application. Search for commonly-occurring nouns. For example:

Use-Case

Customer visits the welcome page and selects a product **category**. **Customer** browses **products** within the selected category page, then adds a **product** to his or her **shopping cart**. **Customer** continues shopping and selects a different **category**. **Customer** adds several **products** from this **category** to **shopping cart**. **Customer** selects 'view cart' option and updates quantities for cart **products** in the cart page. **Customer** verifies shopping cart contents and proceeds to checkout. In the checkout page, **customer** views the cost of the **order** and other information, fills in personal data, then submits his or her details. The **order** is processed and **customer** is taken to a confirmation page. The confirmation page provides a unique reference number for tracking the customer **order**, as well as a summary of the **order**.

The text highlighted above in **bold** indicates the candidates that we can consider for the data model. Upon closer inspection, you may deduce that the shopping cart does not need to be included, since the data it provides (i.e., products and their quantities) is equally offered by a customer order once it is processed. In fact, as will be demonstrated in Unit 8, Managing Sessions, the shopping cart merely serves as a mechanism that retains a user session temporarily while the customer shops online. We can therefore settle on the following list:

- **customer**
- **category**
- **product**
- **order**

With these four entities, we can begin constructing an entity-relationship diagram (ERD).

Note: In this tutorial, we create a database schema from the ERD, then use the IDE's EclipseLink support to generate JPA entity classes from the existing database. (EclipseLink and the Java Persistence API (JPA) are covered in Unit 7, Adding Entity Classes and Session Beans.) This approach is described as *bottom up* development. An equally viable alternative is the *top down* approach.

- **Top down:** In *top down* development, you start with an existing Java implementation of the domain model, and have complete freedom with respect to the design of the database schema. You must create mapping metadata (i.e., annotations used in JPA entity classes), and can optionally use a persistence tool to automatically generate the schema.

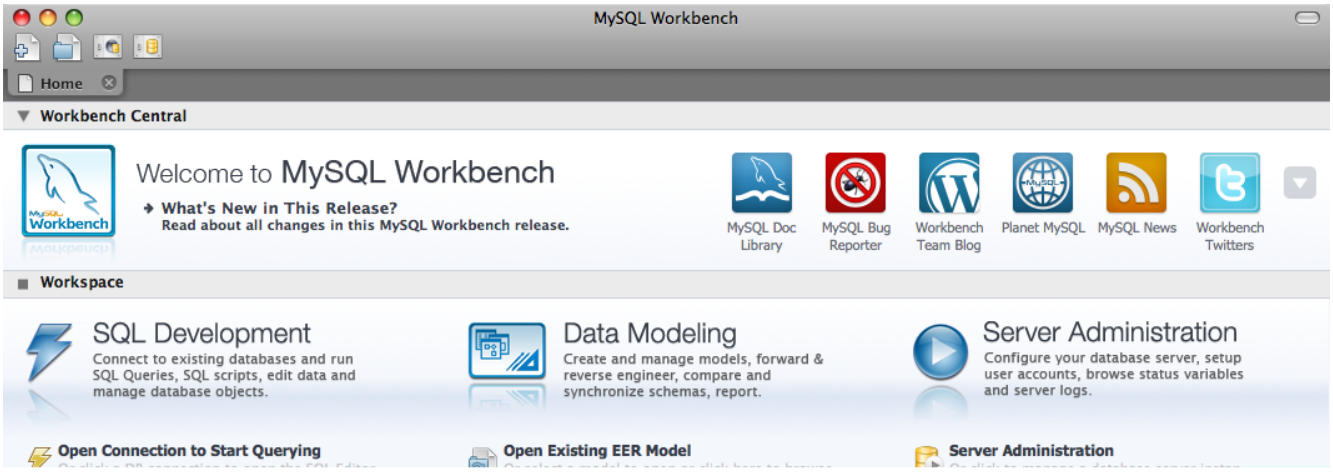
- **Bottom up:** *Bottom up* development begins with an existing database schema. In this case, the easiest way to proceed is to use forward-engineering tools to extract metadata from the schema and generate annotated Java source code (JPA entity classes).

For more information on top down and bottom up design strategies, see Data modeling: Modeling methodologies [Wikipedia].

Creating an Entity-Relationship Diagram

Start by running MySQL Workbench. In this exercise, you use Workbench to design an entity-relationship diagram for the `AffableBean` application.

Note: The following instructions work for MySQL Workbench versions 5.1 *and* 5.2. The images used in this tutorial are taken from version 5.2. There are slight differences in the graphical interface between versions, however the functionality remains consistent. Because version 5.2 incorporates a query editor (previously MySQL Query Browser), as well as a server administration interface (previously MySQL Administrator), you are presented with the Home screen when opening the application (shown below).

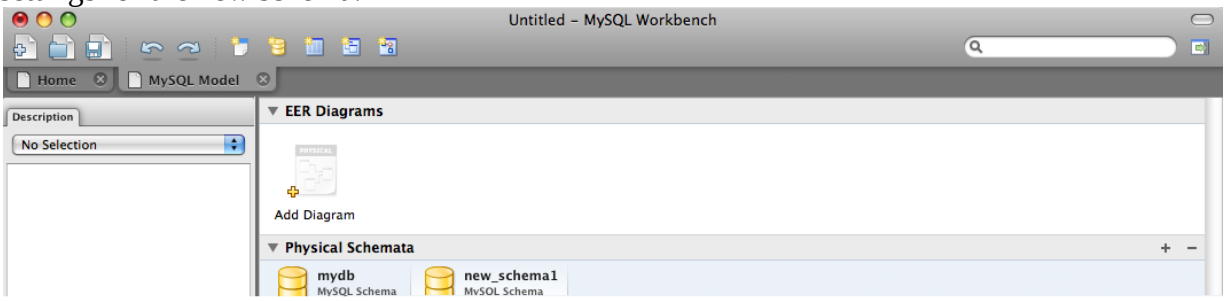


If you are working in Workbench 5.2, click **Create New EER Model** beneath the Data Modeling heading in the Home screen.

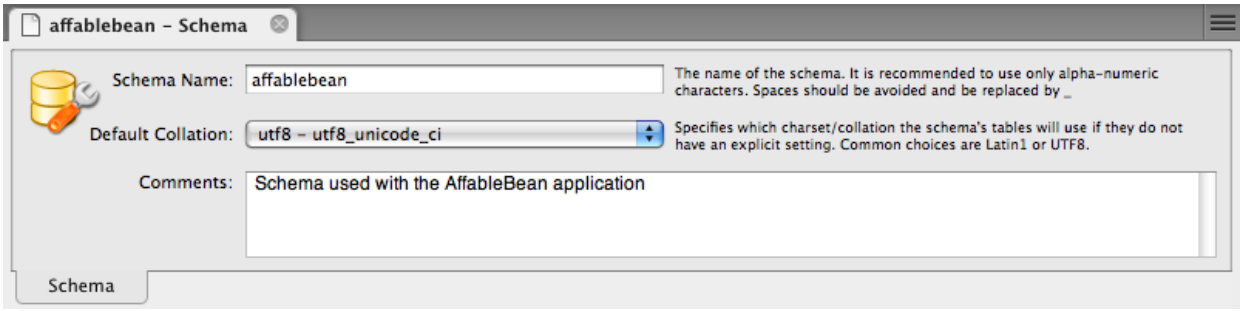
Creating the `affablebean` Schema

1. In the default interface, begin by creating a new schema which will be used with the `AffableBean` application. Click the plus (+) icon located to the right of the **Physical Schemata** heading.

   A new panel opens in the bottom region of the interface, enabling you to specify settings for the new schema.



2. Enter the following settings for the new schema:
   - **Schema Name:** `affablebean`
   - **Default Collation:** `utf8 - utf8_unicode_ci`
   - **Comments:** `Schema used with the AffableBean application`



The new schema is created, and becomes listed under the Catalog tab in the right region of the Workbench interface.

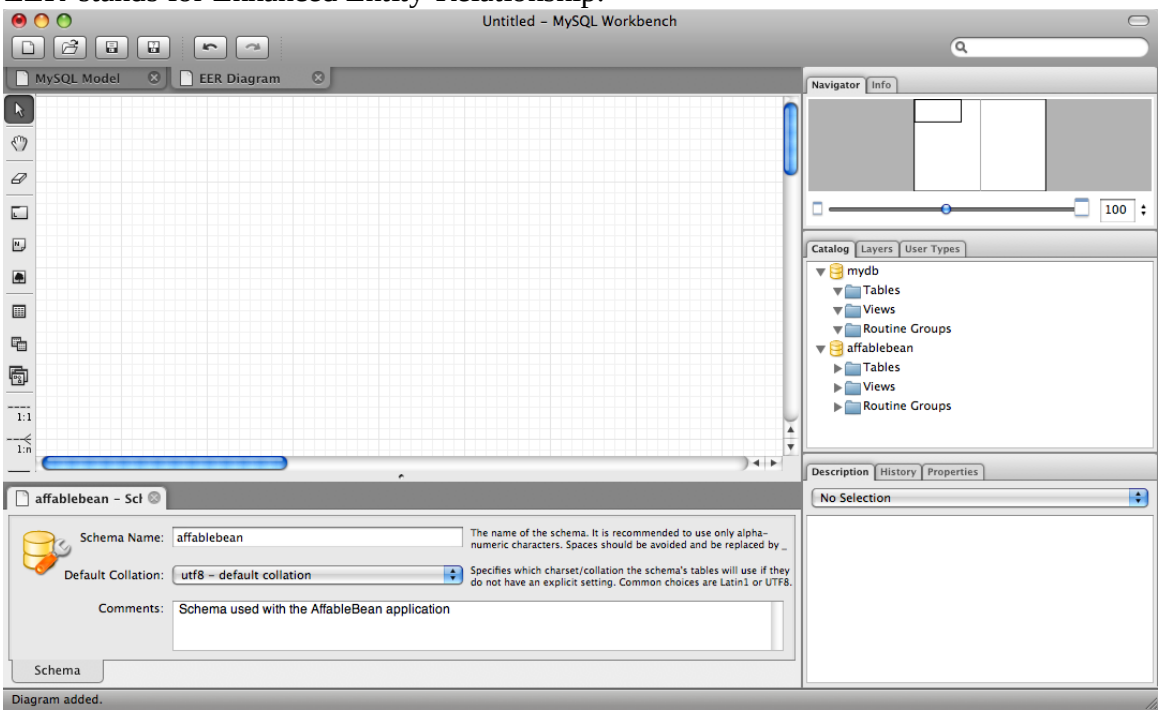
For an explanation of character sets and collations, see the MySQL Server Manual: 9.1.1. Character Sets and Collations in General.

Creating Entities

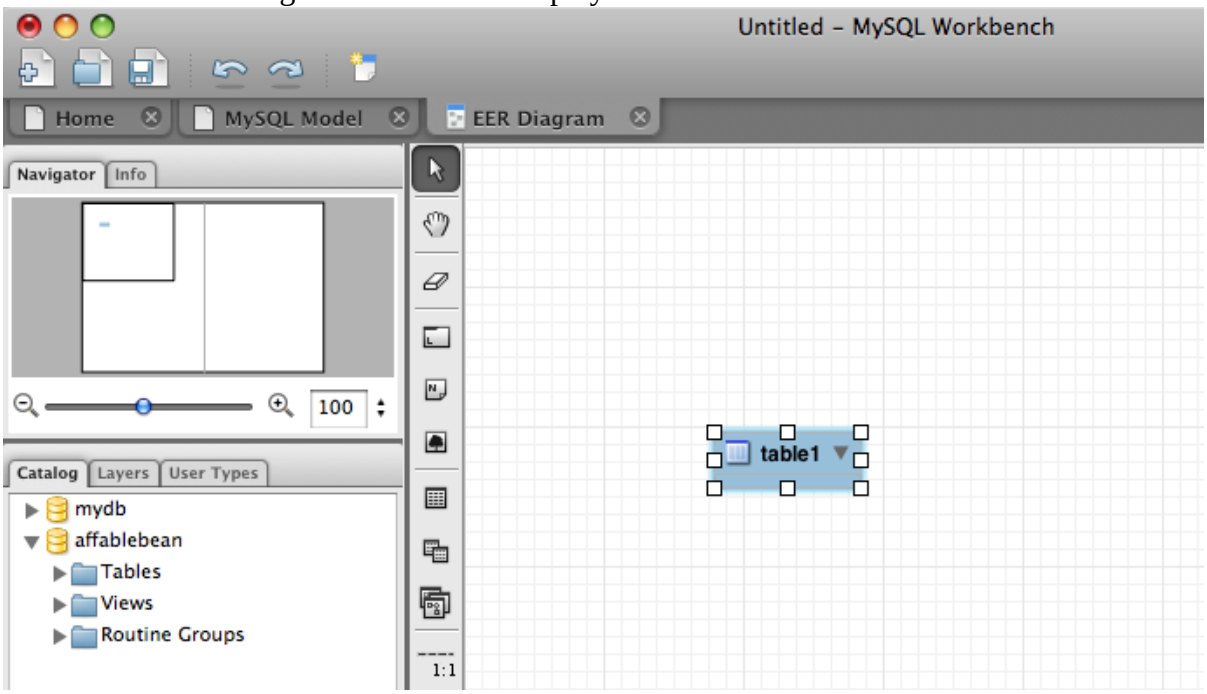
Start by creating a new entity-relationship diagram in MySQL Workbench. You can drag-and-drop entity tables onto the canvas.

1. Under the EER Diagrams heading in WorkBench, double-click the Add Diagram ( ) icon. A new EER Diagram opens displaying an empty canvas.

   'EER' stands for Enhanced Entity-Relationship.



2. Click the New Table ( ) icon located in the left margin, then hover your mouse onto the canvas and click again. A new table displays on the canvas.

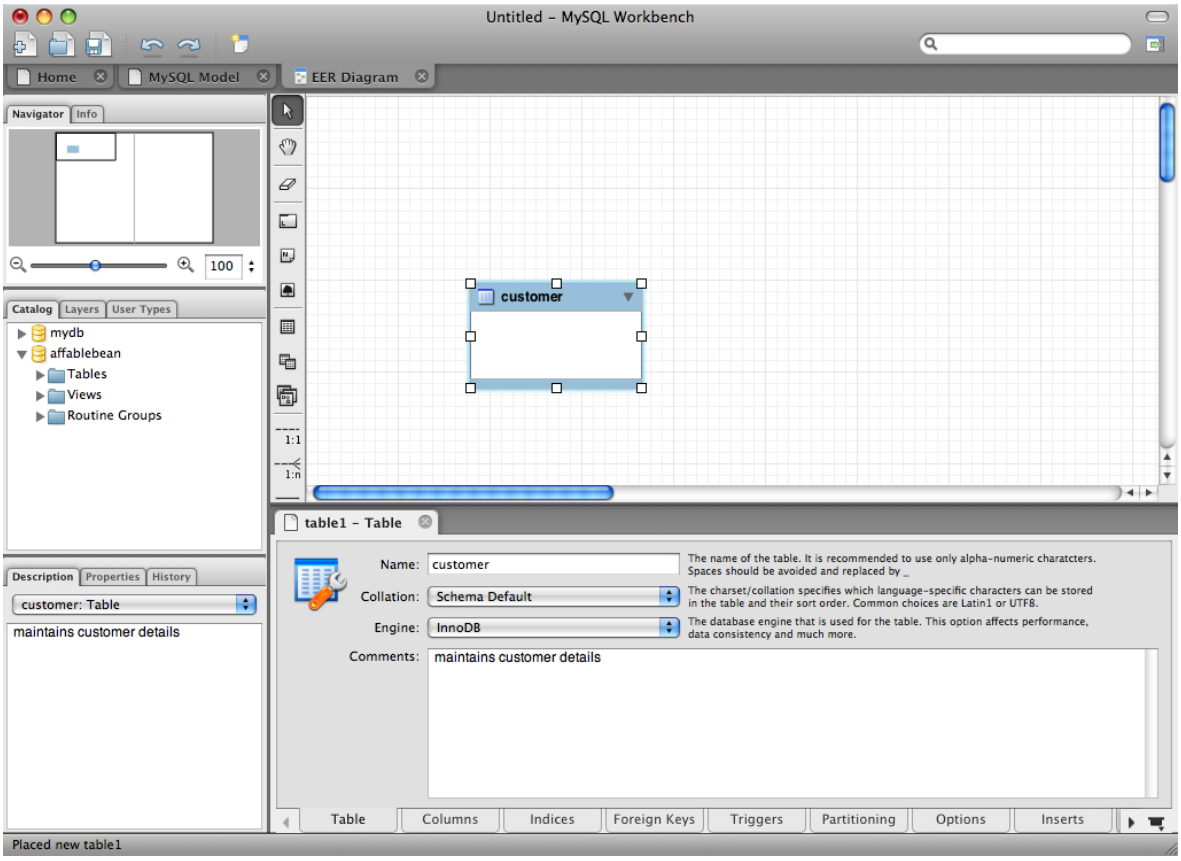


3. Double-click the table. The Table editor opens in the bottom region of the interface, allowing you to configure settings for the table.

   Note: The terms 'table' and 'entity' are nearly synonymous in this tutorial unit. From the point of view of a database schema, you are creating tables. From a data modeling perspective, you are creating entities. Likewise, the columns that you later create for each table correspond to entity *properties*.

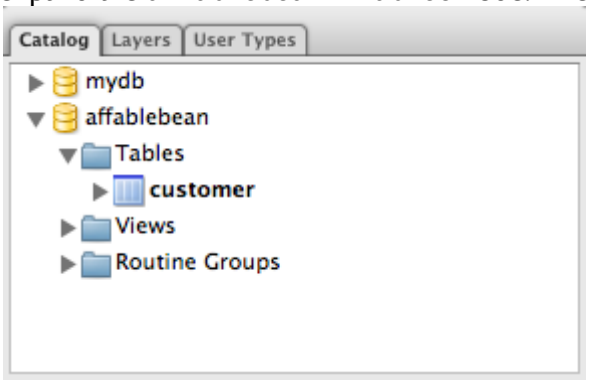
4. In the Table editor, rename the table to one of the nouns you identified from the use-case above. Optionally add a comment describing the purpose of the table. For example:

- **Name:** `customer`
- **Engine:** `InnoDB`
- **Comments:** `maintains customer details`



The InnoDB engine provides foreign key support, which is utilized in this tutorial. Later, under Forward-Engineering to the Database, you set the default storage engine used in Workbench to InnoDB.

5. Under the **Catalog** tab in the left region of WorkBench (right region for version 5.1), expand the `affablebean` > `Tables` node. The **customer** table now displays.

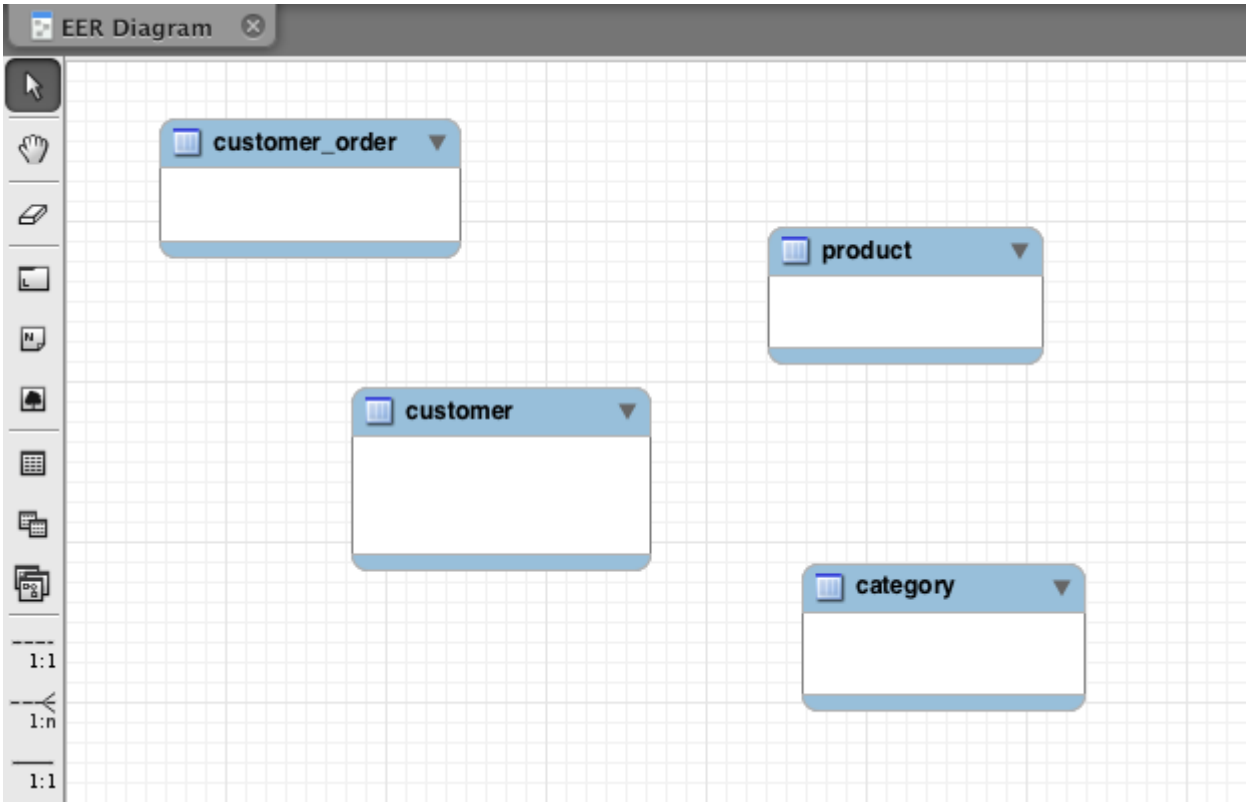


More importantly, note that the new `customer` table is now included in the `affablebean` schema. Because the `affablebean` schema was selected when you created the new EER diagram, any changes you make to the diagram are automatically bound to the schema.

6. Repeat steps 2 - 4 above to add tables to the canvas for the remaining nouns you identified in the use-case above. Before naming your tables however, there is one important consideration which you should take into account. Certain keywords hold special meaning for the SQL dialect used by the MySQL server. Unfortunately,

'`order`' is one of them. (For example, '`order`' can be used in an `ORDER BY` statement.) Therefore, instead of naming your table '`order`', name it '`customer_order`' instead. At this stage, don't worry about arranging the tables on the canvas in any special order.

For a list of reserved words used by the MySQL server, refer to the official manual: 2.2. Reserved Words in MySQL 5.1.



Adding Entity Properties

Now that you've added entities to the canvas, you need to specify their properties. Entity properties correspond to the columns defined in a database table. For example, consider the `customer` entity. In regard to the `AffableBean` application, what aspects of a customer would need to be persisted to the database? These would likely be all of the information gathered in the checkout page's customer details form, as well as some association to the processed order.

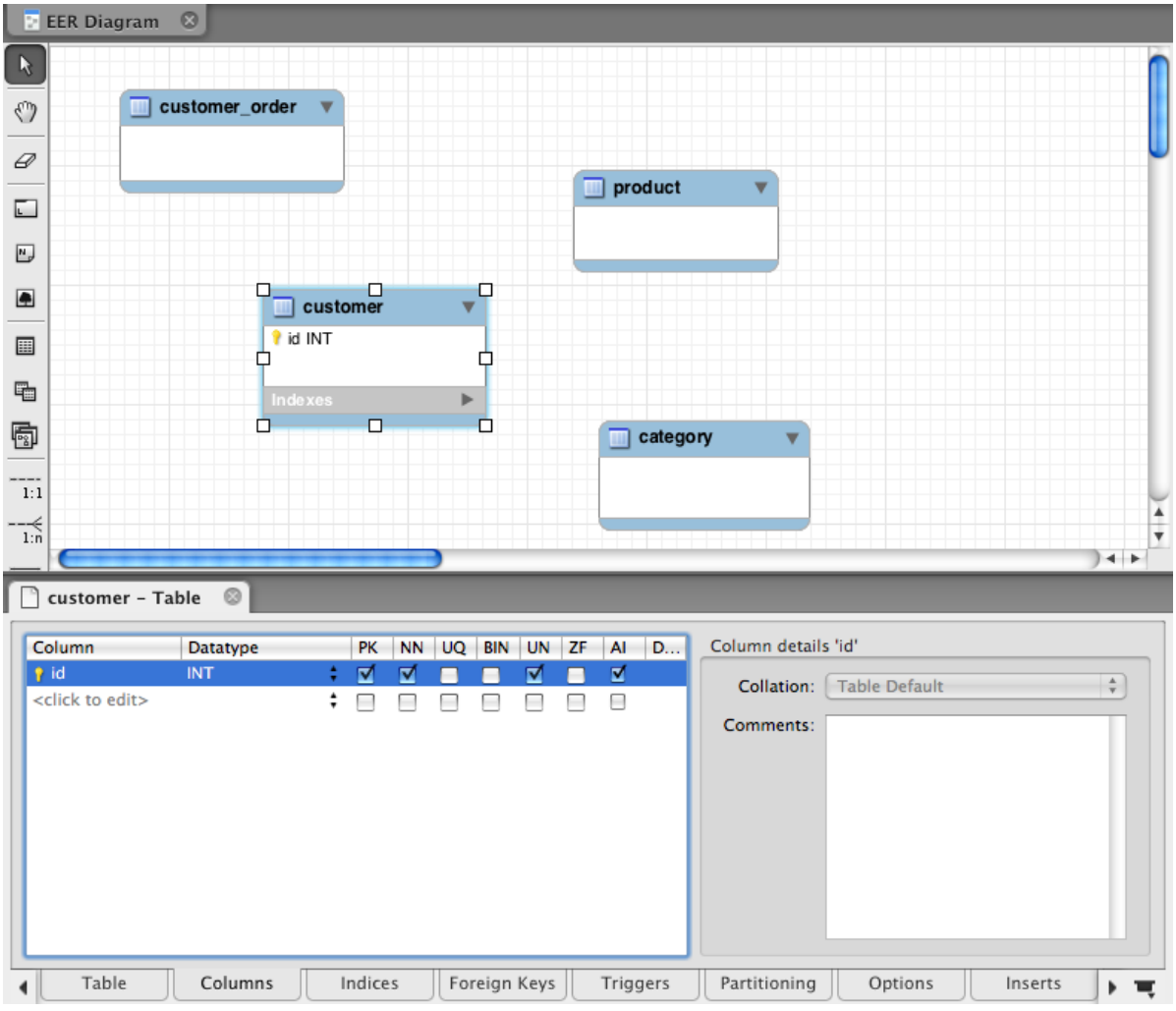
When adding properties, you need to determine the most appropriate data type for each property. MySQL supports a number of data types in several categories: numeric types, date and time types, and string (character) types. Refer to the official manual for a summary of data types within each category: 10.1. Data Type Overview. In this tutorial, the data types have been chosen for you. Choosing the appropriate data type plays a significant role in optimizing storage on your database server. For more information see:

- 10.5. Data Type Storage Requirements
- 10.6. Choosing the Right Type for a Column

The following steps describe how you can use MySQL Workbench to add properties to an existing entity in your ERD. As with most of the initial design steps, determining the entity properties would call for careful consideration of the business problem that needs to be solved, and could require hours of analysis as well as numerous consultations with the client.

1. Double-click the `customer` table heading to bring up the Table editor in WorkBench.
2. In the Table editor click the Columns tab, then click inside the displayed table to edit the first column. Enter the following details:

Column	Datatype	PK (Primary Key)	NN (Not Null)	UN (Unsigned)	AI (Autoincrement)
`id`	`INT`	✓	✓	✓	✓

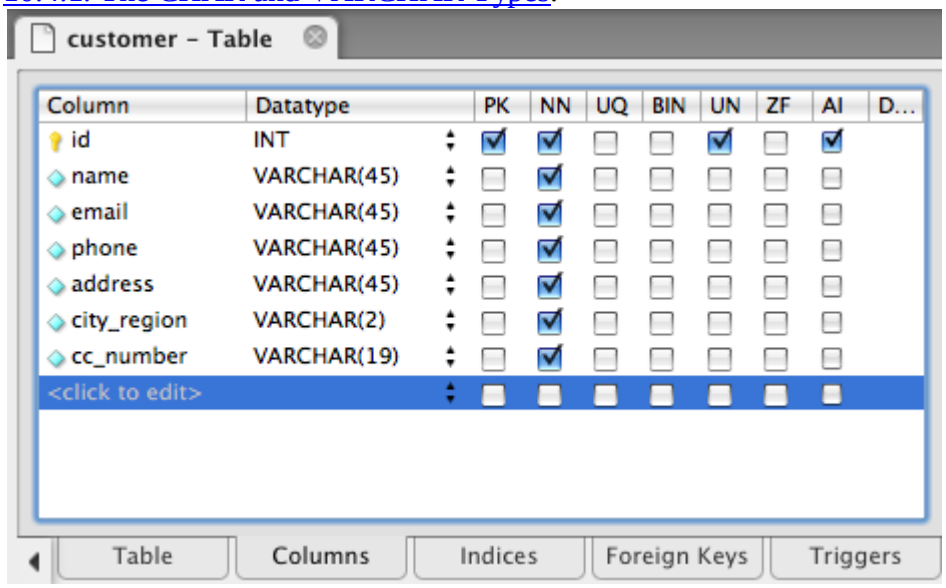
3. 

4. Continue working in the `customer` table by adding the following `VARCHAR` columns. These columns should be self-explanatory, and represent data that would need to be captured for the Affable Bean business to process a customer order and send a shipment of groceries to the customer address.

Column	Datatype	NN (Not Null)
`name`	`VARCHAR(45)`	✓
`email`	`VARCHAR(45)`	✓

phone	VARCHAR(45)	✓
address	VARCHAR(45)	✓
city_region	VARCHAR(2)	✓
cc_number	VARCHAR(19)	✓

5. For an explanation of the `VARCHAR` data type, see the MySQL Reference Manual: 10.4.1. The CHAR and VARCHAR Types.



6. With the `customer` table selected on the canvas, choose Arrange > Reset Object Size to resize the table so that all columns are visible on the canvas. Also click the Indexes row so that any table indexes are also visible. (This includes primary and foreign keys, which becomes useful when you begin creating relationships between tables later in the exercise.)

   When you finish, the `customer` entity looks as follows.

7. Follow the steps outlined above to create columns for the remaining tables.

   category

Column	Datatype	PK	NN	UN	AI
id	TINYINT	✓	✓	✓	✓

name	VARCHAR(45)		✓			

customer_order

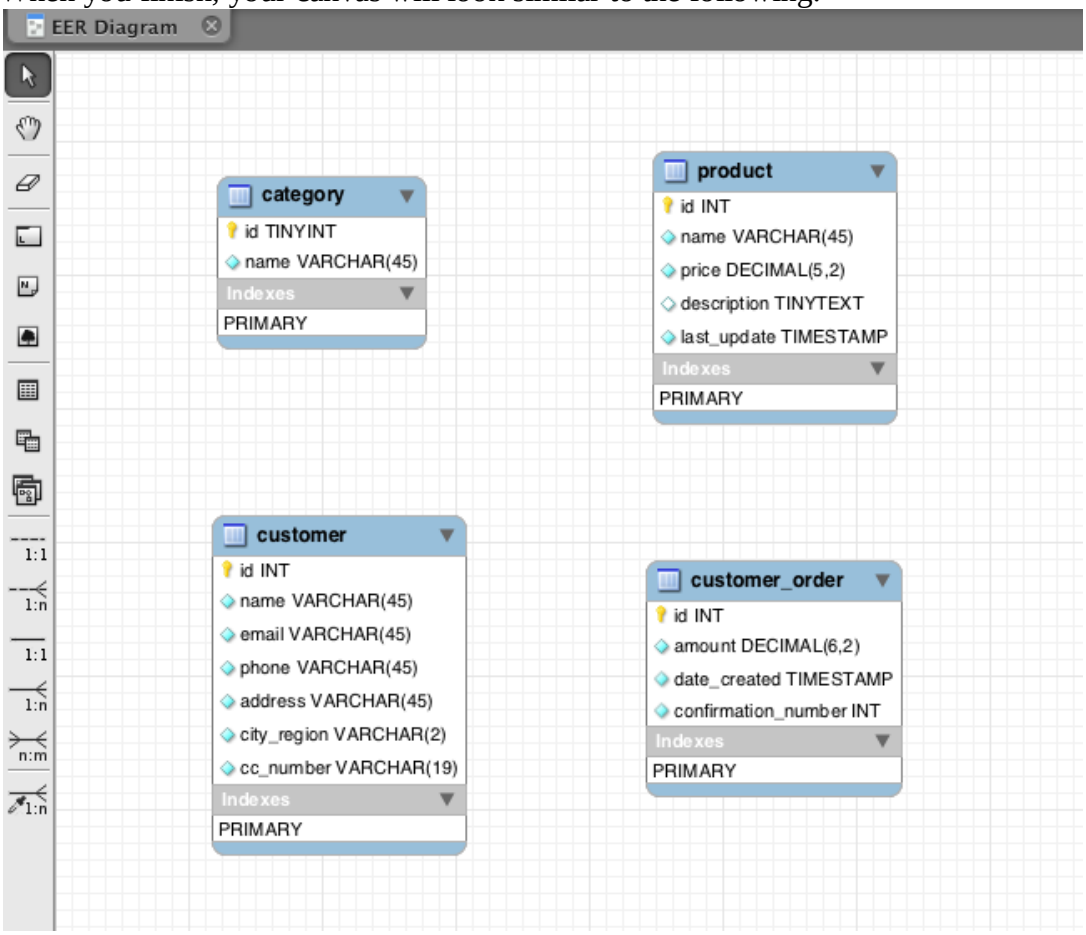
Column	Datatype	PK	NN	UN	AI	Default
id	INT	✓	✓	✓	✓	
amount	DECIMAL(6,2)		✓			
date_created	TIMESTAMP		✓			CURRENT_TIMESTAMP
confirmation_number	INT		✓	✓		

product

Column	Datatype	PK	NN	UN	AI	Default
id	INT	✓	✓	✓	✓	
name	VARCHAR(45)		✓			
price	DECIMAL(5,2)		✓			
description	TINYTEXT					
last_update	TIMESTAMP		✓			CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP

For details on the `TIMESTAMP` data type, see the MySQL Reference Manual: 10.3.1.1. TIMESTAMP Properties.

When you finish, your canvas will look similar to the following.



Identifying Relationships

So far, the entity-relationship diagram contains several entities, but lacks any relationships between them. The data model that we are creating must also indicate whether objects are aware of (i.e., contain references to) one another. If one object contains a reference to another object, this is known as a *unidirectional* relationship. Likewise, if both objects refer to each other, this is called a *bidirectional* relationship.

References correlate to foreign keys in the database schema. You will note that, as you begin linking tables together, foreign keys are added as new columns in the tables being linked.

Two other pieces of information are also commonly relayed in an ERD: *cardinality* (i.e., multiplicity) and *ordinality* (i.e., optionality). These are discussed below, as you begin adding relationships to entities on the canvas. In order to complete the ERD, you essentially need to create two *one-to-many* relationships, and one *many-to-many* relationship. Details follow.

- Creating One-To-Many Relationships
- Creating Many-To-Many Relationships

Creating One-To-Many Relationships

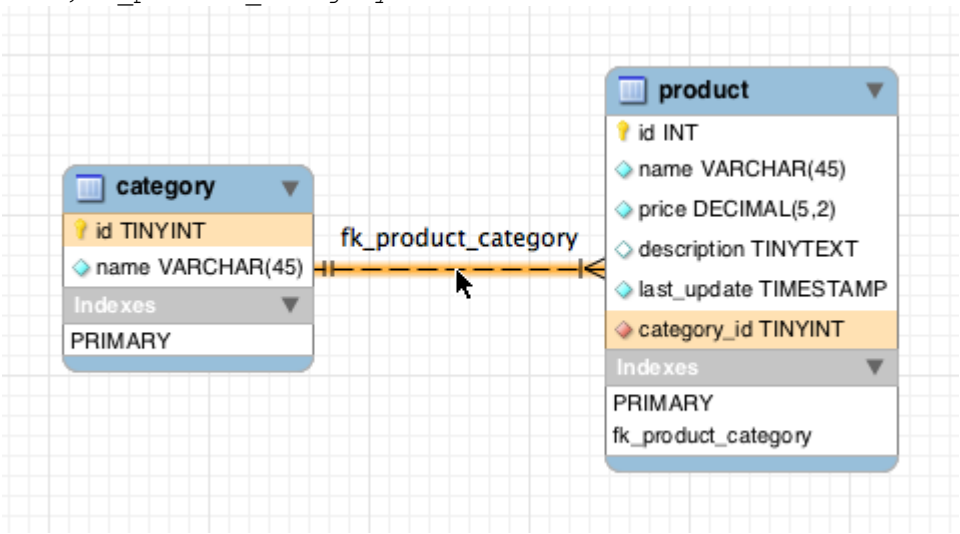
Examine the four objects currently on the canvas while considering the business problem. You can deduce the following two *one-to-many* relationships:

- A category must contain one or more products
- A customer must have placed one or more orders

Incorporate these two relationships into the ERD. You can download a copy of the MySQL Workbench project that contains the four entities required for the following steps: affablebean.mwb.

1. In the left margin, click the 1:n Non-Identifying Relationship (

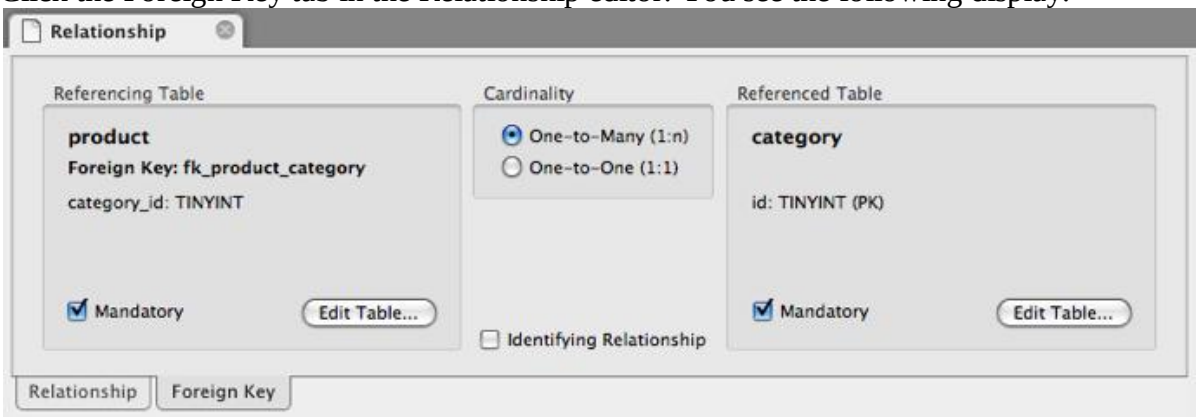
) button. This enables you to create a *one-to-many* relationship.
2. Click the `product` table, then click the `category` table. The first table you click will contain the foreign key reference to the second table. Here, we want the `product` table to contain a reference to `category`. In the image below, you see that a new column, `category_id`, has been added to the `product` table, and that a foreign key index, `fk_product_category` has been added to the table's indexes.



   Since foreign keys must be of the same data type as the columns they reference, notice that `category_id` is of type `TINYINT`, similar to the `category` table's primary key.

   The entity-relationship diagram in this tutorial uses Crow's Foot notation. You can alter the relationship notation in WorkBench by choosing Model > Relationship Notation.
3. Double-click the relationship (i.e., click the dashed line between the two entities). The Relationship editor opens in the bottom region of the interface.
4. Change the default caption to '`belongs to`'. In other words, "product x belongs to category y." Note that this is a *unidirectional* relationship: A `product` object contains a reference to the category it belongs to, but the related `category` object does not contain any references to the products it contains.

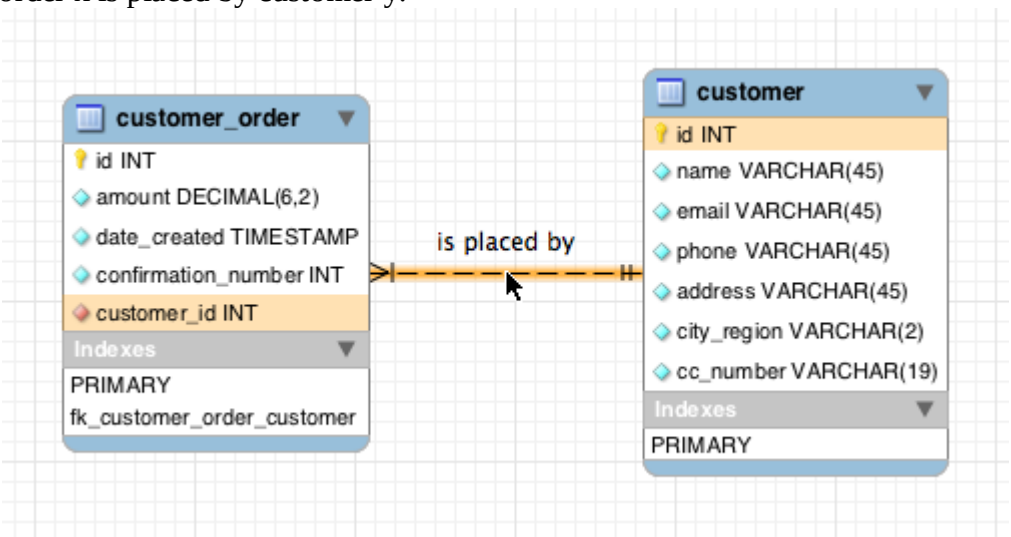
5. Click the Foreign Key tab in the Relationship editor. You see the following display.



Under the Foreign key tab, you can modify a relationship's:

- **cardinality:** whether the relationship between two objects is *one-to-one* or *one-to-many*.
- **ordinality:** whether a reference between entities must exist in order to maintain the integrity of the model. (Toggle the Mandatory checkbox for either side.)
- **type:** (i.e., *identifying* or *non-identifying*). A non-identifying relationship, such as this one, refers to the fact that the child object (`product`) can be identified independently of the parent (`category`). An identifying relationship means that the child cannot be uniquely identified without the parent. An example of this is demonstrated later, when you create a many-to-many relationship between the `product` and `order` tables.

6. Click the 1:n Non-Identifying Relationship () button. In the following steps, you create a *one-to-many* relationship between the `customer` and `customer_order` objects.
7. Click the `order` table first (this table will contain the foreign key), then click the `customer` table. A relationship is formed between the two tables.
8. Click the link between the two tables, and in the Relationship editor that displays, change the default caption to '`is placed by`'. The relationship now reads, "customer order x is placed by customer y."

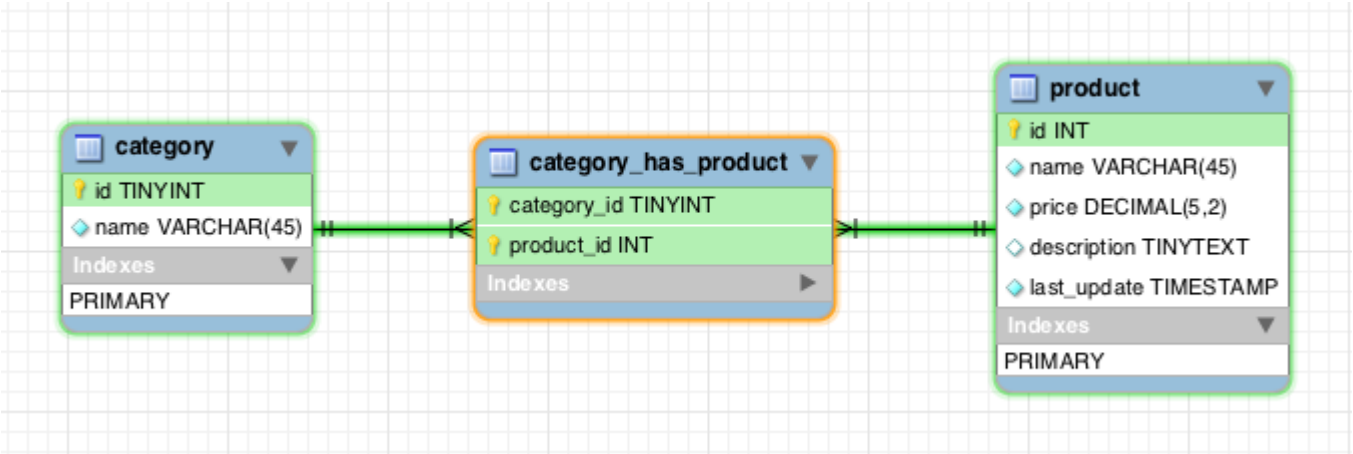


You can click and drag tables on the canvas into whatever position makes the most sense for your model. In the image above, the `order` table has been moved to the left of `customer`.

Creating Many-To-Many Relationships

Many-to-many relationships occur when both sides of a relationship can have numerous references to related objects. For example, imagine the Affable Bean business offered products that could be listed under multiple categories, such as cherry ice cream, sausage rolls, or avocado soufflé. The data model would have to account for this by including a *many-to-many* relationship between `product` and `category`, since a category contains multiple products, and a product can belong to multiple categories.

In order to implement a *many-to-many* relationship in a database, it is necessary to break the relationship down into two *one-to-many* relationships. In doing so, a third table is created containing the primary keys of the two original tables. The `product` - `category` relationship described above might look as follows in the data model.

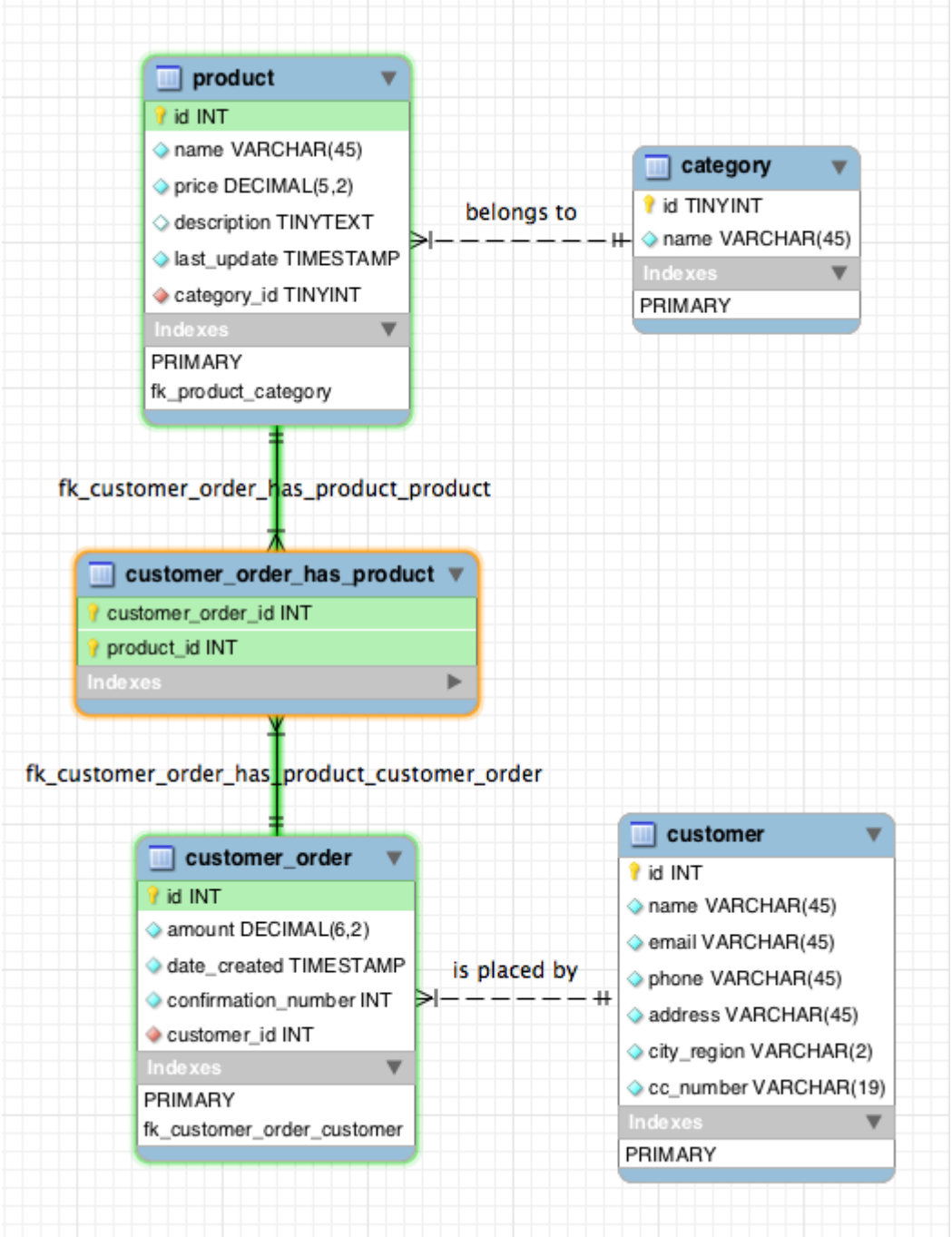


Now, consider how the application will persist customer orders. The `customer_order` entity already contains necessary properties, such as the date it is created, its confirmation number, amount, and a reference to the customer who placed it. However, there currently is no indication of the products contained in the order, nor their quantities. You can resolve this by creating a *many-to-many* relationship between `customer_order` and `product`. This way, to determine which products are contained in a given order, the application's business logic can query the new table that arises from the many-to-many relationship, and search for all records that match an `order_id`. Because customers can specify quantities for products in their shopping carts, we can also add a `quantity` column to the table.

1. In the left margin, click the n:m Identifying Relationship () button. This enables you to create a *many-to-many* relationship.
2. Click the `customer_order` table, then click the `product` table. A new table appears, named `customer_order_has_product`.

   Recall that an *identifying relationship* means that the child cannot be uniquely identified without the parent. Identifying relationships are indicated on the Workbench canvas by a solid line linking two tables. Here, the `customer_order_has_product` table forms an identifying relationship with its two parent tables, `customer_order` and `product`. A record contained in the `customer_order_has_product` table requires references from both tables in order to exist.

3. Arrange the tables according to the following image. The *many-to-many* relationship is highlighted below.

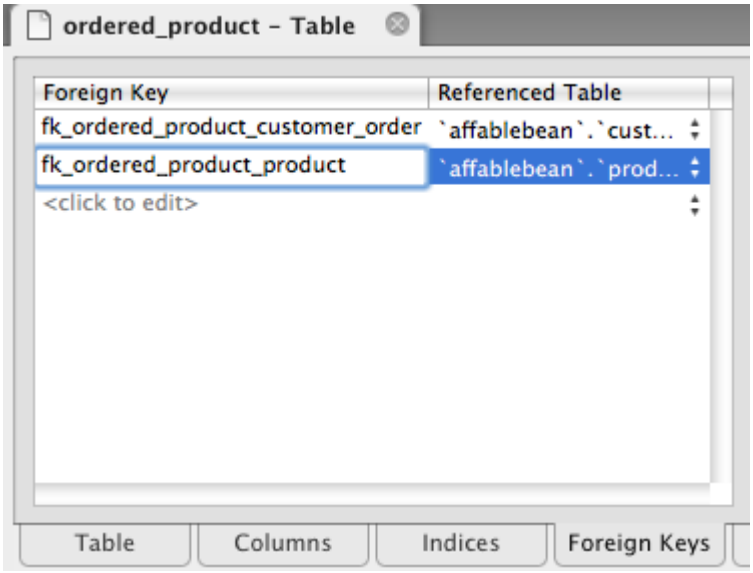


The new `customer_order_has_product` table contains two foreign keys, `fk_customer_order_has_product_customer_order` and `fk_customer_order_has_product_product`, which reference the primary keys of the `customer_order` and `product` tables, respectively. These two foreign keys form a composite primary key for the `customer_order_has_product` table.

4. Change the name of the new `customer_order_has_product` table to '`ordered_product`'. Double-click the `customer_order_has_product` table to open the Table editor. Enter `ordered_product` into the Name field.
5. Rename the foreign key indexes to correspond to the new table name. In the `ordered_product`'s Table editor, click the Foreign Keys tab. Then, click into both

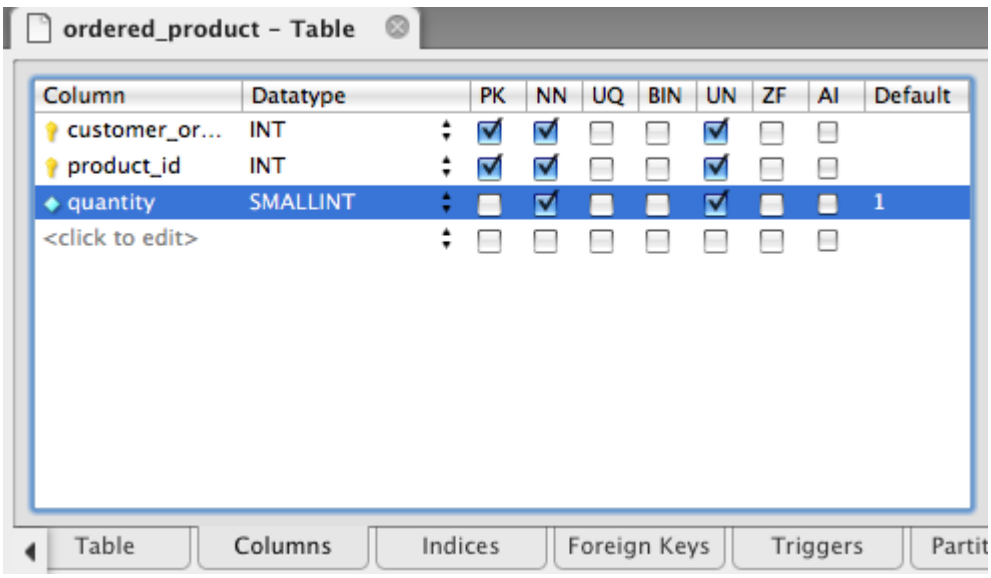
foreign key entries and replace '`customer_order_has_product`' with '`ordered_product`'. When you finish, the two entries should read:

- `fk_`**`ordered_product`**`_customer_order`
- `fk_`**`ordered_product`**`_product`

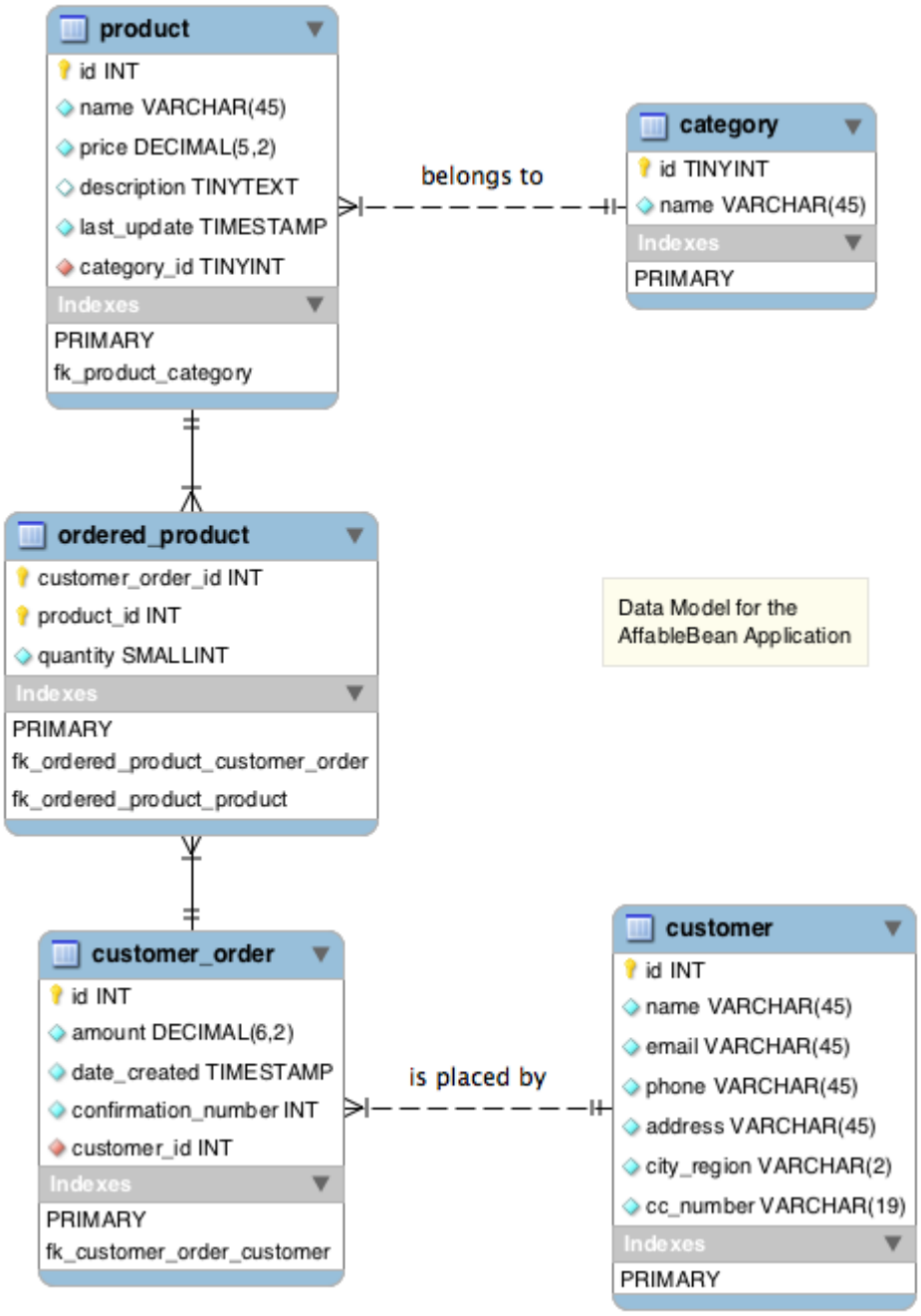


6. Double-click the lines between the two objects and delete the default captions in the Relationship editor.
7. Create a `quantity` column in the `ordered_product` table. To do so, click the Columns tab in the `ordered_product`'s Table editor. Enter the following details.

Column	Datatype	NN (Not Null)	UN (Unsigned)	Default
`quantity`	`SMALLINT`	✓	✓	`1`

8. 

You have now completed the ERD (entity-relationship diagram). This diagram represents the data model for the `AffableBean` application. As will later be demonstrated, the JPA entity classes that you create will be derived from the entities existing in the data model.



Choose View > Toggle Grid to disable the canvas grid. You can also create notes for your diagram using the New Text Object ( ) button in the left margin.

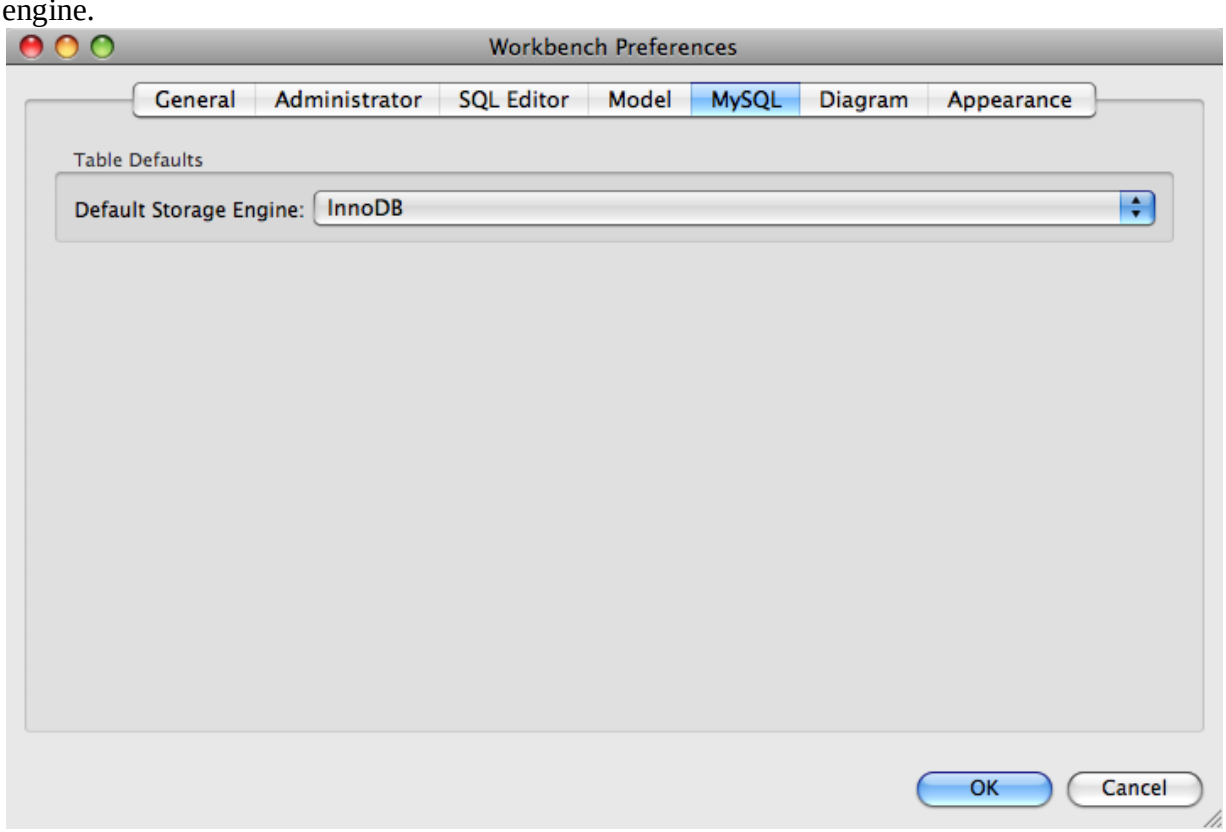
Forward-Engineering to the Database

To incorporate the data model you created into the MySQL database, you can employ WorkBench to forward-engineer the diagram into an SQL script (more precisely, a DDL

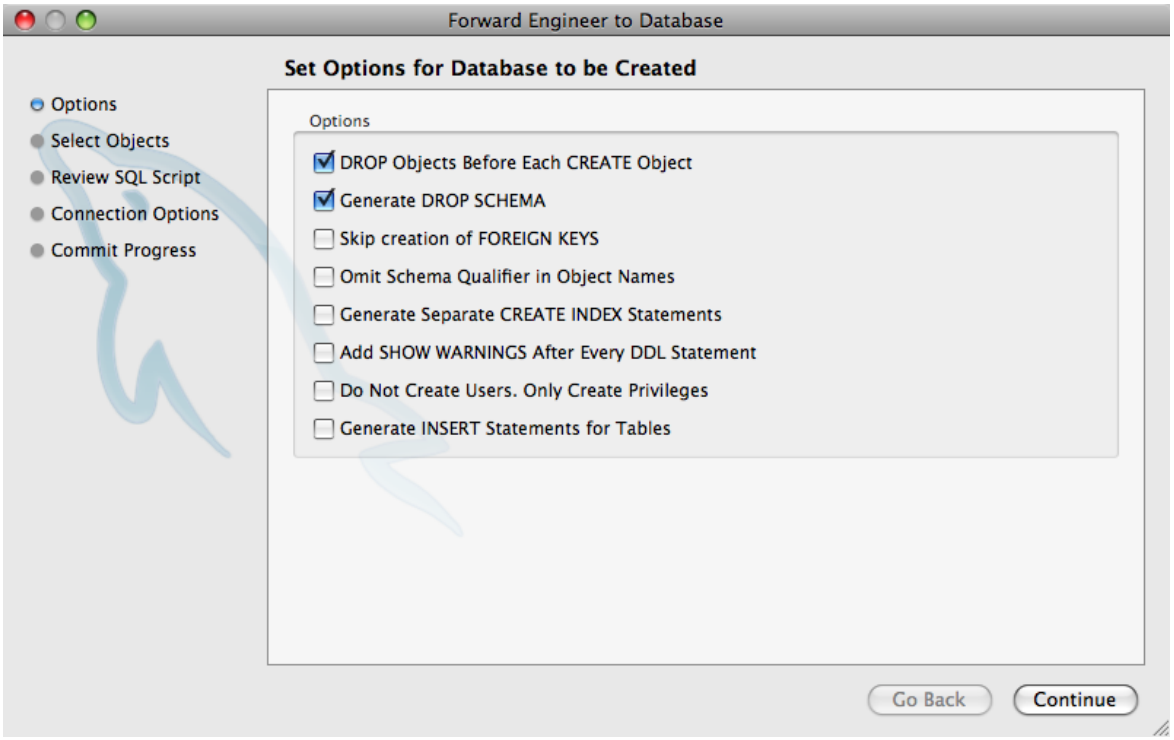
script) to generate the schema. The wizard that you use also enables you to immediately run the script on your database server.

Important: Make sure your MySQL database server is running. Steps describing how to setup and run the database are provided in Setting up the Development Environment: Communicating with the Database Server.

1. Set the default storage engine used in Workbench to InnoDB. Choose Tools > Options (MySQLWorkbench > Preferences on Mac) to open the Workbench Preferences window. Click the MySQL tab, then select InnoDB as the default storage engine.



   The InnoDB engine provides foreign key support, which is utilized in this tutorial.
2. Click OK to exit the Preferences window.
3. Choose Database > Forward Engineer from the main menu.
4. In the first panel of the Forward Engineer to Database wizard, select '`DROP Objects Before Each CREATE Object`', and '`Generate DROP SCHEMA`'.



These `DROP` options are convenient for prototyping - if you need to make changes to the schema or schema tables, the script will first delete (i.e., *drop*) these items before recreating them. (If you attempt to create items on the MySQL server that already exist, the server will flag an error.)

5. Click Continue. In Select Objects to Forward Engineer panel, note that the Export MySQL Table Objects option is selected by default. Click the Show Filter button and note that all five tables in the `affablebean` schema are included.
6. Click Continue. In the Review SQL Script panel, you can examine the SQL script that has been generated based on the data model. Optionally, click Save to File to save the script to a location on your computer.

   Note: In examining the script, you may notice that the following variables are set at the top of the file:

   ```
   SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
   SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
   FOREIGN_KEY_CHECKS=0;
   SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';
   ```

   For an explanation of what these variables are, and their purpose in the script, see the official Workbench manual: Chapter 11. MySQL Workbench FAQ.

7. Click Continue. In the Connection Options panel, set the parameters for connecting to the running MySQL server.

- **Hostname:** `127.0.0.1` (*or* *`localhost`*)
- **Port:** `3306`
- **Username:** `root`
- **Password:** `nbuser`

(The parameters you set should correspond to those from Setting up the Development Environment: Communicating with the Database Server.)

8. Click Execute. In the final panel of the wizard, you receive confirmation that the wizard was able to connect to and execute the script successfully.
9. Click Close to exit the wizard.

The `affablebean` schema is now created and exists on your MySQL server. In the next step, you connect to the schema, or *database*, from the IDE. At this stage you may ask, "What's the difference between a schema and a database?" In fact, the MySQL command `CREATE SCHEMA` is a synonym for `CREATE DATABASE`. (See 12.1.10. CREATE DATABASE Syntax.) Think of a schema as a blueprint that defines the contents of the database, including tables, relationships, views, etc. A database implements the schema by containing data in a way that adheres to the structure of the schema. This is similar to the object-oriented world of Java classes and objects. A class defines an object. When a program runs however, objects (i.e., class instances) are created, managed, and eventually destroyed as the program runs its course.

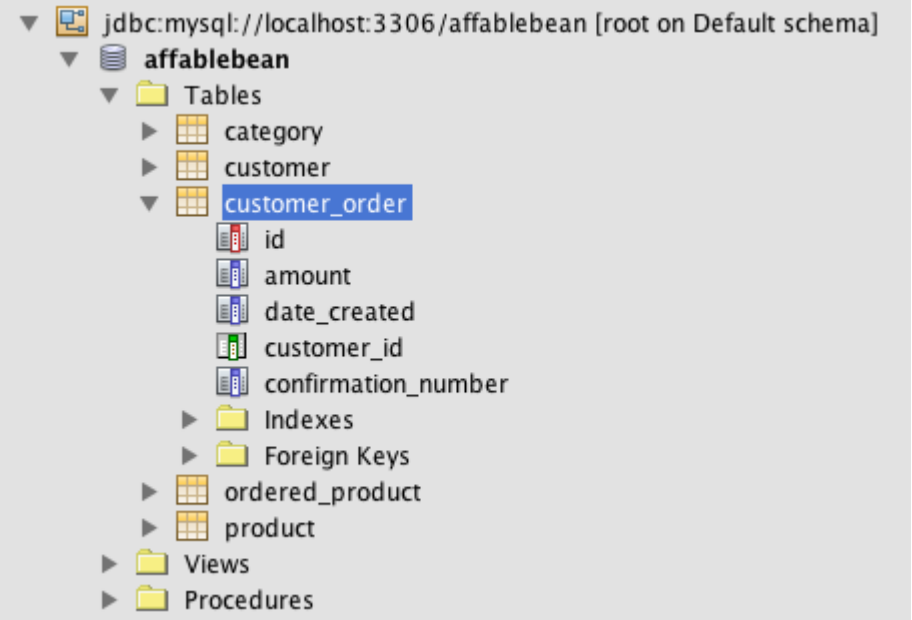
Connecting to the Database from the IDE

Now that the `affablebean` schema exists on your MySQL server, ensure that you can view the tables you created in the ERD from the IDE's Services window.

Important: Make sure that you have followed the steps outlined in Setting up the Development Environment: Communicating with the Database Server. This heading describes how to run the MySQL database server, register it with the IDE, create a database instance, and form a connection to the instance from the IDE.

1. In the IDE, open the Services window (Ctrl-5; ⌘-5 on Mac) and locate the database connection node () for the `affablebean` database instance you created in the previous tutorial unit.
2. Refresh the connection to the `affablebean` database. To do so, right-click the connection node and choose Refresh.
3. Expand the Tables node. You can now see the five tables defined by the schema.

4. Expand any of the table nodes. Each table contains the columns and indexes that you created when working in MySQL Workbench.



The IDE is now connected to a database that uses the schema you created for the `AffableBean` application. From the IDE, you can now view any table data you create in the database, as well as directly modify, add and delete data. You will explore some of these options later, in Connecting the Application to the Database, after you've added sample data to the database.

The NetBeans E-commerce Tutorial - Preparing the Page Views and Controller Servlet

Tutorial Contents

8. Managing Sessions
9. Integrating Transactional Business Logic
10. Adding Language Support (Coming Soon)
11. Securing the Application (Coming Soon)
12. Load Testing the Application (Coming Soon)
13. Conclusion

This tutorial unit demonstrates how to create project files in the IDE, and introduces you to some of the facilities available for HTML and CSS development. After creating necessary project files, you begin organizing the front-end of the application. That is, you'll place JSP files in their proper locations within the project structure, create a header and footer which will be applied to all views, and set up the controller servlet to handle incoming requests.

In this unit, you also create a web deployment descriptor (`web.xml` file) for the application. You can use the deployment descriptor to specify configuration information which is read by the server during deployment. Although the Servlet 3.0 specification, included in Java EE 6, enables you to use class annotations in place of XML, you may still require the deployment descriptor to configure certain elements of your application. Specifically, in this unit you add directives for the header and footer and specify which files they will be applied to.

One of the goals of this tutorial unit is to create JSP pages that correspond to the views specified in the application design. Referring back to the page mockups and process flow diagram, you begin implementing page layouts according to the mockups by creating *placeholders* for all visual and functional components. This unit provides a guide for implementing the layout of the welcome page. You can apply the outlined steps to create the other pages on your own, or download project snapshot 1, which provides completed layouts for all pages.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
GlassFish server	v3 or Open Source Edition 3.0.1

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.
- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could download the GlassFish server independently, but the version provided with the NetBeans download has the added benefit of being automatically registered with the IDE.

Creating Project Files

To create new files for your project, access the IDE's File wizard. You can click the New File () button, press Ctrl-N (⌘-N on Mac), or in the Projects window, right-click the folder node that will contain the new file, and choose New > [file-type]. In the following sub-sections, create JSP pages and a stylesheet for the project.

- Creating JSP Pages
- Creating a Stylesheet

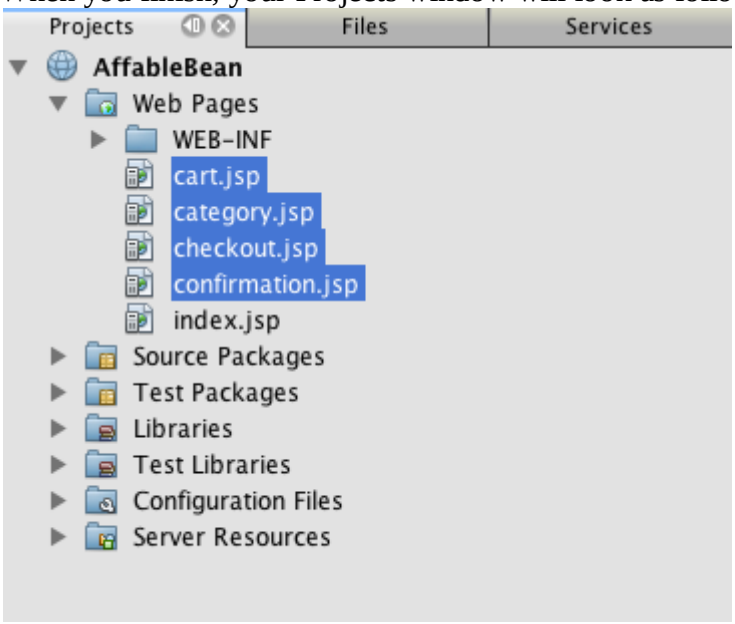
Creating JSP Pages

Begin working in the project by creating JSP pages that correspond to the views displayed in the process flow diagram.

The `index.jsp` page that was generated by the IDE will become the project's welcome page. Create JSP pages for the four remaining views and, for now, place them in the project's webroot with `index.jsp`.

1. Click the New File () button to open the File wizard.
2. Select the Web category, then select JSP and click Next.
3. Name the file '`category`'. Note that the Location field is set to `Web Pages`, indicating that the file will be created in the project's webroot. This corresponds to the project's `web` folder, which you can later verify in the IDE's Files window.
4. Click Finish. The IDE generates the new JSP page and opens it in the editor.
5. Repeat steps 1 - 4 above to create the remaining `cart.jsp`, `checkout.jsp`, `confirmation.jsp` pages.

When you finish, your Projects window will look as follows:

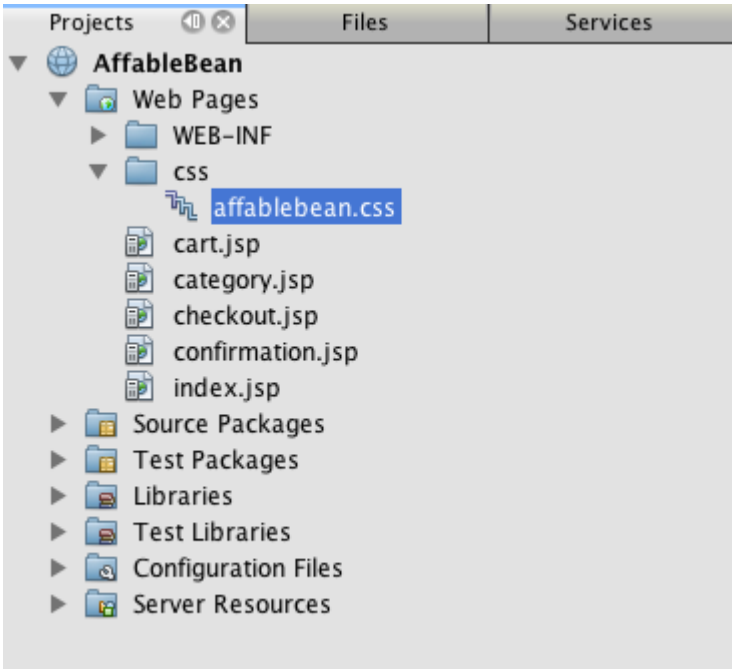


Creating a Stylesheet

Create a CSS file to contain all styles specific to the application.

1. In the Projects window, right-click the Web Pages node and choose New > Folder.
2. In the New Folder wizard, name the folder `css` and click Finish.
3. Right-click the new `css` folder and choose New > Cascading Style Sheet. (If the Cascading Style Sheet item is not listed, choose Other. In the File wizard, select the Web category, then select Cascading Style Sheet and choose Next.)
4. Name the stylesheet `affablebean`, then click Finish.

   When you finish, you'll see the `affablebean.css` file displayed in your Projects window.



Implementing HTML and CSS content

The purpose of this section is to design the page views so that they begin to mirror the provided page mockups. As such, they'll serve as a scaffolding which you can use to insert dynamic content during later stages of project development. To do so, you'll utilize the IDE's HTML and CSS editors, along with several CSS support windows.

Browser compatibility note: This tutorial uses Firefox 3 and *does not* guarantee that page view markup is compatible with other modern browsers. Naturally, when working with front-end web technologies (HTML, CSS, JavaScript) you would need take measures to ensure that your web pages render properly in the browsers and browser versions that you expect visitors to your site will be using (typically Internet Explorer, Firefox, Safari, Chrome, and Opera). When working in the IDE, you can set the browser you want your application to open in. Choose Tools > Options (NetBeans > Preferences on Mac), and under the General tab in the Options window, select the browser you want to use from the Web Browser drop-down. The IDE detects browsers installed to their default locations. If a browser installed on your computer is not displayed, click the Edit button and register the browser manually.

Preparing the display of your web pages is usually an iterative process which you would fine-tune with regular feedback from the customer. The following steps are designed to introduce you to the facilities provided by the IDE, and demonstrate how to get started using the welcome page mockup as an example.

1. In the Projects window, double-click `index.jsp` to open it in the editor.
2. Begin by creating `<div>` tags for the main areas of the page. You can create five tags altogether: four for main areas (header, footer, left column, and right column), and the fifth to contain the others. Remove any content within the `<body>` tags and replace with the following. (New code is shown in **bold**.)

```
<body>
    <div id="main">
        <div id="header">
            header
        </div>

        <div id="indexLeftColumn">
            left column
        </div>

        <div id="indexRightColumn">
            right column
        </div>

        <div id="footer">
            footer
        </div>
    </div>
</body>
```

21. Add a reference to the stylesheet in the page's head, and change the title text.

```
<head>
```

```
23.      <meta http-equiv="Content-Type" content="text/html;
   charset=UTF-8">
24.      <link rel="stylesheet" type="text/css"
   href="css/affablebean.css">
25.      <title>The Affable Bean</title>
   </head>
```

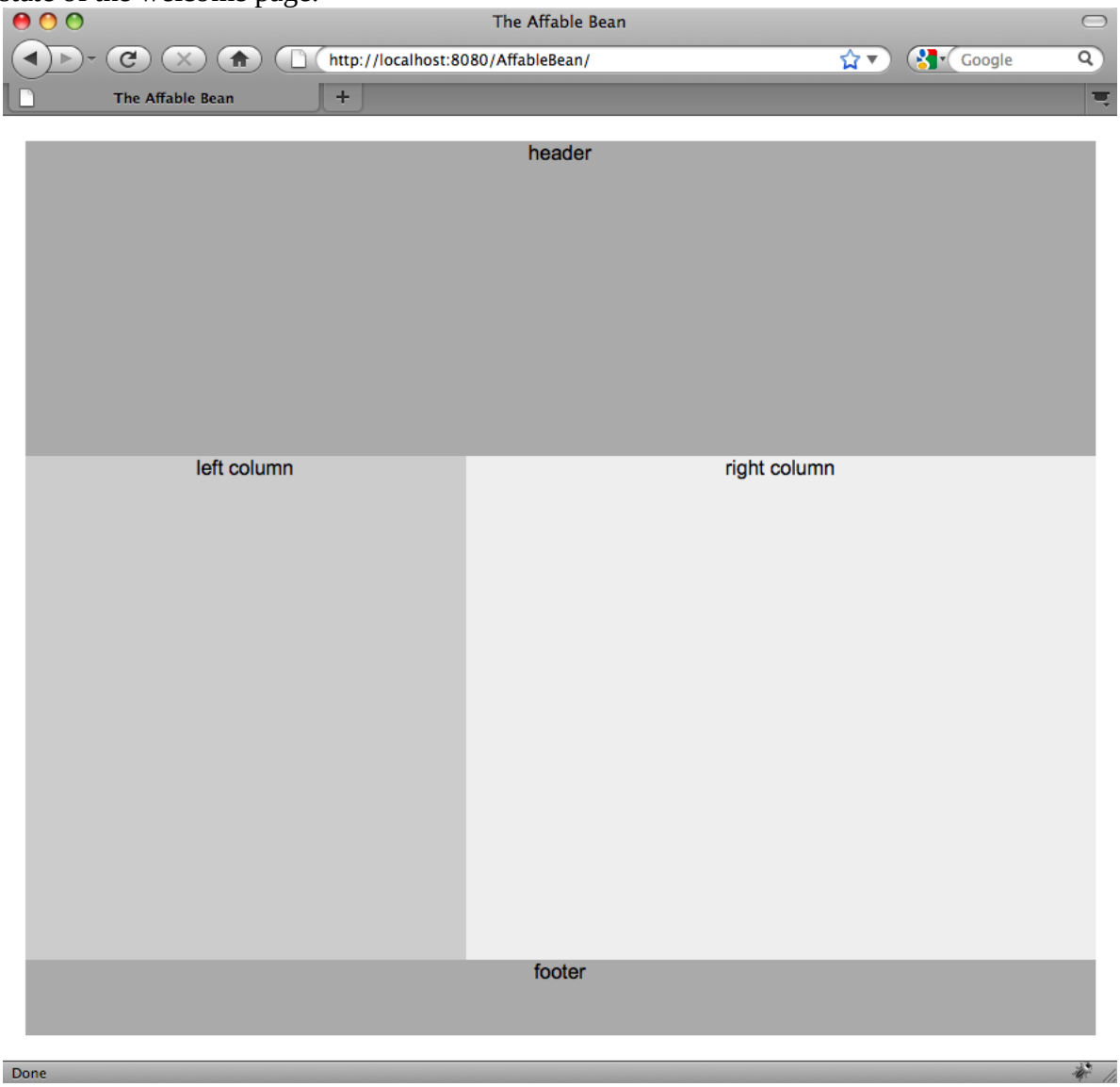
26. Open the `affablebean.css` stylesheet in the editor. Begin creating style rules for the `<div>` IDs you just created.

- Use the `width` and `height` properties to create space for each area.
- Use the `background` property to discern the areas when you view the page.
- In order to horizontally center the four areas in the page, you can include `margin: 20px auto` to the `body` rule. (`20px` applies to the top and bottom; `auto` creates equal spacing to the left and right.) Then include `float: left` to the left and right columns.
- The footer requires `clear: left` so that its top border displays after the bottom borders of any left-floating areas above it (i.e., the left and right columns).

```
27.  body {
28.      font-family: Arial, Helvetica, sans-serif;
29.      width: 850px;
30.      text-align: center;
31.      margin: 20px auto;
32.  }
33.
34.  #main { background: #eee }
35.
36.  #header {
37.      height: 250px;
38.      background: #aaa;
39.  }
40.
41.  #footer {
42.      height: 60px;
43.      clear: left;
44.      background: #aaa;
45.  }
46.
47.  #indexLeftColumn {
48.      height: 400px;
49.      width: 350px;
50.      float: left;
51.      background: #ccc;
52.  }
53.
54.  #indexRightColumn {
55.      height: 400px;
56.      width: 500px;
57.      float: left;
58.      background: #eee;
   }
```

59. Click the Run Project ( ) button in the IDE's main toolbar. Project files that contain changes are automatically saved, any Java code in the project compiles, the project is packaged and deployed to GlassFish, and your browser opens to display the current

state of the welcome page.



60. Now, begin creating placeholders for page components within each of the four visible areas. Start with the header. Reviewing the welcome page mockup, the header should contain the following components:

- logo
- logo text
- shopping cart widget
- language toggle

Make the following changes to the `index.jsp` file. (New code shown in **bold**.)

```
<div id="header">
    <div id="widgetBar">

        <div class="headerWidget">
            [ language toggle ]
        </div>

        <div class="headerWidget">
            [ shopping cart widget ]
        </div>
```

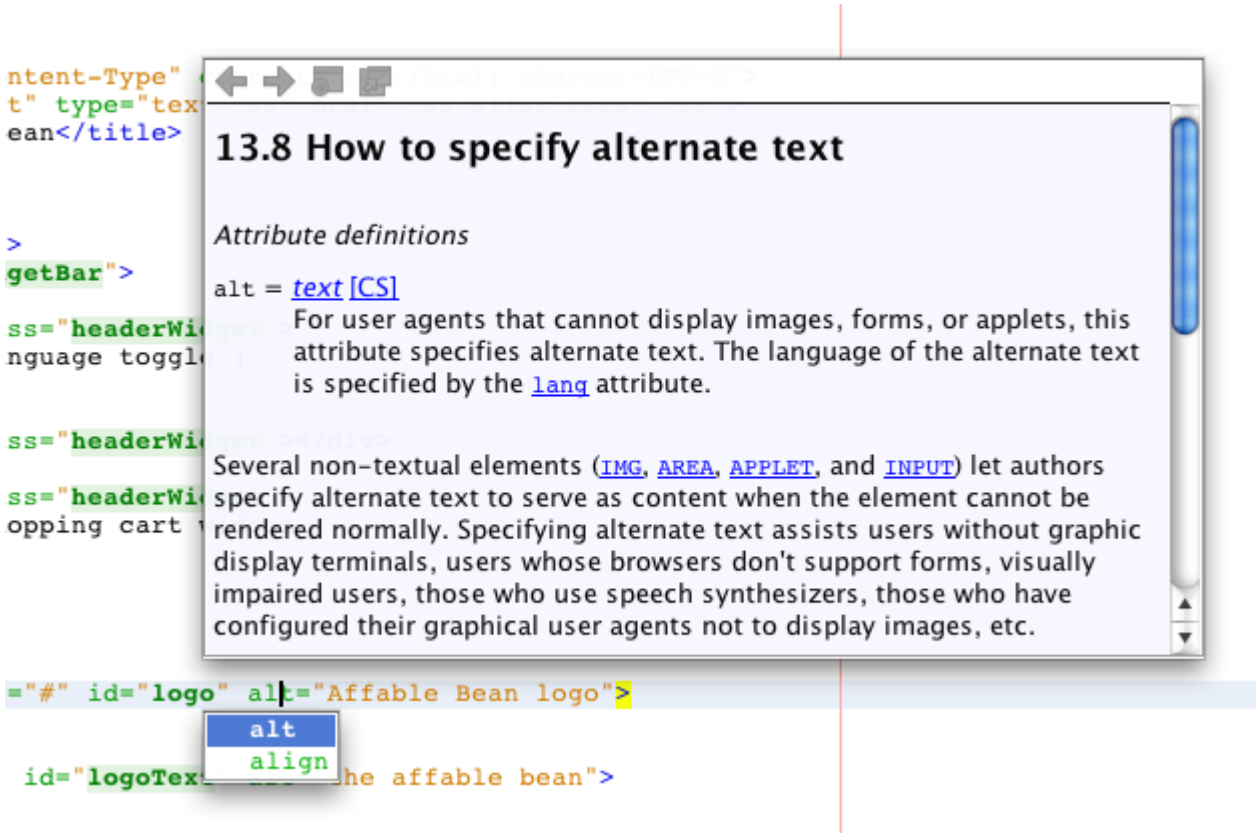
```
    </div>

    <a href="#">
        <img src="#" id="logo" alt="Affable Bean logo">
    </a>

    <img src="#" id="logoText" alt="the affable bean">
</div>
```

In the above code, you use a `<div id="widgetBar">` element to contain the the language toggle and shopping cart widget.

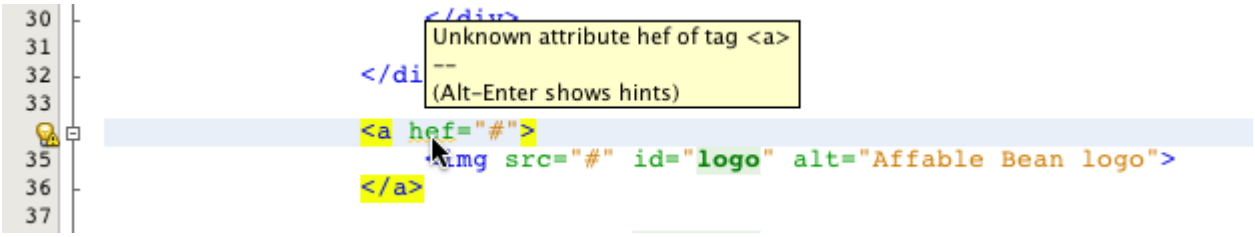
NetBeans HTML Editor Support

When you work in the editor, take advantage of the IDE's HTML support. Aside from typical syntax highlighting that lets you differentiate between tags, attributes, attribute values, and text, there are plenty of other features.

When typing tags and attributes in the editor, you can invoke code-completion and documentation support by pressing Ctrl-Space. The IDE presents a list of suggestions which you can choose from, as well as a documentation window that defines the selected item and provides code examples.



The IDE detects errors in your code and provides you with warnings, error messages, and in some cases, suggestions. Warning messages are displayed in yellow, while errors are shown in red. You can hover your pointer over a designated area to view the message in a tooltip.



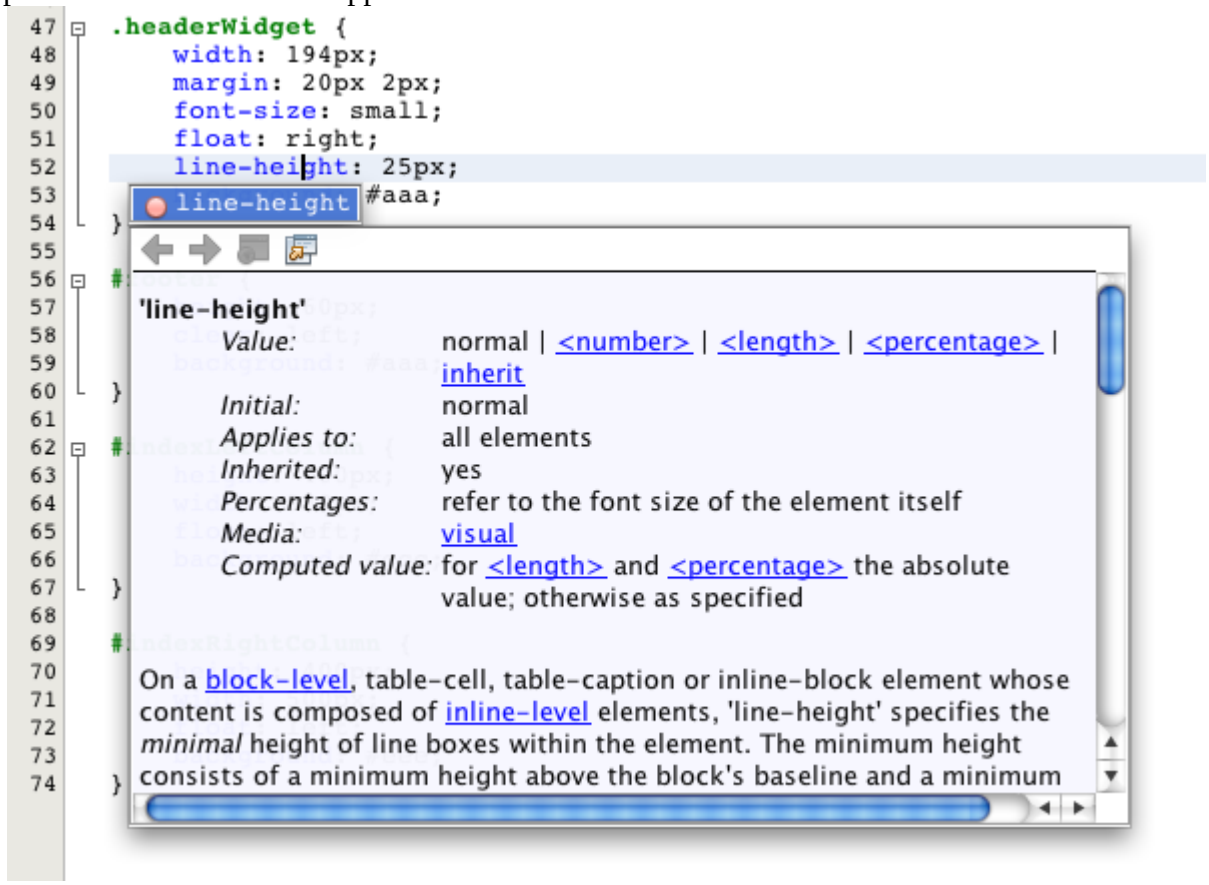
You can also take advantage of numerous keyboard shortcuts. Choose Help > Keyboard Shortcuts Card from the main menu.

61. In the stylesheet, create rules for the new IDs and classes. Add the following rules beneath the `header` rule. (New code shown in **bold**.)

```
#header {
    height: 250px;
    background: #aaa;
}

#logo {
    height: 155px;
    width: 155px;
    float: left;
    margin-left: 30px;
    margin-top: -20px;
}

#logoText {
    float: left;
    margin: 20px 0 0 70px;
    /* font styles apply to text within alt tags */
    font-family: 'American Typewriter', Courier, monospace;
    font-size: 50px;
    color: #333;
}

#widgetBar {
    height: 50px;
    width: 850px;
    float: right;
    background: #ccc;
}

.headerWidget {
    width: 194px;
    margin: 20px 2px;
    font-size: small;
    float: right;
    line-height: 25px;
    background: #aaa;
}
```

For the `logo` rule, you apply `margin-left` and `margin-top` properties to position the component on the page.

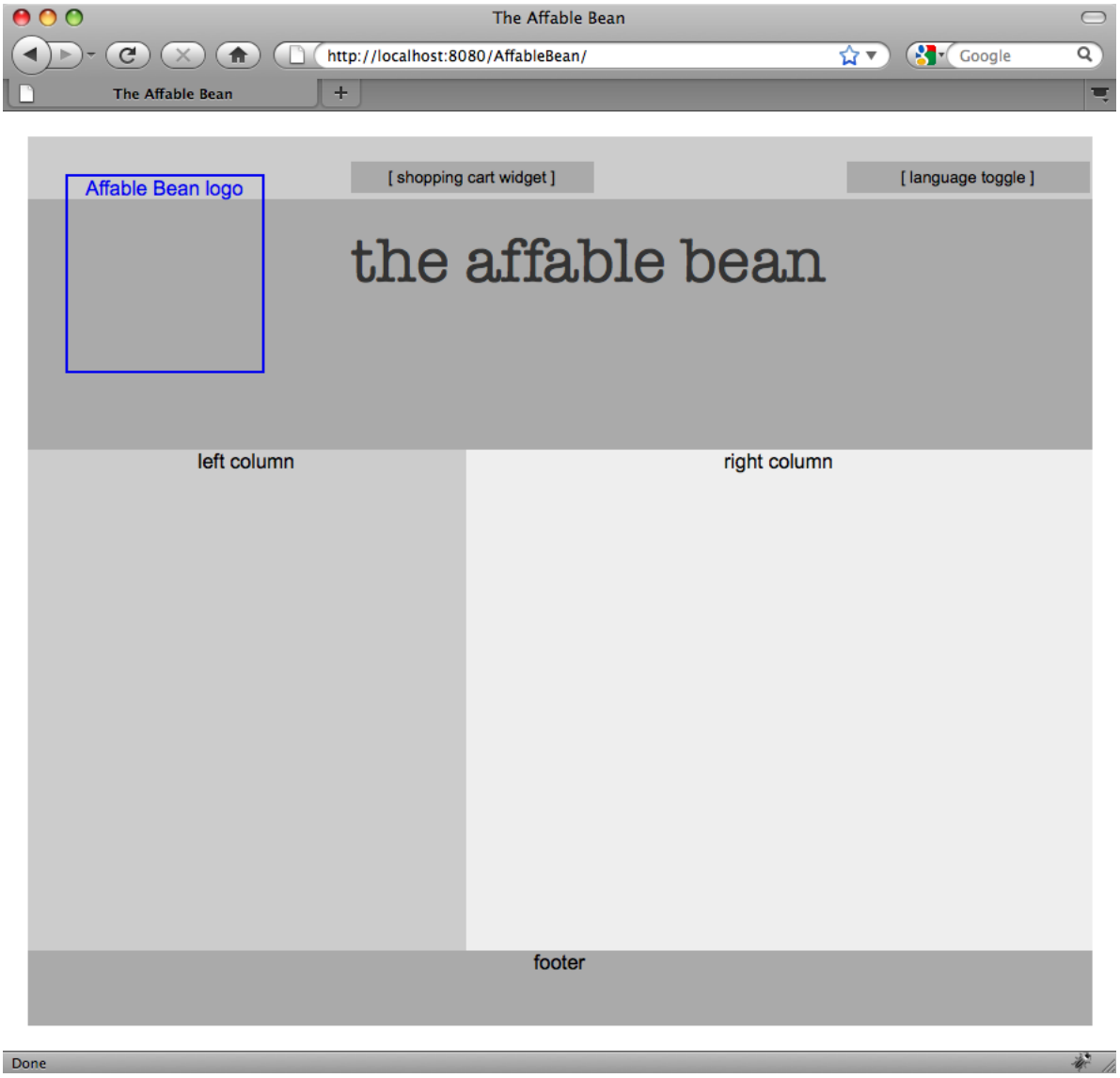
If there are properties in the above code that you are unfamiliar with, position your cursor on the given property and press Ctrl-Space to invoke a pop-up window that provides documentation support.



To see how a property is affecting your page, you can comment it out, then refresh the page in the browser. To comment out code, position your cursor on a line, or highlight a block of code, then press Ctrl-/ (⌘-/ on Mac).

98. Save (Ctrl-S; ⌘-S on Mac) the `index.jsp` and `affablebean.css` files, then switch to your browser and refresh the page to view its current state.

    Note: The IDE's 'Deploy on Save' facility is automatically activated for Java web projects. This means that every time you save a file, the file is automatically compiled (i.e., if it is a Java class or JSP page) and the project is newly packaged and deployed to your server. Therefore, when you make HTML or CSS changes, you don't need to explicitly rerun the project to view the updated version in a browser. Simply save your file(s), then switch to the browser and refresh the page.



By following the previous steps, you are probably able to see a pattern emerging. For each area on the page, you perform three steps.

a. Create the structure in HTML.
b. Create a set of styles to define the appearance.
c. View the page to examine the results of your changes.

Following these three steps, let's implement the components in the remaining areas.

Create placeholders for components in the right column. According to the welcome page mockup, the right column contains four evenly-spaced boxes.

Create the structure for the four boxes. Insert the following code between the `<div id="indexRightColumn">` tags. (New code shown in **bold**.)

```
        <div id="indexRightColumn">
            <div class="categoryBox">
                <a href="#">
                    <span class="categoryLabelText">dairy</span>
                </a>
```

```
            </div>
            <div class="categoryBox">
                <a href="#">
                    <span class="categoryLabelText">meats</span>
                </a>
            </div>
            <div class="categoryBox">
                <a href="#">
                    <span class="categoryLabelText">bakery</span>
                </a>
            </div>
            <div class="categoryBox">
                <a href="#">
                    <span class="categoryLabelText">fruit & veg</span>
                </a>
            </div>
        </div>
```

Add style rules to `affablebean.css` for the new `categoryBox` and `categoryLabelText` classes. (New code shown in **bold**.)

```
    #indexRightColumn {
        height: 400px;
        width: 500px;
        float: left;
        background: #eee;
    }

    .categoryBox {
        height: 176px;
        width: 212px;
        margin: 21px 14px 6px;
        float: inherit;
        background: #ccc;
    }

    .categoryLabelText {
        line-height: 150%;
        font-size: x-large;
}
```

NetBeans CSS Support

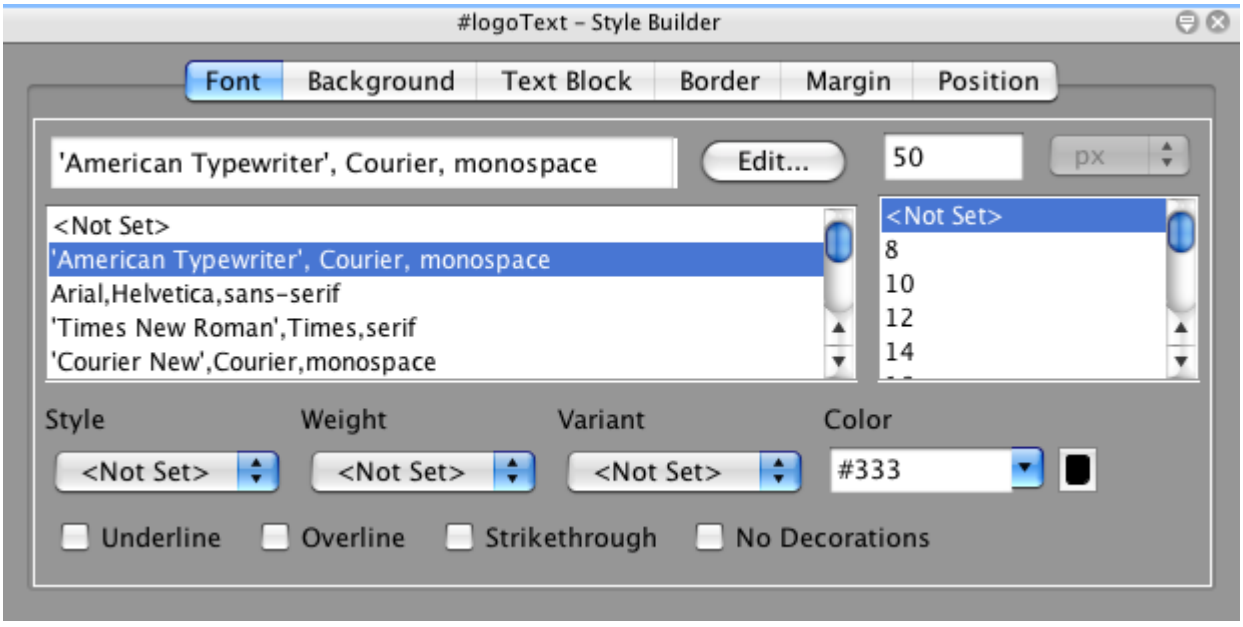
When working in stylesheets, there are two windows that can be particularly helpful. The CSS Preview enables you to view style rules as they are rendered in a browser. To open the CSS Preview, choose Window > Other > CSS Preview from the main menu. When you place your cursor within a style rule in the editor, the CSS Preview automatically refreshes to display sample text according to the properties defined in the rule.

```
31  #logoText {
32      float: left;
33      margin: 20px 0 0 70px;
34      /* font styles apply to text within img alt attribute */
35      font-family: 'American Typewriter', Courier, monospace;
36      font-size: 50px;
37      color: #333;
38  }
39
```

#logoText – Preview

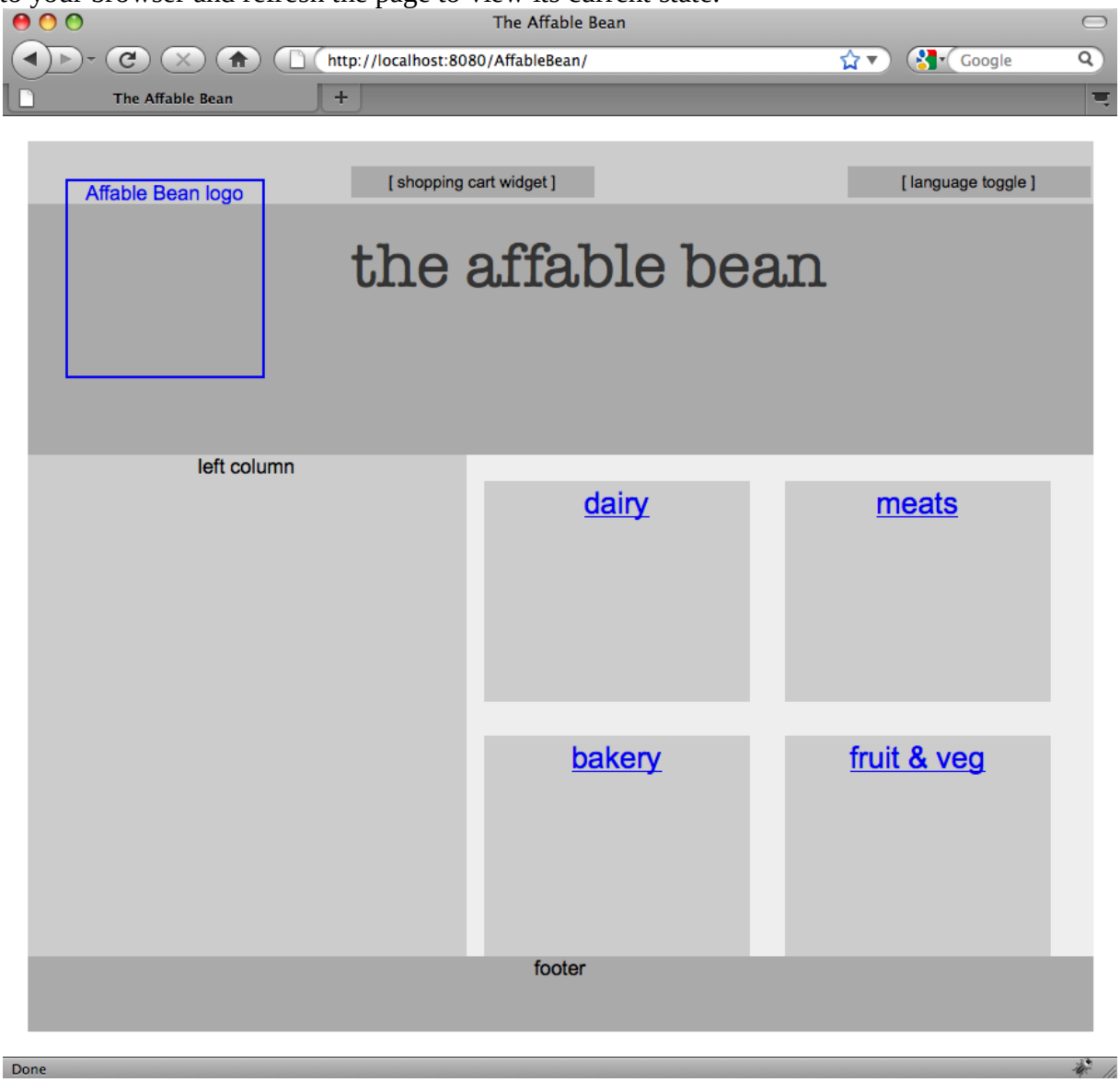
Sample Text

The CSS Style Builder is useful if you do not like to code style rules by hand. To open the CSS Style Builder, choose Window > Other > CSS Style Builder from the main menu. Using this interface, you can construct rules by choosing properties and values from a graphical interface.



Like the CSS Preview, the Style Builder is synchronized with the editor. When you make a selection in the Style Builder, the style rule is automatically updated in the editor. Likewise, when you type changes into the editor, the selections in the Style Builder are instantly updated.

Save (Ctrl-S; ⌘-S on Mac) the `index.jsp` and `affablebean.css` files, then switch to your browser and refresh the page to view its current state.



The left column and footer only require placeholders for static text, so let's implement both simultaneously.

Insert the following code between the `<div id="indexLefttColumn">` and `<div id="footer">` tags. (New code shown in **bold**.)

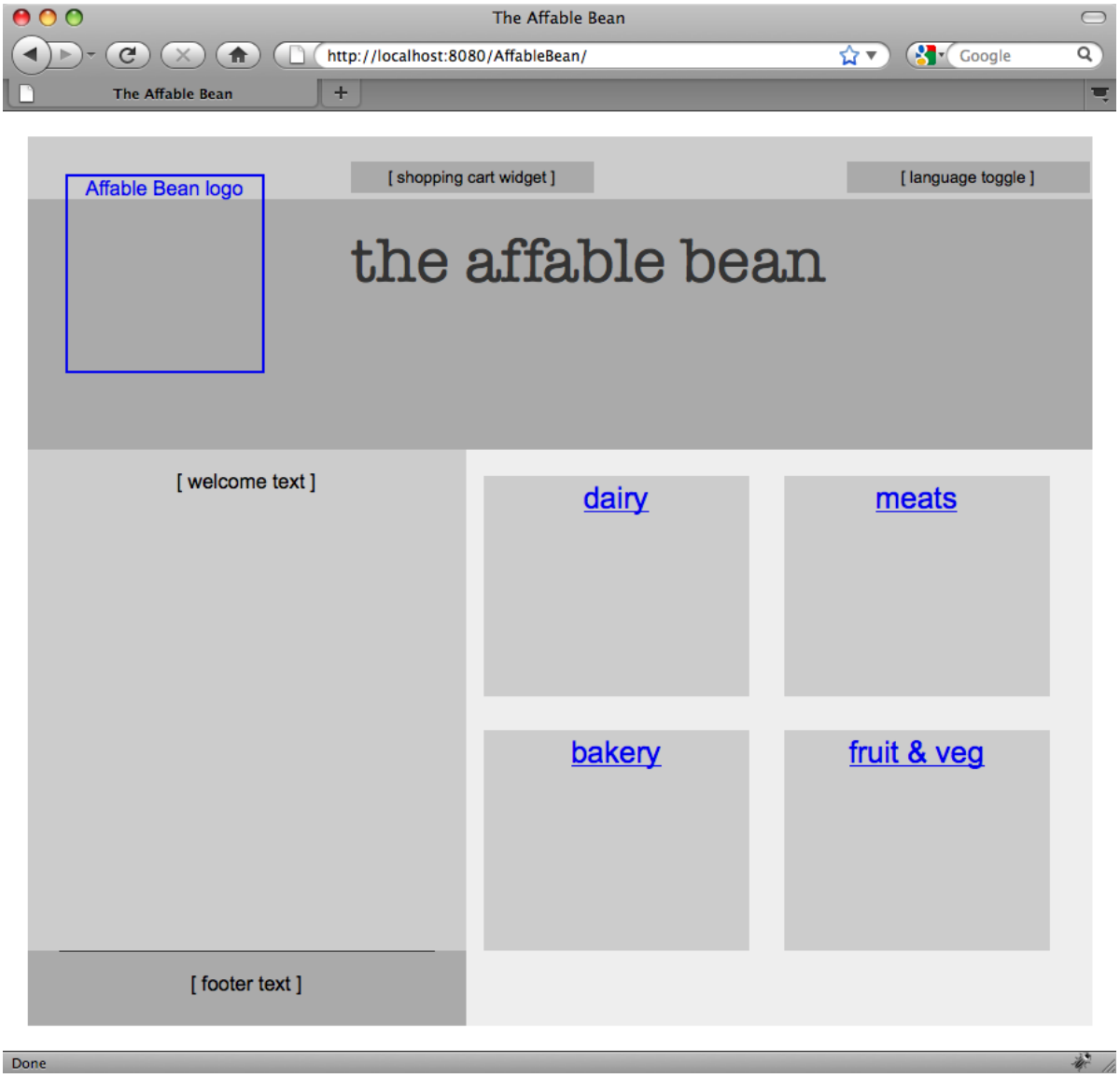
```
        <div id="indexLeftColumn">
            <div id="welcomeText">
                <p>[ welcome text ]</p>
            </div>
        </div>

        ...

        <div id="footer">
            <hr>
            <p id="footerText">[ footer text ]</p>
    </div>
```

Make changes to the `affablebean.css` stylesheet. It's not necessary to account for all new IDs and classes - you can fine-tune the appearance at a later point when you receive text and images from the customer.

The horizontal rule (`<hr>`) tag runs the full length of its containing element (`<div id="footer"`). Therefore, to shorten it in accordance with the mockup image, you can adjust the width of `<div id="footer">`. (New code shown in **bold**.)

```
#footer {
    height: 60px;
    width: 350px;
    clear: left;
    background: #aaa;
}

hr {
    border: 0;
    background-color: #333;
    height: 1px;
    margin: 0 25px;
    width: 300px;
}
```

Save (Ctrl-S; ⌘-S on Mac) the `index.jsp` and `affablebean.css` files, then switch to your browser and refresh the page to view its current state.



The welcome page is complete. You've created all necessary placeholders for components that will exist on the page.

You've now completed the initial design of the application's welcome page. All placeholders for page components exist. Later in the tutorial, when you begin to apply dynamic logic to the page views, you can simply plug JSTL and EL expressions into these placeholders.

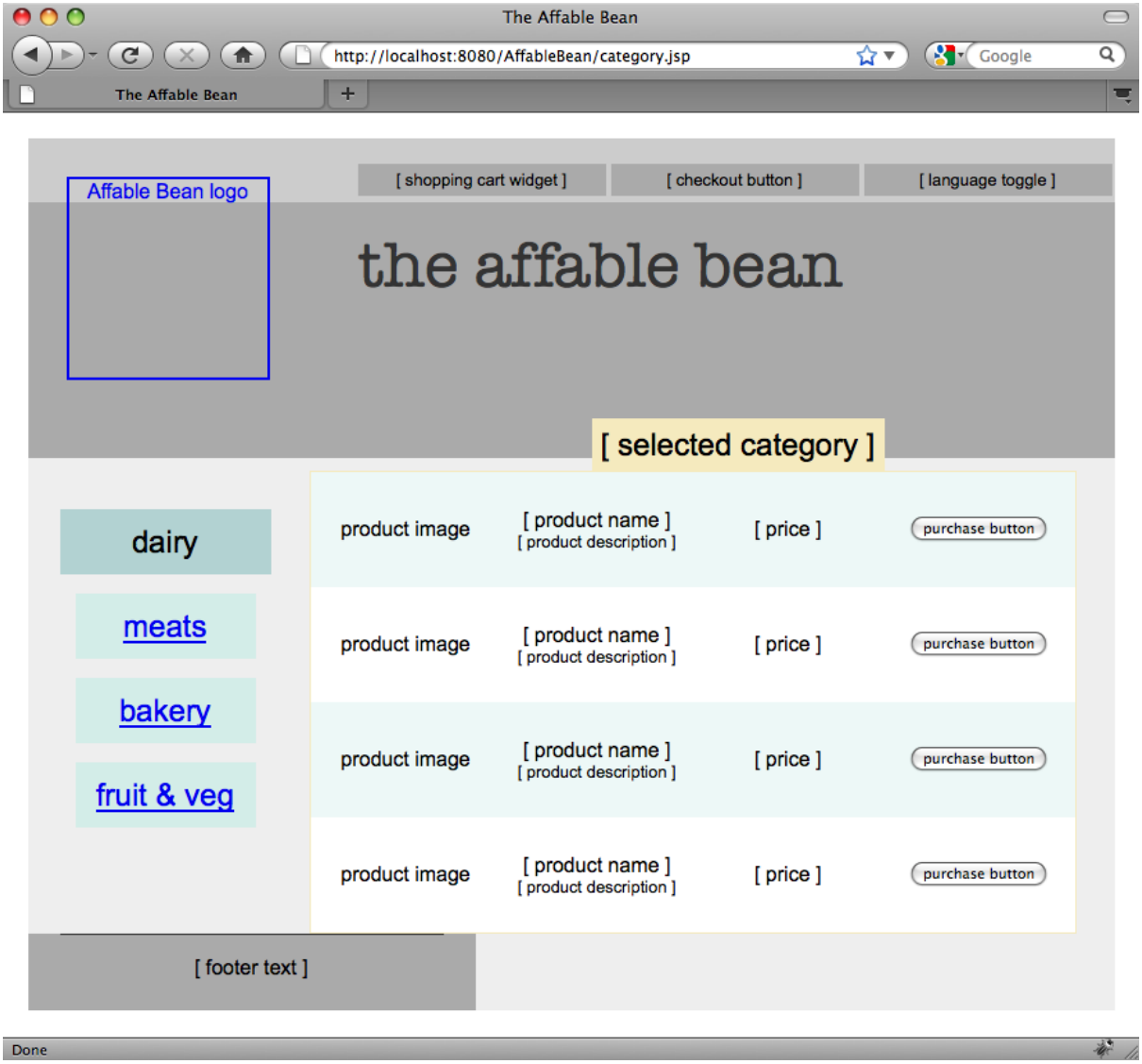
The task remains for you to implement the initial design for the other pages based on the mockups. To accomplish this, follow the pattern outlined above, namely:

1. Create `<div>` tags for the main page areas.
2. Iterate through each area and perform three steps:
    1. Create the structure in HTML.
    2. Create a set of styles to define the appearance.
    3. View the page to examine the results of your changes.

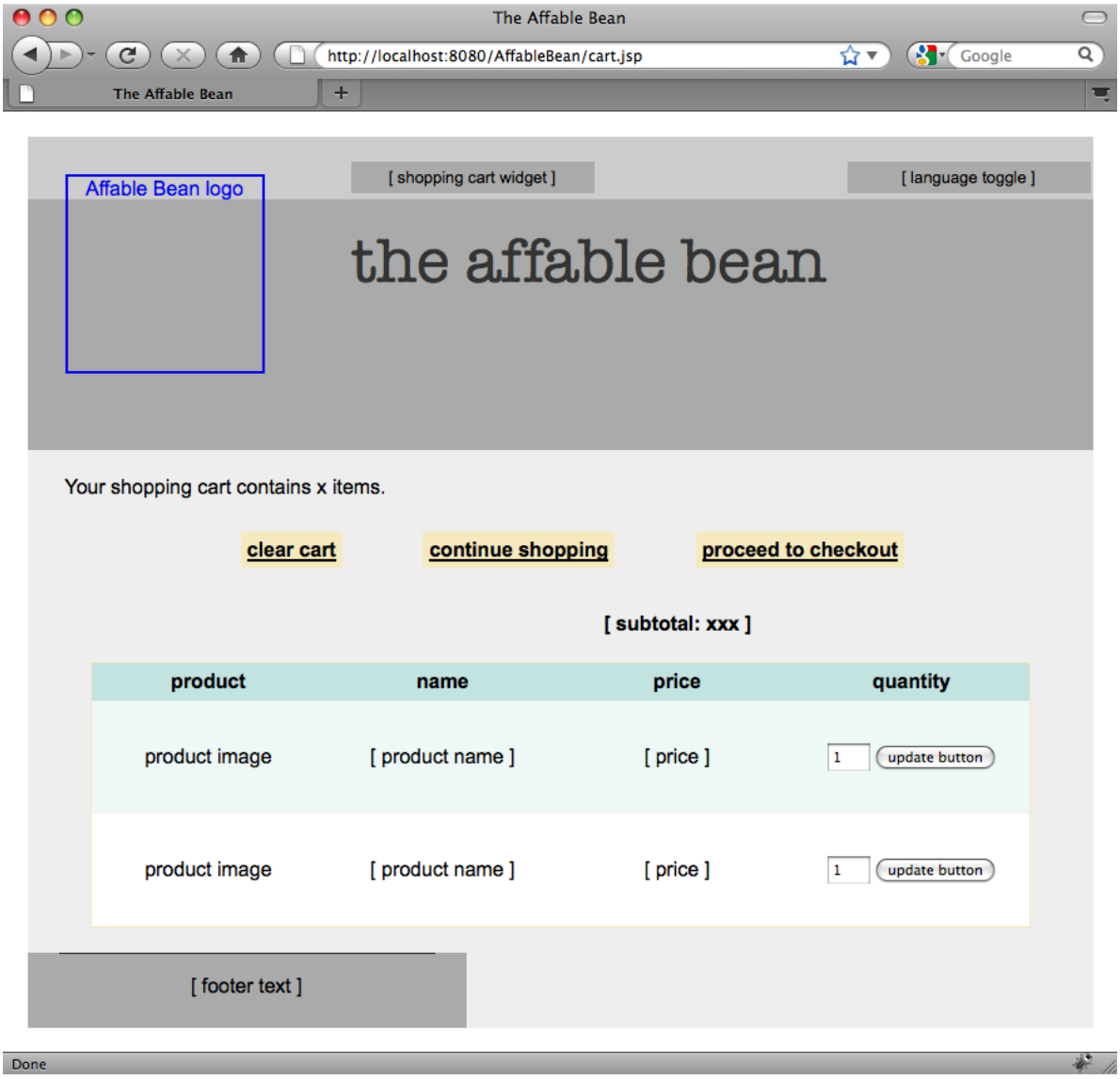
Be sure to take advantage of the HTML and CSS support that the IDE provides for you. Some tips and tricks are outlined below. If you just want to grab the code for the remaining

pages and proceed with the tutorial, you can download snapshot 1 of the `AffableBean` project. Images of initial mockup implementations for the remaining pages are included here.

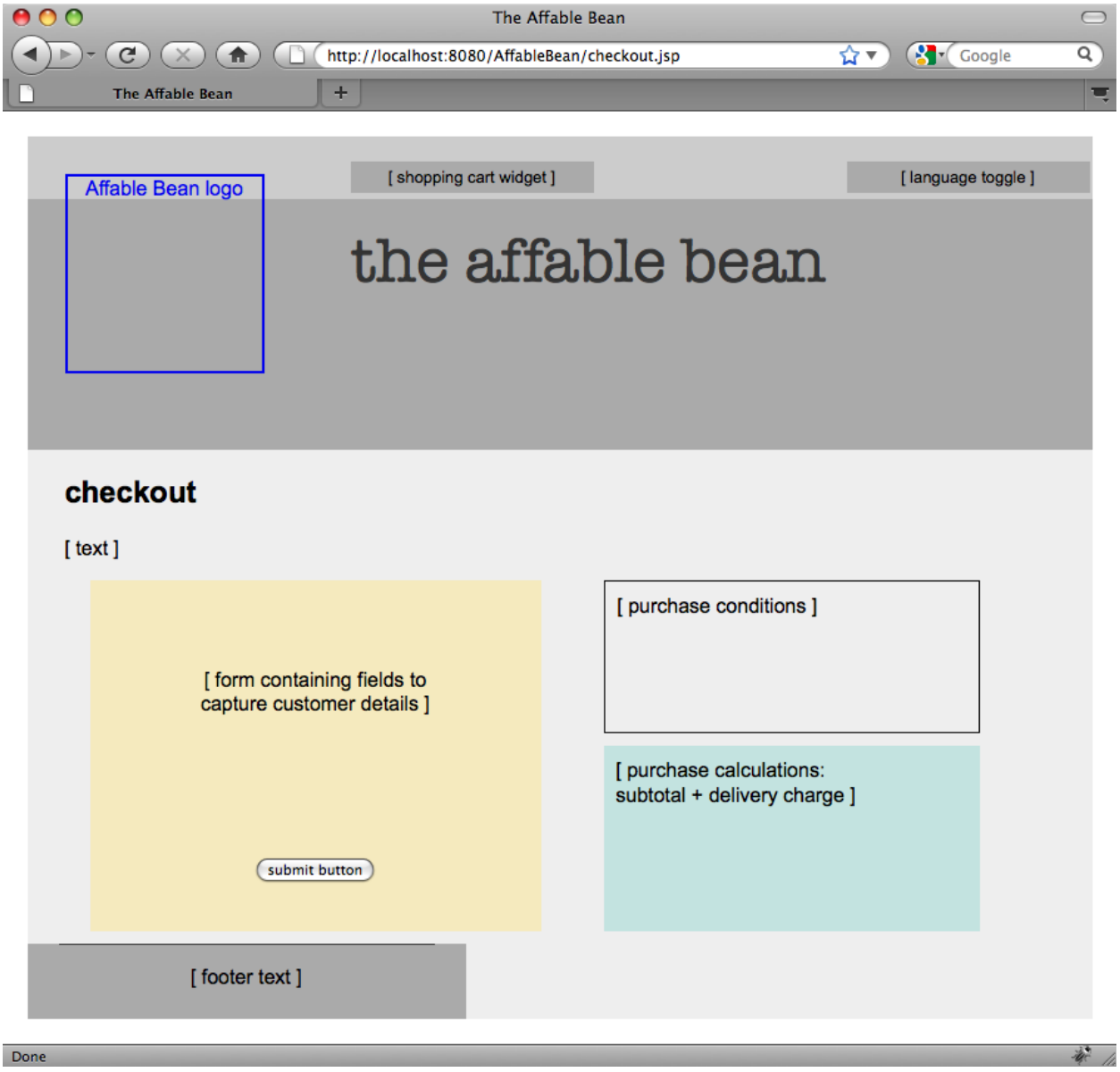
category page



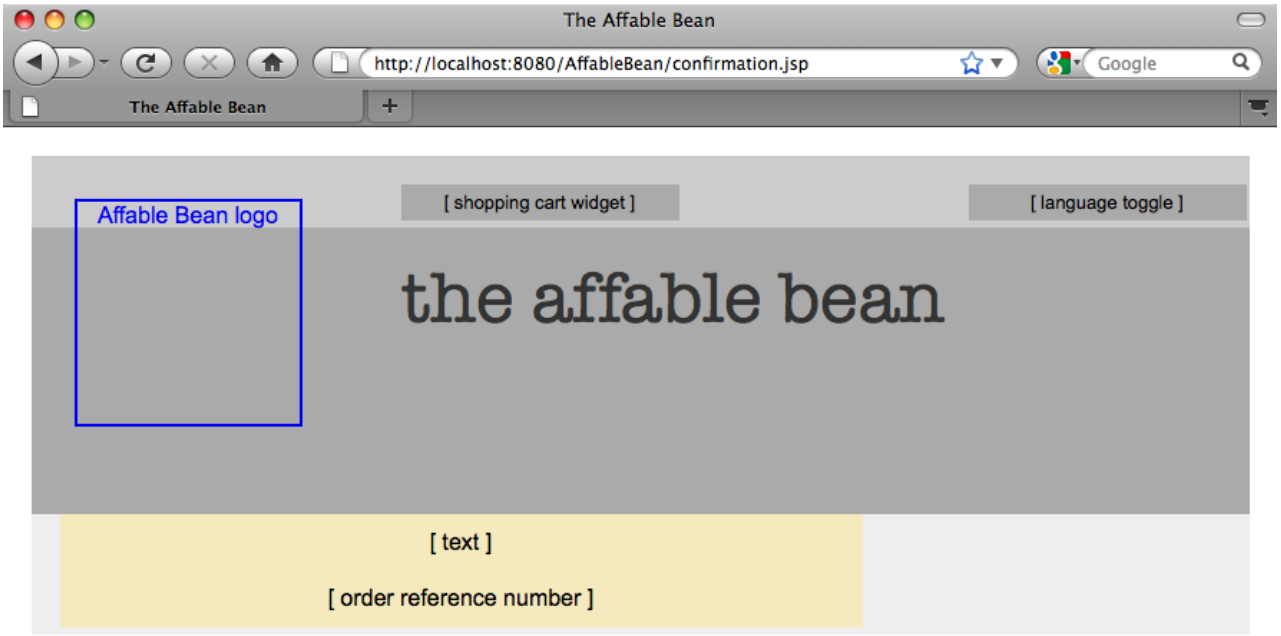
cart page



checkout page



confirmation page



Note: The background colors for each page area only serve to help you position elements while developing the application. Eventually, you'll want to remove them from the stylesheet and apply a background color more suitable for the application. You can do this by adjusting the background rule for the `main` class:

```
#main { background: #f7f7e9 }
```

Tips and Tricks

The IDE's editor provides many facilities that help you to work more efficiently. If you familiarize yourself with keyboard shortcuts and buttons in the editor toolbar, you can increase your productivity. The following list of tips applies to the editor for HTML and CSS files. To view more keyboard shortcuts, open the IDE's Keyboard Shortcuts Card by choosing Help > Keyboard Shortcuts Card from the main menu.

- **Code completion:** When you type in tags and attributes, suggestions for code completion automatically appear in a pop-up box. Pressing Enter completes the suggested tag.
- **Format your code:** Right-click in the editor and choose Format.
- **Toggle line numbers:** Right-click in the left margin and choose Show Line Numbers.

- **Find occurrences:** Highlight a block of text, and press Ctrl-F (⌘-F on Mac). All matches become highlighted in the editor. To toggle highlighting, press the Toggle Highlight Search () button (Ctrl-Shift-H) in the editor's toolbar.
- **Create a bookmark:** Press the Toggle Bookmark () button (Ctrl-Shift-M) to create a bookmark in the editor's left margin. Wherever you are in the file, you can then jump to the bookmark by pressing the Previous/Next Bookmark buttons in the editors's toolbar.
- **Copy a code snippet up or down:** Highlight a code snippet, then press Ctrl-Shift-Up/Down.
- **Highlight opening and closing tags:** Place your cursor on either the opening or closing tag, and both are highlighted in yellow.

Placing JSP Pages in WEB-INF

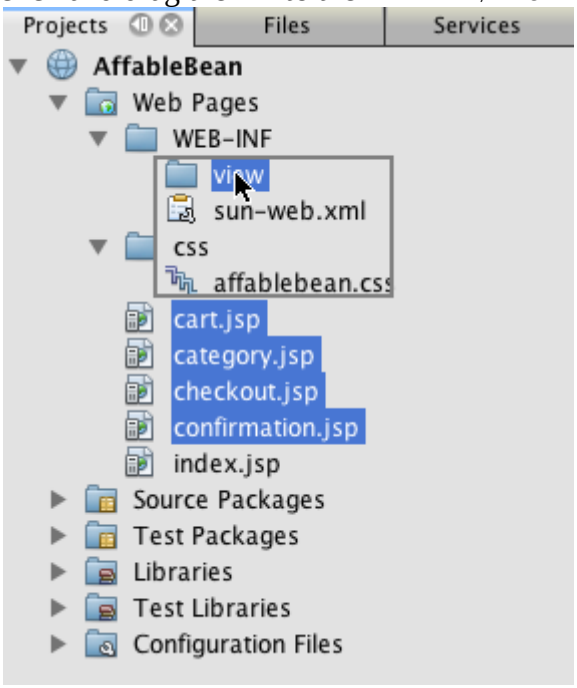
Looking back at the page mockups that were created, you can see that the welcome page should look the same whenever it is requested, for whomever requests it. That is, the content that displays on the welcome page is not determined by a user's *session*. (Sessions are discussed in Unit 8, Managing Sessions.) Notice however that all other pages do need some form of user-specific information to display properly. For example, the category page requires that the user select a category in order to display, and the cart page needs to know all items currently held in a shopper's cart. These pages will not render properly if the server isn't able to associate user-specific information with an incoming request. Therefore, we do not want these pages to be accessed directly from a browser's address bar. The project's `WEB-INF` folder can be used for this purpose: any resources contained in the `WEB-INF` folder are not directly accessible from a browser.

Create a new folder named `view`, and place it in the `WEB-INF` folder. Then move all JSP pages other than the welcome page into this new folder.

1. In the Projects window, right-click the WEB-INF node and choose New > Folder.
2. In the New Folder wizard, name the folder `view` and click Finish. Notice that a new folder node appears in the Projects window.
3. Move the `category.jsp`, `cart.jsp`, `checkout.jsp`, and `confirmation.jsp` pages into the `view` folder.

   You can do this by clicking on `cart.jsp` to select it, then Shift-clicking on `confirmation.jsp`. This selects the four files. Then, with the four files selected,

click and drag them into the `WEB-INF/view` folder.



To demonstrate that these pages are no longer accessible from a browser, click the Run Project ( ) button to run the project. When the application displays in your browser, enter the full path to any of these files in the address bar. For example, type in:

```
http://localhost:8080/AffableBean/WEB-INF/view/category.jsp
```

You receive an HTTP Status 404 message, indicating that the resource is not available.

Creating a Header and Footer

Looking at the page mockups, it is easy to see that all of the five views share identical content; at the top, they contain the company logo, a language toggle, and other widgets associated with shopping cart functionality. At the bottom, they contain some text with Privacy Policy and Contact links. Rather than including this code in each page source file, we can factor it out into two JSP fragments: a header and a footer. We'll then include the fragment files into page views whenever they need to be rendered.

For these fragments, let's create a new folder named `jspf`, and place it within `WEB-INF`.

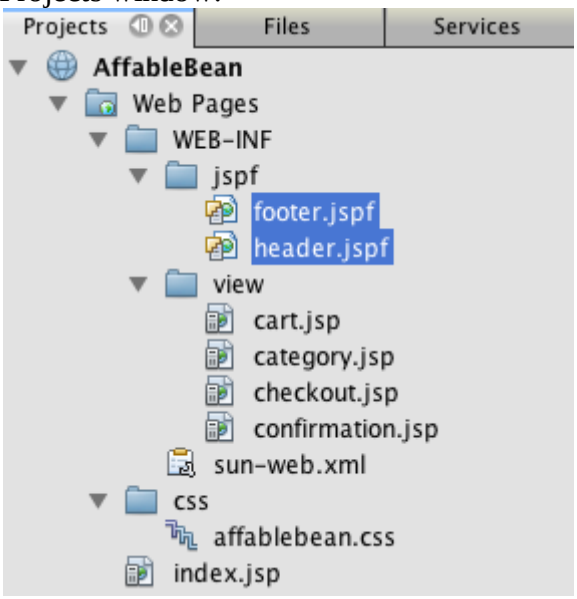
1. In the Projects window, right-click the WEB-INF node and choose New > Folder.
2. In the New Folder wizard, name the folder `jspf` and click Finish.

   Menu items provided by the IDE are often context-sensitive. For example, because you right-clicked the WEB-INF node, when the New Folder wizard displayed, `web/WEB-INF` was automatically entered in the Parent Folder field. Likewise, when

you right-click a node in the Projects window and choose New, the list of file types is partially determined by your previous selections.

3. Create two JSP segments: `header.jspf` and `footer.jspf`. To do so, right-click the newly created `jspf` folder and choose New > JSP. In the New JSP wizard, enter the file name, and under Options, select the Create as a JSP Segment option, then click Finish.

   When you finish, you'll see `header.jspf` and `footer.jspf` displayed in your Projects window:



   Now, you can copy the header code from any of the JSP pages and paste it into the `header.jspf` file. Likewise, you can copy the footer code from any of the JSP pages and paste it into the `footer.jspf` file. When you finish this task, you can remove the header and footer code from all of the JSP pages.
4. Copy the header code from any of the JSP pages and paste it into the `header.jspf` file. The header should include the page doctype and the opening `<html>`, `<head>`, and `<body>` tags through to the closing tag for the `<div id="header">` element. Be sure to include placeholders for the shopping cart widget, language toggle, and 'proceed to checkout' button used along the top of page views. After you paste code into `header.jspf`, the file will look as follows.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
        <link rel="stylesheet" type="text/css"
href="css/affablebean.css">
        <title>The Affable Bean</title>
    </head>
    <body>
        <div id="main">
            <div id="header">
                <div id="widgetBar">
```

```
19.
20.                     <div class="headerWidget">
21.                         [ language toggle ]
22.                     </div>
23.
24.                     <div class="headerWidget">
25.                         [ checkout button ]
26.                     </div>
27.
28.                     <div class="headerWidget">
29.                         [ shopping cart widget ]
30.                     </div>
31.
32.                 </div>
33.
34.                 <a href="#">
35.                     <img src="#" id="logo" alt="Affable Bean logo">
36.                 </a>
37.
38.                 <img src="#" id="logoText" alt="the affable bean">
            </div>
```

39. Copy the footer code from any of the JSP pages and paste it into the `footer.jspf` file. The footer code should include the `<div id="footer">` element, through to the closing `<html>` tag. After you paste code into `footer.jspf`, the file will look as follows.

```
40.             <div id="footer">
41.                 <hr>
42.                 <p id="footerText">[ footer text ]</p>
43.             </div>
44.         </div>
45.     </body>
</html>
```

46. Remove the header and footer code from all five JSP pages (`index.jsp`, `category.jsp`, `cart.jsp`, `checkout.jsp`, and `confirmation.jsp`).

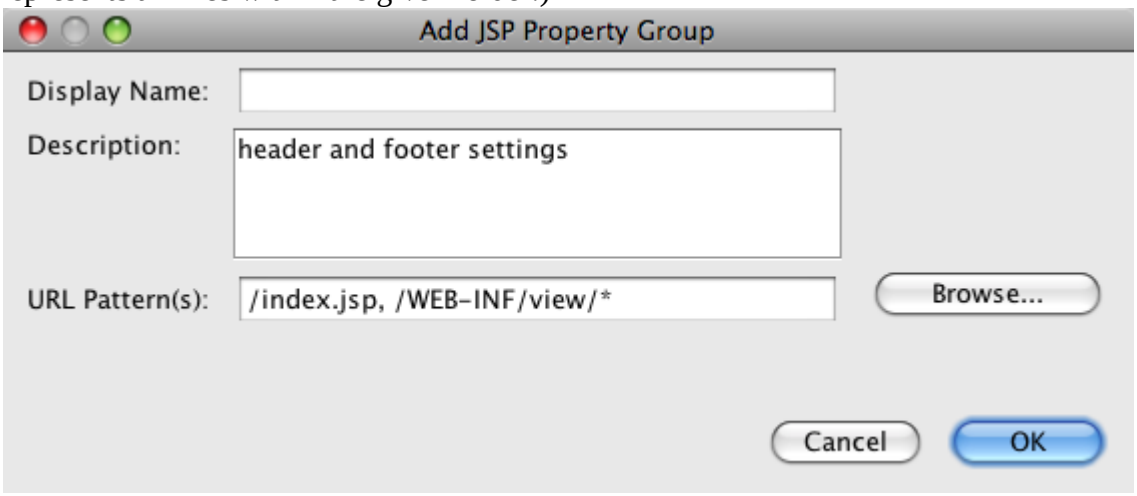
Adding a Directive to the Deployment Descriptor

So far, you've placed views in their proper location and have factored out common header and footer code into the `header.jspf` and `footer.jspf` files. The application still needs to know which pages the header and footer files will be applied to. You could add `<jsp:include>` tags to each of the page views. Doing so however would just reintroduce the code repetition which we've just made efforts to eliminate. An alternative solution would be to create a `web.xml` deployment descriptor, and add a JSP Property Group directive to specify which page views the header and footer fragments should apply to.

1. Press Ctrl-N (⌘-N on Mac) to open the New File wizard. Select the Web category, then under File Types, select Standard Deployment Descriptor (web.xml).
2. Click Next. Note that the file is named `web.xml`, and that the wizard will place it in the project's `WEB-INF` directory upon completion.

3. Click Finish. The `web.xml` file is created and added to the project. The IDE's graphical interface for the deployment descriptor opens in the editor.

   The interface is categorized by the areas that can be configured in a web application. These areas are displayed as tabs in the editor toolbar, and include topics such as Servlets, Filters, References, and Security. The XML tab displays the entire source code for the file. Any changes you make in the graphical interface will cause immediate updates to the deployment descriptor's source code, which you can verify by switching to the XML tab. This is demonstrated in the following steps.
4. Click the Pages tab, then click the Add JSP Property Group button. The Add JSP Property Group dialog opens.
5. Type in '`header and footer settings`' for the Description field. Leave Display Name blank. Both the Display Name and Description fields are optional.
6. For URL Patterns, specify the paths to the five views. Type in '`/index.jsp`' and '`/WEB-INF/view/*`'. Separate the two paths with a comma. (The '`*`' is a wildcard that represents all files within the given folder.)



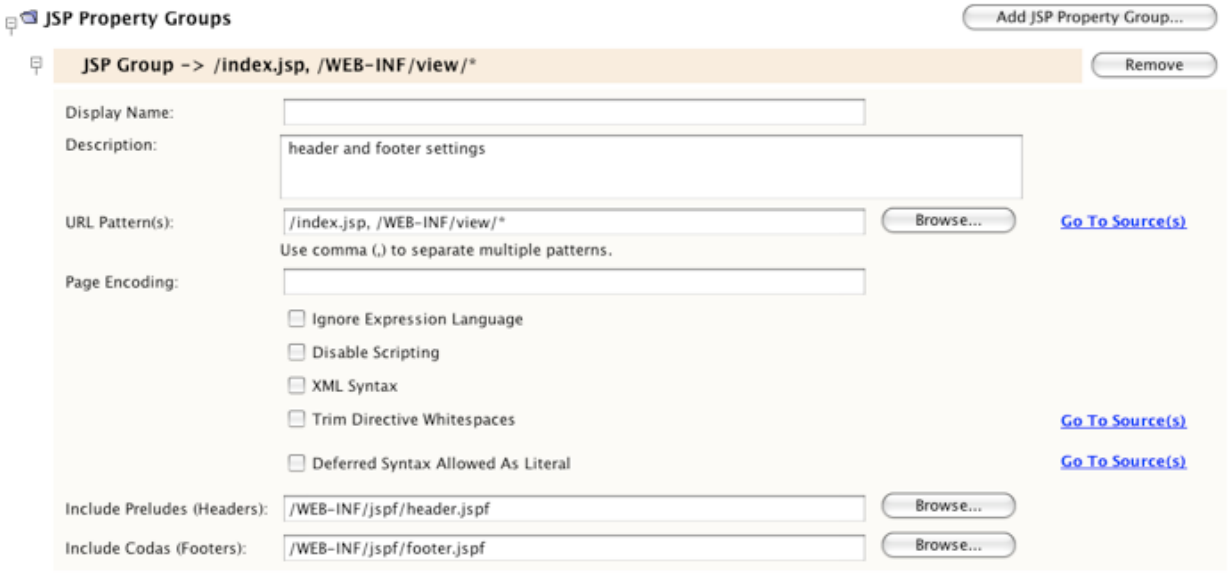
7. Click OK. An entry is added to the JSP Properties Groups category in the Pages tab.
8. Switch back to the XML tab. Notice that the following code has been added to the deployment descriptor.

```
9. <jsp-config>
10.     <jsp-property-group>
11.         <description>header and footer settings</description>
12.         <url-pattern>/index.jsp</url-pattern>
13.         <url-pattern>/WEB-INF/view/*</url-pattern>
14.     </jsp-property-group>
</jsp-config>
```

   Note: You may need to add carriage returns to the code so that it displays on multiple lines. You can right-click in the editor and choose Format (Alt-Shift-F; Ctrl-Shift-F on Mac) to have the code properly indented.

15. Switch to the Pages tab again, and in the Include Preludes and Include Codas fields, enter the paths to the `header.jspf` and `footer.jspf` files, respectively. You can

click the Browse button and navigate to the files in the provided dialog.



16. Switch back to the XML tab. Note that the following code has been added. (Changes in **bold**.)

```
17.  <jsp-config>
18.      <jsp-property-group>
19.          <description>header and footer settings</description>
20.          <url-pattern>/index.jsp</url-pattern>
21.          <url-pattern>/WEB-INF/view/*</url-pattern>
22.          <include-prelude>/WEB-INF/jspf/header.jspf</include-
    prelude>
23.          <include-coda>/WEB-INF/jspf/footer.jspf</include-coda>
24.      </jsp-property-group>
    </jsp-config>
```

    The above directive specifies that for all files found within the given `url-pattern`s, the `header.jspf` file will be prepended, and the `footer.jspf` file appended.

    To view the definitions of the above tags, as well as all tags available to you in the web deployment descriptor, consult the Servlet Specification.

25. Run the application again (press F6; fn-F6 on Mac). You've already removed the header and footer code from the `index.jsp` file, so you can determine whether it is automatically being added when the file is requested.

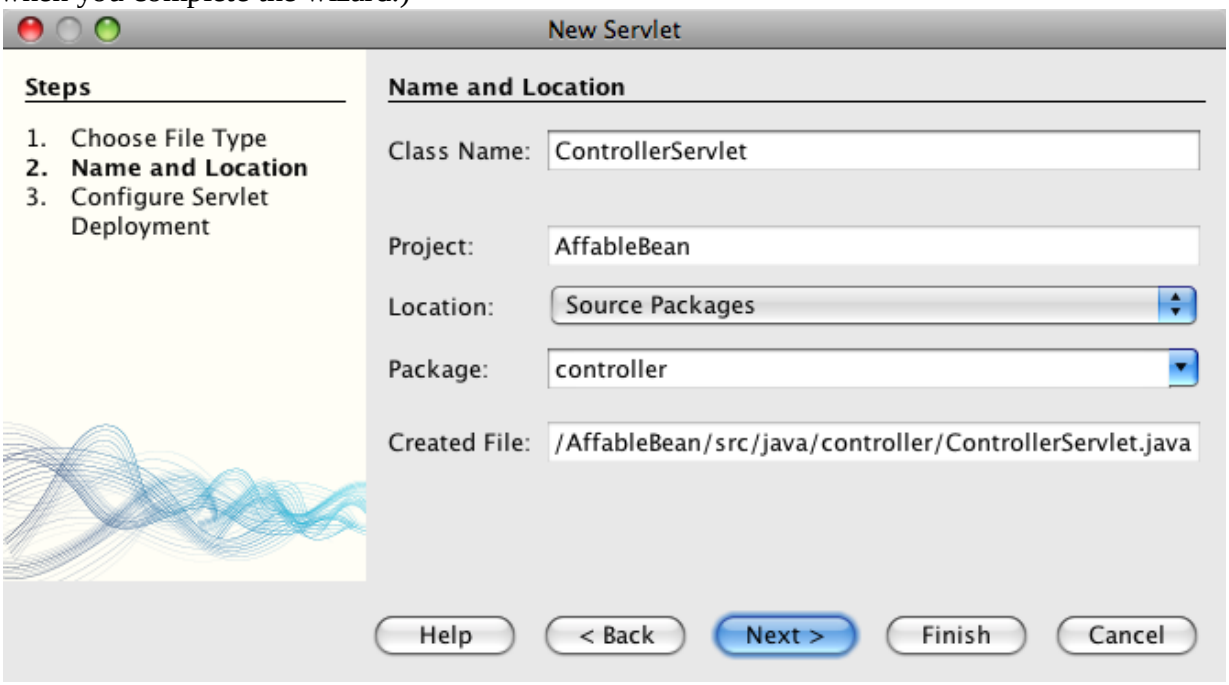
    You will see that the welcome page displays as it did previously, with header and footer content included.

Creating the Controller Servlet

The controller servlet handles incoming requests by initiating any actions needed to generate the model for the request, then forwarding the request to the appropriate view. For a visual representation, refer back to the MVC diagram for the AffableBean project.

The IDE provides a Servlet wizard that enables you to define the servlet component in a web application either by including the `@WebServlet` annotation in the generated class, or by adding the necessary directives to the deployment descriptor. In the following steps, you create the `ControllerServlet` and define it in the application context using the `@WebServlet` annotation.

1. In the Projects window, right-click the `AffableBean` project node and choose New > Servlet.
2. In the wizard, type `ControllerServlet` in the Class Name field.
3. In the Package field, type `controller`. (The new package is automatically created when you complete the wizard.)



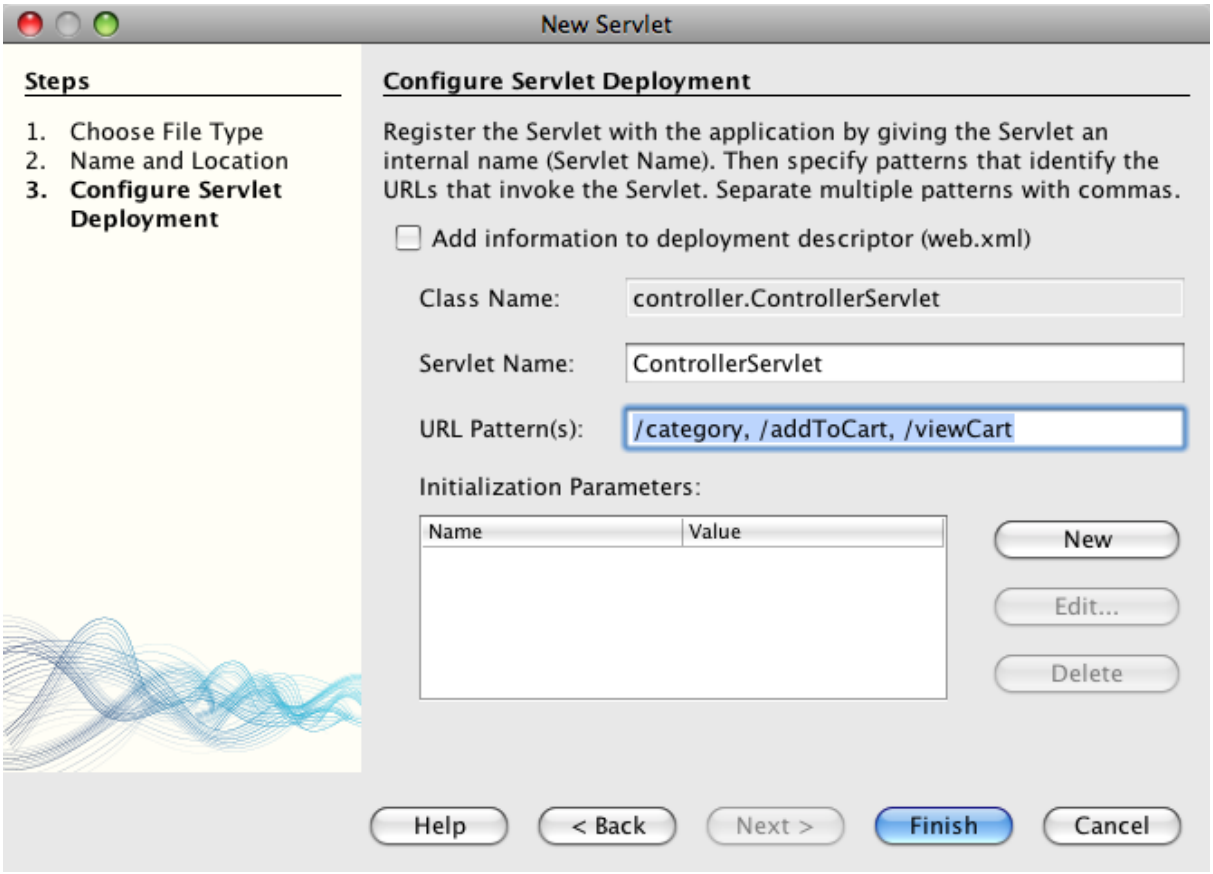
4. Click Next. Step 3 of the wizard lets you configure the servlet. Of primary importance are the URL patterns that you need to specify. The patterns identify the URLs that invoke the servlet. For example, if you enter '`/category`', you are directing the servlet to handle a request that appears as follows.

   `http://localhost/AffableBean/category`

   The URL patterns should correspond to the views and actions that a user can initiate. Looking at the welcome page mockup, a user should be able to select a category. We can therefore associate the `/category` URL with the action of clicking on a category image. Likewise, in the category page, users should be able to add an item to the shopping cart. We can therefore specify `/addToCart`.

5. In the URL Pattern(s) field, type in '`/category, /addToCart, /viewCart`'. Patterns are separated by commas. You can add more patterns directly in the servlet class once

it's created.



6. Click Finish. The IDE generates the `ControllerServlet` and opens it in the editor. The servlet and URL patterns are included in the `@WebServlet` annotation that appears above the class signature.

```
7. @WebServlet(name="ControllerServlet", urlPatterns={"/category",
   "/addToCart", "/viewCart"})
   public class ControllerServlet extends HttpServlet {
```

In the previous step, if you had chosen the '`Add information to deployment descriptor (web.xml)`' option in the wizard, the following markup would have been generated in the application's `web.xml` file instead.

```
<servlet>
    <servlet-name>ControllerServlet</servlet-name>
    <servlet-class>controller.ControllerServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ControllerServlet</servlet-name>
    <url-pattern>/category</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ControllerServlet</servlet-name>
    <url-pattern>/addToCart</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ControllerServlet</servlet-name>
    <url-pattern>/viewCart</url-pattern>
</servlet-mapping>
```

8. Add other URL patterns directly to the `@WebServlet` annotation's `urlPatterns` element. The application requires more URL patterns for other actions and views. You can type in the following patterns:
   - `/updateCart`
   - `/checkout`
   - `/purchase`
   - `/chooseLanguage`

   Be sure to separate each pattern with a comma. You can also reformat the annotation as follows:

```
@WebServlet(name="ControllerServlet",
            urlPatterns = {"/category",
                           "/addToCart",
                           "/viewCart",
                           "/updateCart",
                           "/checkout",
                           "/purchase",
                           "/chooseLanguage"})
```

9. Finally, include the `loadOnStartup` element so that the servlet is instantiated and initialized when the application is deployed. A value of `0` or greater will cause this to happen (`-1` is the default).

```
@WebServlet(name="ControllerServlet",
            loadOnStartup = 1,
            urlPatterns = {"/category",
                           "/addToCart",
                           "/viewCart",
                           "/updateCart",
                           "/checkout",
                           "/purchase",
                          "/chooseLanguage"})
```

Implementing the Controller Servlet

As previously stated, the controller servlet handles incoming requests by initiating any actions needed to generate the model for the request, then forwarding the request to the appropriate view. For a visual representation, refer back to the MVC diagram for the AffableBean project.

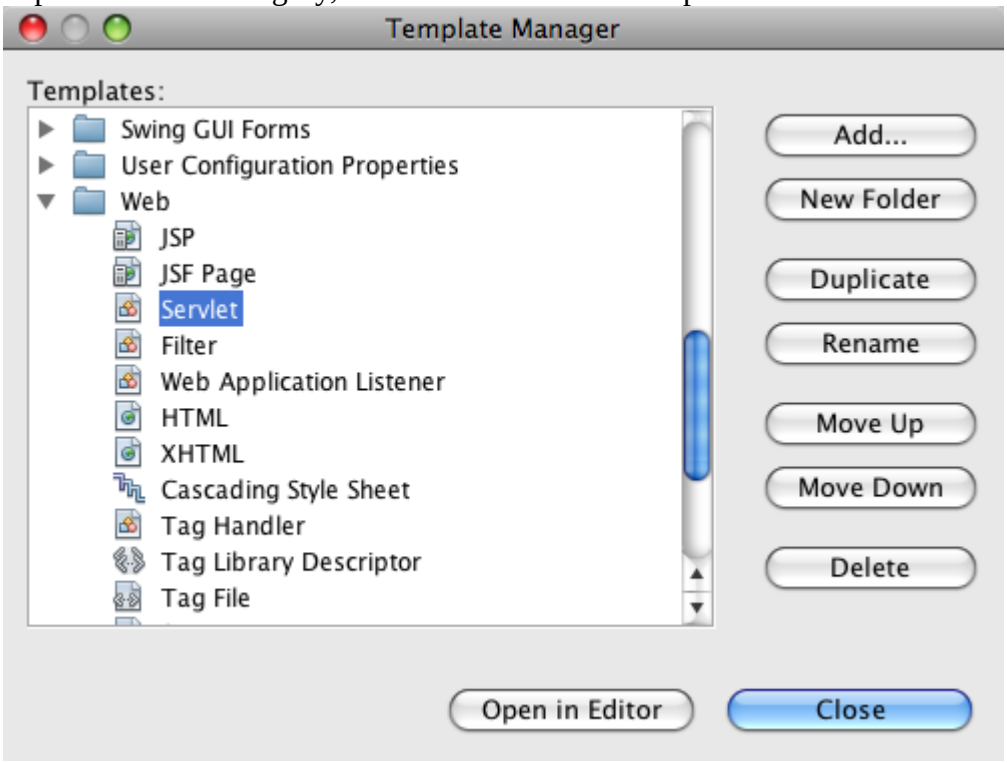
Looking at the generated code for the new `ControllerServlet`, you can see that the IDE's servlet template employs a `processRequest()` method which is called by both `doGet()` and `doPost()` methods. (You may need to expand the code fold by clicking the plus icon (⊞) in the editor's left margin to view these methods.) Because this application differentiates between `doGet()` and `doPost()`, you'll add code directly to these methods and remove the `processRequest()` method altogether.

Modifying File Templates with the IDE's Template Manager

The IDE provides you with a basic template for any new file you create. If the template is not optimal for your work patterns, you can alter it using the IDE's Template Manager. The IDE provides a template for virtually any file type.

For example, to modify the servlet template:

1. Open the Template Manager by choosing Tools > Templates from the main menu.
2. Expand the Web category, then select the Servlet template.



3. Click the Open in Editor button.
4. Modify the template in the editor. The next time you create a new servlet (e.g., using the Servlet wizard), the new version will be applied.

Now that you've mapped URL patterns to the servlet using the `@WebServlet` annotation, set up the `ControllerServlet` to handle these patterns. Also, instantiate a `RequestDispatcher` to forward the requested pattern to the appropriate view.

1. Replace the `ControllerServlet` class template code with the following code.

```
2. public class ControllerServlet extends HttpServlet {
3.
4.     /**
5.      * Handles the HTTP <code>GET</code> method.
6.      * @param request servlet request
7.      * @param response servlet response
8.      * @throws ServletException if a servlet-specific error occurs
9.      * @throws IOException if an I/O error occurs
10.       */
11.      @Override
12.      protected void doGet(HttpServletRequest request,
   HttpServletResponse response)
```

```
13.      throws ServletException, IOException {
14.
15.          String userPath = request.getServletPath();
16.
17.          // if category page is requested
18.          if (userPath.equals("/category")) {
19.              // TODO: Implement category request
20.
21.          // if cart page is requested
22.          } else if (userPath.equals("/viewCart")) {
23.              userPath = "/cart";
24.              // TODO: Implement cart page request
25.
26.          // if checkout page is requested
27.          } else if (userPath.equals("/checkout")) {
28.              // TODO: Implement checkout page request
29.
30.          // if user switches language
31.          } else if (userPath.equals("/chooseLanguage")) {
32.              // TODO: Implement language request
33.
34.          }
35.
36.          // use RequestDispatcher to forward request internally
37.          String url = "/WEB-INF/view" + userPath + ".jsp";
38.
39.          try {
40.              request.getRequestDispatcher(url).forward(request,
   response);
41.          } catch (Exception ex) {
42.              ex.printStackTrace();
43.          }
44.      }
45.
46.      /**
47.       * Handles the HTTP <code>POST</code> method.
48.       * @param request servlet request
49.       * @param response servlet response
50.       * @throws ServletException if a servlet-specific error occurs
51.       * @throws IOException if an I/O error occurs
52.       */
53.      @Override
54.      protected void doPost(HttpServletRequest request,
   HttpServletResponse response)
55.      throws ServletException, IOException {
56.
57.          String userPath = request.getServletPath();
58.
59.          // if addToCart action is called
60.          if (userPath.equals("/addToCart")) {
61.              // TODO: Implement add product to cart action
62.
63.          // if updateCart action is called
64.          } else if (userPath.equals("/updateCart")) {
65.              // TODO: Implement update cart action
66.
67.          // if purchase action is called
68.          } else if (userPath.equals("/purchase")) {
69.              // TODO: Implement purchase action
70.
71.          }
```

```
72.
73.          // use RequestDispatcher to forward request internally
74.          String url = "/WEB-INF/view" + userPath + ".jsp";
75.
76.          try {
77.              request.getRequestDispatcher(url).forward(request,
   response);
78.          } catch (Exception ex) {
79.              ex.printStackTrace();
80.          }
81.      }
82.
   }
```

83. Examine the code above. There are several points to note:

- The servlet uses a `userPath` instance variable to get the requested URL pattern from the client:

  ```
  String userPath = request.getServletPath();
  ```

  `userPath` is used by both `doGet()` and `doPost()` methods.

- URL patterns associated primarily with page requests are managed by the `doGet()` method. For example, `/category`, `/viewCart`, and `/checkout` result in the display of the category, cart, and checkout pages.)
- URL patterns associated with form submits and the transport of sensitive user data (e.g., `/addToCart`, `/updateCart`, and `/purchase`) are managed by the `doPost()` method.
- For both `doGet()` and `doPost()` methods, the path to the appropriate view is formed using a `url` string:

  ```
  String url = "/WEB-INF/view" + userPath + ".jsp";
  ```

- The `RequestDispatcher` is obtained from the `HttpServletRequest` and applies the `url` to forward the request:

  ```
  request.getRequestDispatcher(url).forward(request, response);
  ```

- `TODO` notes have been used to denote work that still needs to be done. For example:
- `// if category page is requested`
- `if (userPath.equals("/category")) {`
  `// TODO: Implement category request`

  Applying `TODO` notes in your code is a useful way to keep track of tasks that you need to complete. You can use the IDE's Tasks window (Ctrl-6; ⌘-6 on Mac) to view all TODO

notes, as well as any syntax or compile errors contained in your project.

You can control the keywords that display in the Tasks window. Open the Options window (Tools > Options; NetBeans > Preferences on Mac), then choose Miscellaneous > Tasks.

84. Run the project (press F6; fn-F6 on Mac) and test to see whether the `ControllerServlet` is forwarding requests to the appropriate views.

- Type in `http://localhost:8080/AffableBean/category` in the browser's address bar. The application's category page displays.
- Type in `http://localhost:8080/AffableBean/viewCart` in the browser's address bar. The application's cart page displays.
- Type in `http://localhost:8080/AffableBean/checkout` in the browser's address bar. The application's checkout page displays.

At this stage, you've created JSP pages that contain placeholders for functional components. You've also set up the front-end structure of the application. JSP pages now reside within the `WEB-INF` folder, header and footer code has been factored out into separate files, your deployment descriptor is properly configured, and you've set up the `ControllerServlet` to handle incoming requests. In the next tutorial unit, you take measures to enable connectivity between the application and the database.

If you'd like to compare your work with the sample solution for this unit, you can download snapshot 2 of the AffableBean project.

The NetBeans E-commerce Tutorial - Connecting the Application to the Database

Tutorial Contents

- Setting Context Parameters
- Working with JSTL
- Troubleshooting
- See Also

7. Adding Entity Classes and Session Beans
8. Managing Sessions
9. Integrating Transactional Business Logic
10. Adding Language Support (Coming Soon)
11. Securing the Application (Coming Soon)
12. Load Testing the Application (Coming Soon)
13. Conclusion

This tutorial unit focuses on communication between the database and the application. You begin by adding sample data to the database and explore some of the features provided by the IDE's SQL editor. You set up a data source and connection pool on the GlassFish server, and proceed by creating a simple JSP page that tests the data source by performing a simple query on the database.

This unit also addresses how the application retrieves and displays images necessary for web presentation, and how to set context parameters and retrieve their values from web pages. Once you are certain the data source is working correctly, you apply JSTL's `core` and `sql` tag libraries to retrieve and display category and product images for the index and category pages.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
MySQL database server	version 5.x
GlassFish server	v3 or Open Source Edition 3.0.1
AffableBean project	snapshot 2
website images	n/a

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.
- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could download the GlassFish server independently, but the version provided with the NetBeans download has the added benefit of being automatically registered with the IDE.
- You can follow this tutorial unit without having completed previous units. To do so, perform the following three steps:

1. **Set up your MySQL database server.** Follow the steps outlined in: Communicating with the Database Server.
2. **Create the `affablebean` schema on the database server.**

a. Click on affablebean-schema-creation.sql and copy (Ctrl-C; ⌘-C on Mac) the entire contents of the file.
b. Open the IDE's SQL editor. In the Services window (Ctrl-5; ⌘-5 on Mac), right-click the `affablebean` database connection ( ) node and choose Execute Command. The IDE's SQL editor opens.
c. Paste (Ctrl-V; ⌘-V on Mac) the entire contents of the `affablebean.sql` file into the editor.
d. Click the Run SQL ( ) button in the editor's toolbar. The script runs on your MySQL server. Tables are generated for the `affablebean` database.

3. Open the project snapshot in the IDE. In the IDE, press Ctrl-Shift-O (⌘-Shift-O on Mac) and navigate to the location on your computer where you unzipped the downloaded file.

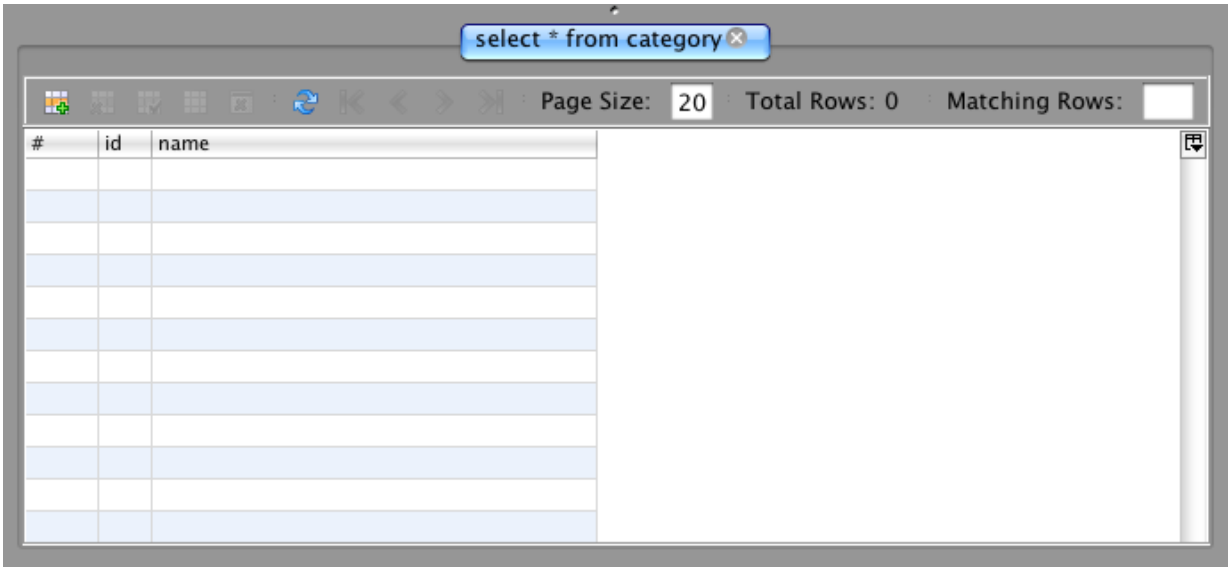
Adding Sample Data to the Database

Begin by adding sample data to the `category` and `product` tables. You can do this using the IDE's SQL editor, which allows you to interact directly with the database using native SQL. The IDE's SQL support also includes a GUI editor that enables you to add, remove, modify and delete table records.

- category table
- product table

category table

1. In the Services window (Ctrl-5; ⌘-5 on Mac), right-click the `category` table ( ) node and choose View Data. The SQL editor opens and displays with a GUI representation of the `category` table in the lower region. Note that the table is empty, as no data has yet been added.



Also, note that the native SQL query used to generate the GUI representation is displayed in the upper region of the editor: '`select * from category`'.

2. Delete '`select * from category`' and enter the following SQL statement:

```
INSERT INTO `category` (`name`) VALUES
('dairy'),('meats'),('bakery'),('fruit & veg');
```

This statement inserts four new records, each with a unique entry for the '`name`' column. Because the `id` column was specified as `AUTO_INCREMENT` when you created the schema, you do not need to worry about supplying a value.

3. Click the Run SQL ( ) button in the editor's toolbar. The SQL statement is executed.
4. To confirm that the data has been added, run the '`select * from category`' query again. To do so, you can use the SQL History window. Click the SQL History ( ) button in the editor's toolbar and double-click the '`select * from category`' entry. The SQL History window lists all SQL statements that you recently executed in the IDE.

Watch the screencast below to see how you can follow the above steps. When typing in the editor, be sure to take advantage of the IDE's code completion and suggestion facilities.

product table

1. Right-click the `product` table ( ) node and choose Execute Command. Choosing the Execute Command menu option in the Services window opens the SQL editor in the IDE.
2. Copy and paste the following `INSERT` statements into the editor.

```
3. --
4. -- Sample data for table `product`
```

```
5. --
6.
7. INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('milk', 1.70, 'semi skimmed (1L)', 1);
8. INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('cheese', 2.39, 'mild cheddar (330g)', 1);
9. INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('butter', 1.09, 'unsalted (250g)', 1);
10.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('free range eggs', 1.76, 'medium-sized (6 eggs)', 1);
11.
12.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('organic meat patties', 2.29, 'rolled in fresh herbs<br>2
   patties (250g)', 2);
13.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('parma ham', 3.49, 'matured, organic (70g)', 2);
14.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('chicken leg', 2.59, 'free range (250g)', 2);
15.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('sausages', 3.55, 'reduced fat, pork<br>3 sausages (350g)',
   2);
16.
17.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('sunflower seed loaf', 1.89, '600g', 3);
18.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('sesame seed bagel', 1.19, '4 bagels', 3);
19.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('pumpkin seed bun', 1.15, '4 buns', 3);
20.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('chocolate cookies', 2.39, 'contain peanuts<br>(3 cookies)',
   3);
21.
22.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('corn on the cob', 1.59, '2 pieces', 4);
23.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('red currants', 2.49, '150g', 4);
24.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('broccoli', 1.29, '500g', 4);
25.  INSERT INTO `product` (`name`, price, description, category_id)
   VALUES ('seedless watermelon', 1.49, '250g', 4);
```

Examine the above code and note the following points:

- By examining the `affablebean` schema generation script, you'll note that the `product` table contains a non-nullable, automatically incremental primary key. Whenever you insert a new record into the table (and don't explicitly set the value of the primary key), the SQL engine sets it for you. Also, note that the `product` table's `last_update` column applies `CURRENT_TIMESTAMP` as its default value. The SQL engine will therefore provide the current date and time for this field when a record is created.

  Looking at this another way, if you were to create an `INSERT` statement that didn't indicate which columns would be affected by the insertion action, you would need to account for all columns. In this case, you could enter a `NULL` value to enable the SQL engine to automatically handle fields that have default values specified. For example, the following statement elicits the same result as the first line of the above code:

```
INSERT INTO `product` VALUES (NULL, 'milk', 1.70, 'semi skimmed (1L)',
NULL, 1);
```

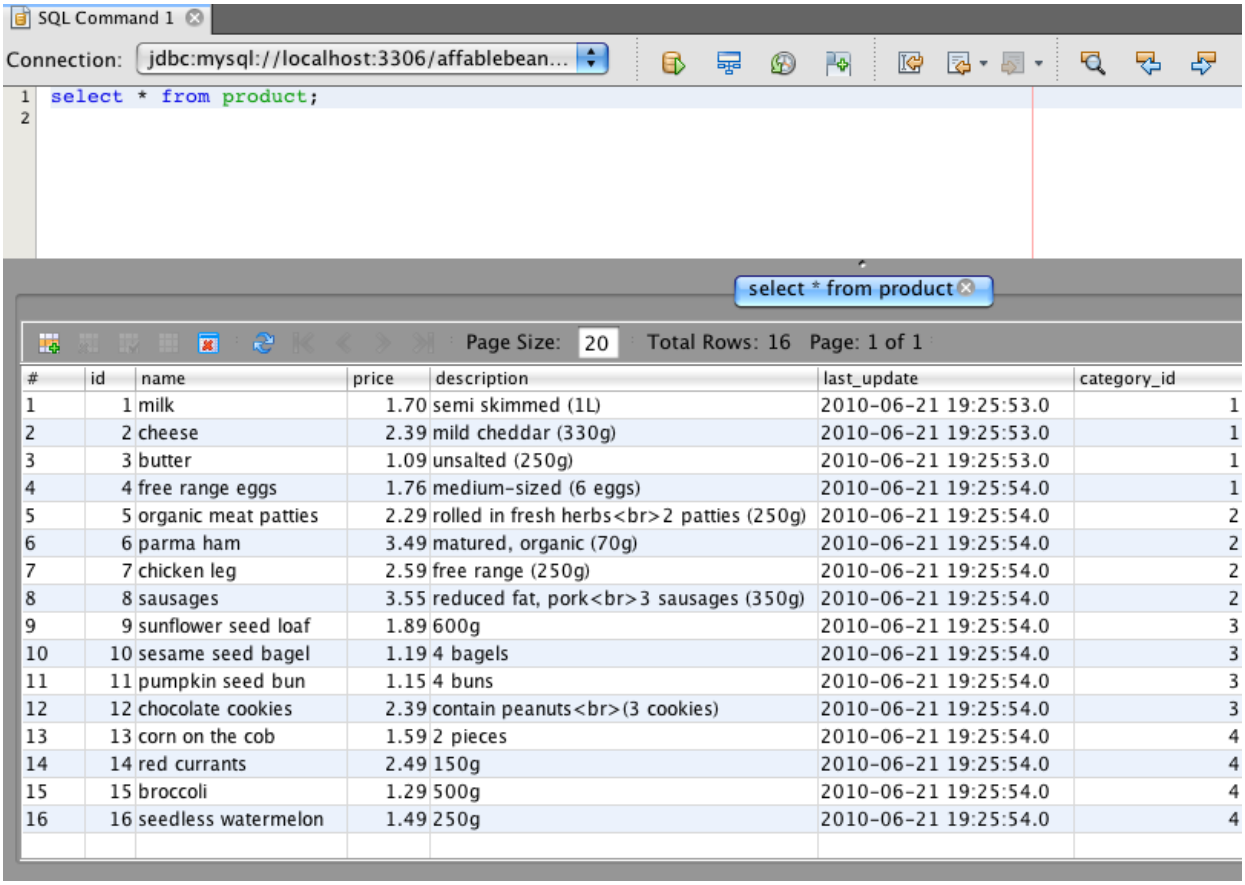
After running the statement, you'll see that the record contains an automatically incremented primary key, and the `last_update` column lists the current date and time.

- The value for the final column, '`category_id`', must correspond to a value contained in the `category` table's `id` column. Because you have already added four records to the `category` table, the `product` records you are inserting reference one of these four records. If you try to insert a `product` record that references a `category_id` that doesn't exist, a foreign key constraint fails.

26. Click the Run SQL ( ) button in the editor's toolbar.

    Note: View the Output window (Ctrl-4; ⌘-4 on Mac) to see a log file containing results of the execution.

27. Right-click the `product` table ( ) node and choose View Data. You can see 16 new records listed in the table.



#	id	name	price	description	last_update	category_id
1	1	milk	1.70	semi skimmed (1L)	2010-06-21 19:25:53.0	1
2	2	cheese	2.39	mild cheddar (330g)	2010-06-21 19:25:53.0	1
3	3	butter	1.09	unsalted (250g)	2010-06-21 19:25:53.0	1
4	4	free range eggs	1.76	medium-sized (6 eggs)	2010-06-21 19:25:54.0	1
5	5	organic meat patties	2.29	rolled in fresh herbs 2 patties (250g)	2010-06-21 19:25:54.0	2
6	6	parma ham	3.49	matured, organic (70g)	2010-06-21 19:25:54.0	2
7	7	chicken leg	2.59	free range (250g)	2010-06-21 19:25:54.0	2
8	8	sausages	3.55	reduced fat, pork 3 sausages (350g)	2010-06-21 19:25:54.0	2
9	9	sunflower seed loaf	1.89	600g	2010-06-21 19:25:54.0	3
10	10	sesame seed bagel	1.19	4 bagels	2010-06-21 19:25:54.0	3
11	11	pumpkin seed bun	1.15	4 buns	2010-06-21 19:25:54.0	3
12	12	chocolate cookies	2.39	contain peanuts (3 cookies)	2010-06-21 19:25:54.0	3
13	13	corn on the cob	1.59	2 pieces	2010-06-21 19:25:54.0	4
14	14	red currants	2.49	150g	2010-06-21 19:25:54.0	4
15	15	broccoli	1.29	500g	2010-06-21 19:25:54.0	4
16	16	seedless watermelon	1.49	250g	2010-06-21 19:25:54.0	4

NetBeans GUI Support for Database Tables

In the Services window, when you right-click a table ( ) node and choose View Data, the IDE displays a visual representation of the table and the data it contains (as depicted in the image above). You can also use this GUI support to add, modify, and delete table data.

- **Add new records:** To add new records, click the Insert Record ( ) button. An Insert Records dialog window displays, enabling you to enter new records. When you click OK, the new data is committed to the database, and the GUI representation of the table is automatically updated.

  Click the Show SQL button within the dialog window to view the SQL statement(s) that will be applied upon initiating the action.

- **Modify records:** You can make edits to existing records by double-clicking directly in table cells and modifying field entries. Modified entries display as green text. When you are finished editing data, click the Commit Record ( ) button to commit changes to the actual database. (Similarly, click the Cancel Edits ( ) button to cancel any edits you have made.
- **Delete individual records:** Click a row in the table, then click the Delete Selected Record ( ) button. You can also delete multiple rows simultaneously by holding Ctrl (⌘ on Mac) while clicking to select rows.
- **Delete all records:** Deleting all records within a table is referred to as '*truncating*' the table. Click the Truncate Table ( ) button to delete all records contained in the displayed table.

If the displayed data needs to be resynchronized with the actual database, you can click the Refresh Records ( ) button. Note that much of the above-described functionality can also be accessed from the right-click menu within the GUI editor.

Creating a Data Source

A data source (a.k.a. a JDBC resource) provides applications with the means of connecting to a database. Applications get a database connection from a connection pool by looking up a data source using the Java Naming and Directory Interface (JNDI) and then requesting a connection. The connection pool associated with the data source provides the connection for the application.

A connection pool contains a group of reusable connections for a particular database. Because creating each new physical connection is time-consuming, the server maintains a pool of available connections to increase performance. When an application requests a connection, it obtains one from the pool. When an application closes a connection, the connection is returned to the pool. Connection pools use a JDBC driver to create physical database connections.

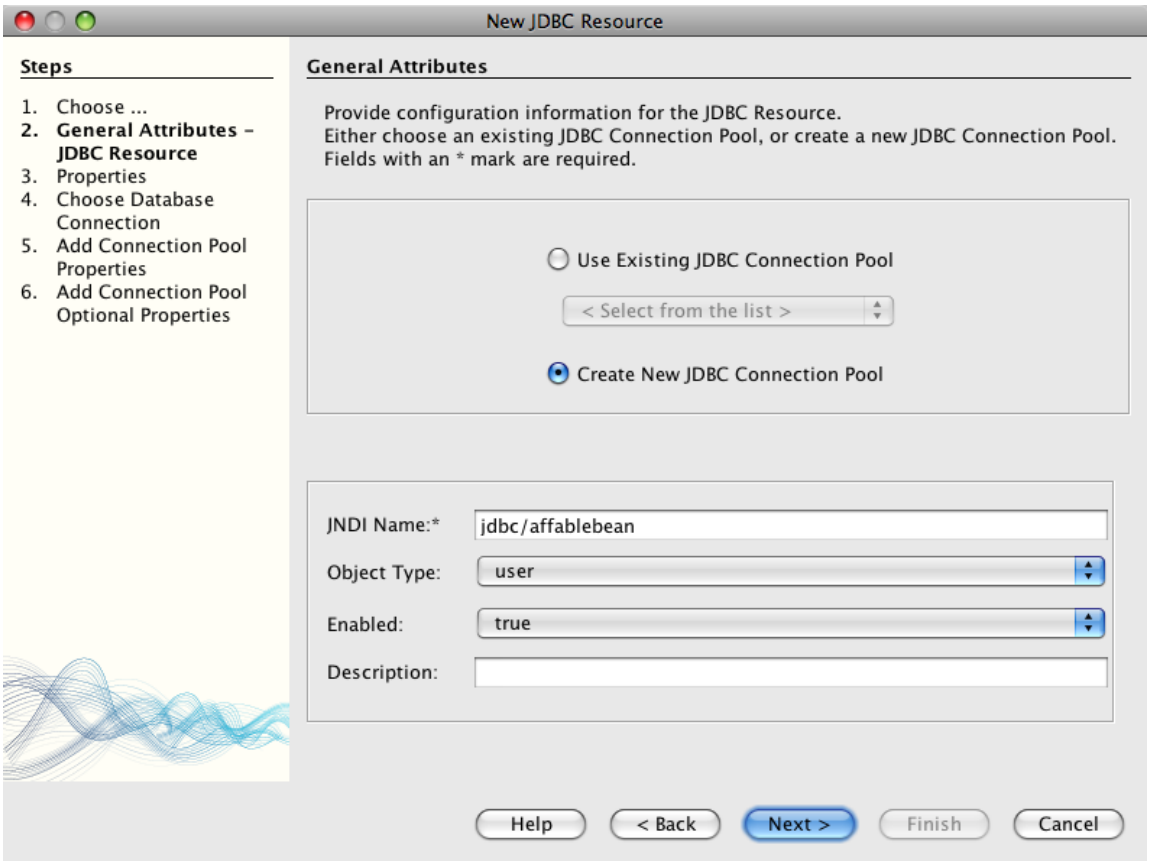
In order to enable the application access to the `affablebean` database, you need to create a connection pool and a data source that uses the connection pool. Use the NetBeans GlassFish JDBC Resource wizard to accomplish this.

Note: You can also create connection pools and data sources directly on the GlassFish server using the GlassFish Administration Console. However, creating these resources in this manner requires that you manually enter database connection details (i.e., username, password and URL). The benefit of using the NetBeans wizard is that it extracts any connection details directly from an existing database connection, thus eliminating potential connectivity problems.

To access the console from the IDE, in the Services window right-click the Servers > GlassFish node and choose View Admin Console. The default username/password is: `admin/adminadmin`. If you'd like to set up the connection pool and data source using the GlassFish Administration console, follow steps 3-15 of the NetBeans E-commerce Tutorial Setup Instructions. The setup instructions are provided for later tutorial units.

1. Click the New File ( ) button in the IDE's toolbar. (Alternatively, press Ctrl-N; ⌘-N on Mac.)
2. Select the **GlassFish** category, then select **JDBC Resource** and click Next.
3. In Step 2 of the JDBC Resource wizard, select the `Create New JDBC Connection Pool` option. When you do so, three new steps are added to the wizard, enabling you to specify connection pool settings.
4. Enter details to set up the data source:

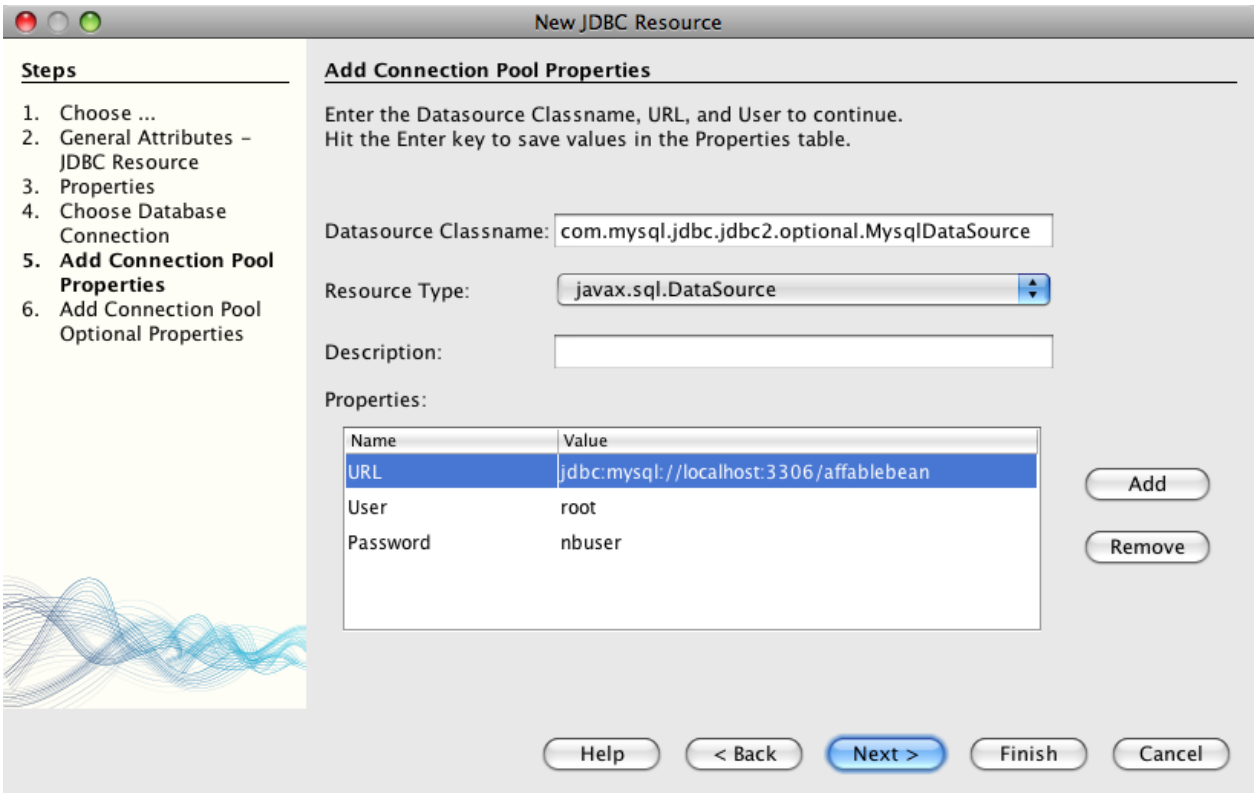
- **JNDI Name:** `jdbc/affablebean`
  By convention, the JNDI name begins with the '`jdbc/`' string.
- **Object Type:** `user`
- **Enabled:** `true`



5. Click Next. In Step 3, Additional Properties, you do not need to specify any additional configuration information for the data source.
6. Click Next. In Step 4, Choose Database Connection, type in `AffableBeanPool` as the JDBC connection pool name. Also, ensure that the `Extract from Existing Connection` option is selected, and that the `jdbc:mysql://localhost:3306/affablebean` connection is listed.
7. Click Next. In Step 5, Add Connection Pool Properties, specify the following details:

- **Datasource Classname:** `com.mysql.jdbc.jdbc2.optional.MysqlDataSource`
- **Resource Type:** `javax.sql.ConnectionPoolDataSource`
- **Description:** *(Optional)* `Connects to the affablebean database`

Also, note that the wizard extracts and displays properties from the existing connection.



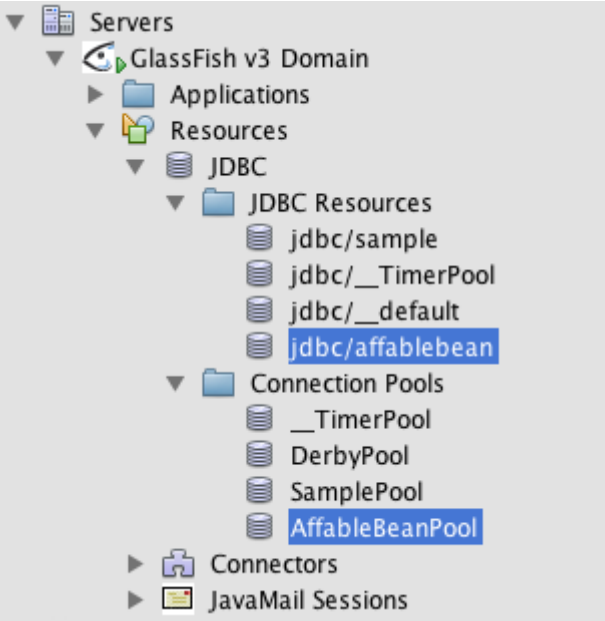
8. Click Finish. The wizard generates a `sun-resources.xml` file for the project that contains all information required to set up the connection pool and data source on GlassFish. The `sun-resources.xml` file is a deployment descriptor specific to the GlassFish application server. When the project next gets deployed, the server will read in any configuration data contained in `sun-resources.xml`, and set up the connection pool and data source accordingly. Note that once the connection pool and data source exist on the server, your project no longer requires the `sun-resources.xml` file.
9. In the Projects window (Ctrl-1; ⌘-1 on Mac), expand the Server Resources node and double-click the `sun-resources.xml` file to open it in the editor. Here you see the XML configuration required to set up the connection pool and data source. (Code below is formatted for readability.)

```
<resources>
  <jdbc-resource enabled="true"
                 jndi-name="jdbc/affablebean"
```

```
13.                   object-type="user"
14.                   pool-name="AffableBeanPool">
15.    </jdbc-resource>
16.
17.    <jdbc-connection-pool allow-non-component-callers="false"
18.                          associate-with-thread="false"
19.                          connection-creation-retry-attempts="0"
20.                          connection-creation-retry-interval-in-
   seconds="10"
21.                          connection-leak-reclaim="false"
22.                          connection-leak-timeout-in-seconds="0"
23.                          connection-validation-method="auto-commit"
24.                          datasource-
   classname="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
25.                          fail-all-connections="false"
26.                          idle-timeout-in-seconds="300"
27.                          is-connection-validation-required="false"
28.                          is-isolation-level-guaranteed="true"
29.                          lazy-connection-association="false"
30.                          lazy-connection-enlistment="false"
31.                          match-connections="false"
32.                          max-connection-usage-count="0"
33.                          max-pool-size="32"
34.                          max-wait-time-in-millis="60000"
35.                          name="AffableBeanPool"
36.                          non-transactional-connections="false"
37.                          pool-resize-quantity="2"
38.                          res-
   type="javax.sql.ConnectionPoolDataSource"
39.                          statement-timeout-in-seconds="-1"
40.                          steady-pool-size="8"
41.                          validate-atmost-once-period-in-seconds="0"
42.                          wrap-jdbc-objects="false">
43.
44.      <description>Connects to the affablebean database</description>
45.      <property name="URL"
   value="jdbc:mysql://localhost:3306/affablebean"/>
46.      <property name="User" value="root"/>
47.      <property name="Password" value="nbuser"/>
48.    </jdbc-connection-pool>
   </resources>
```

49. In the Projects window (Ctrl-1; ⌘-1 on Mac), right-click the `AffableBean` project node and choose Deploy. The GlassFish server reads configuration data from the `sun-resources.xml` file and creates the `AffableBeanPool` connection pool, and `jdbc/affablebean` data source.
50. In the Services window, expand the Servers > GlassFish > Resources > JDBC node. Here you can locate the `jdbc/affablebean` data source listed under JDBC

Resources, and the `AffableBeanPool` connection pool listed under Connection Pools.



Right-click data source and connection pool nodes to view and make changes to their properties. You can associate a data source with any connection pool registered on the server. You can edit property values for connection pools, and unregister both data sources and connection pools from the server.

Testing the Data Source

Start by making sure the GlassFish server can successfully connect to the MySQL database. You can do this by pinging the `AffableBeanPool` connection pool in the GlassFish Administration Console.

Then proceed by adding a reference in your project to the data source you created on the server. To do so, you create a `<resource-ref>` entry in the application's `web.xml` deployment descriptor.

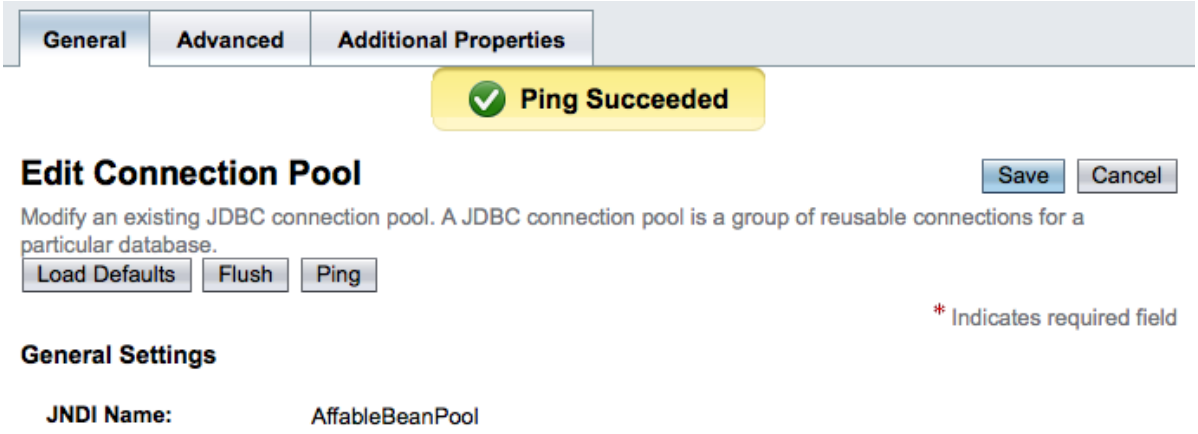
Finally, use the IDE's editor support for the JSTL `<sql>` tag library, and create a JSP page that queries the database and outputs data in a table on a web page.

Pinging the Connection Pool

1. Ensure that the GlassFish server is already running. In the Services window (Ctrl-5; ⌘-5 on Mac), expand the Servers node. Note the small green arrow next to the GlassFish icon ( ).

(If the server is not running, right-click the server node and choose Start.)

2. Right-click the server node and choose View Admin Console. The GlassFish Administration Console opens in a browser.
3. Log into the administration console. The default username/password is: `admin/adminadmin`.
4. In the console's tree on the left, expand the Resources > JDBC > Connection Pools nodes, then click `AffableBeanPool`. In the main window, the Edit Connection Pool interface displays for the selected connection pool.
5. Click the Ping button. If the ping succeeds, the GlassFish server has a working connection to the `affablebean` database on the MySQL server.

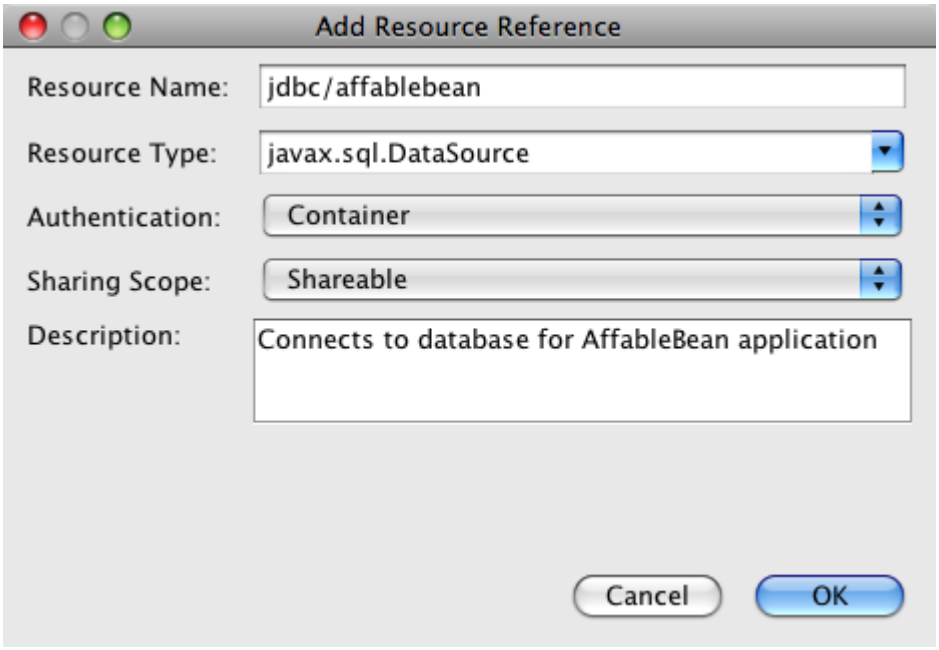


(If the ping fails, see suggestions in the Troubleshooting section below.)

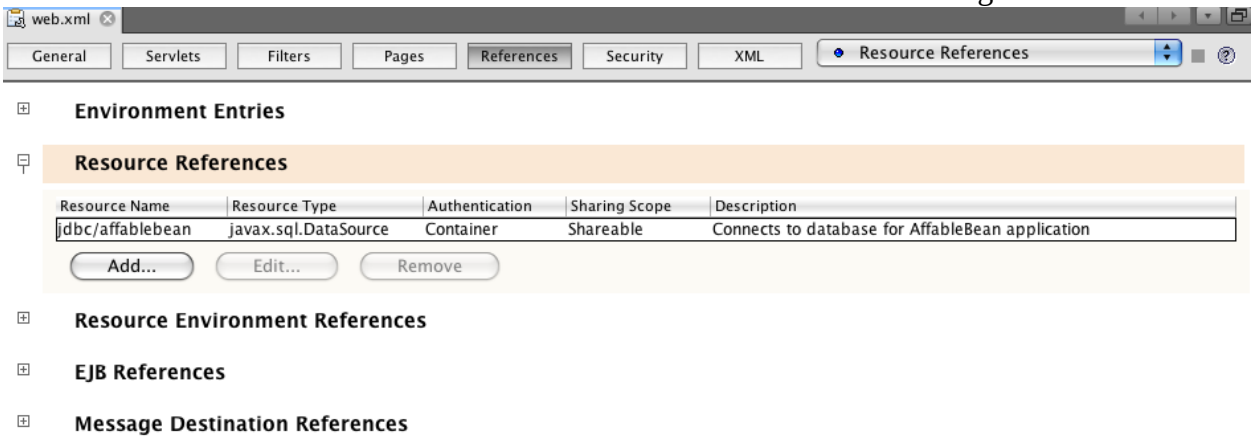
Creating a Resource Reference to the Data Source

1. In the Projects window, expand the Configuration Files folder and double-click `web.xml`. A graphical interface for the file displays in the IDE's main window.
2. Click the References tab located along the top of the editor. Expand the Resource References heading, then click Add. The Add Resource Reference dialog opens.
3. Enter the following details into the dialog:

- **Resource Name:** `jdbc/affablebean`
- **Resource Type:** `javax.sql.DataSource`
- **Authentication:** `Container`
- **Sharing Scope:** `Shareable`
- **Description:** *(Optional)* `Connects to database for AffableBean application`



4. Click OK. The new resource is added under the Resource References heading.



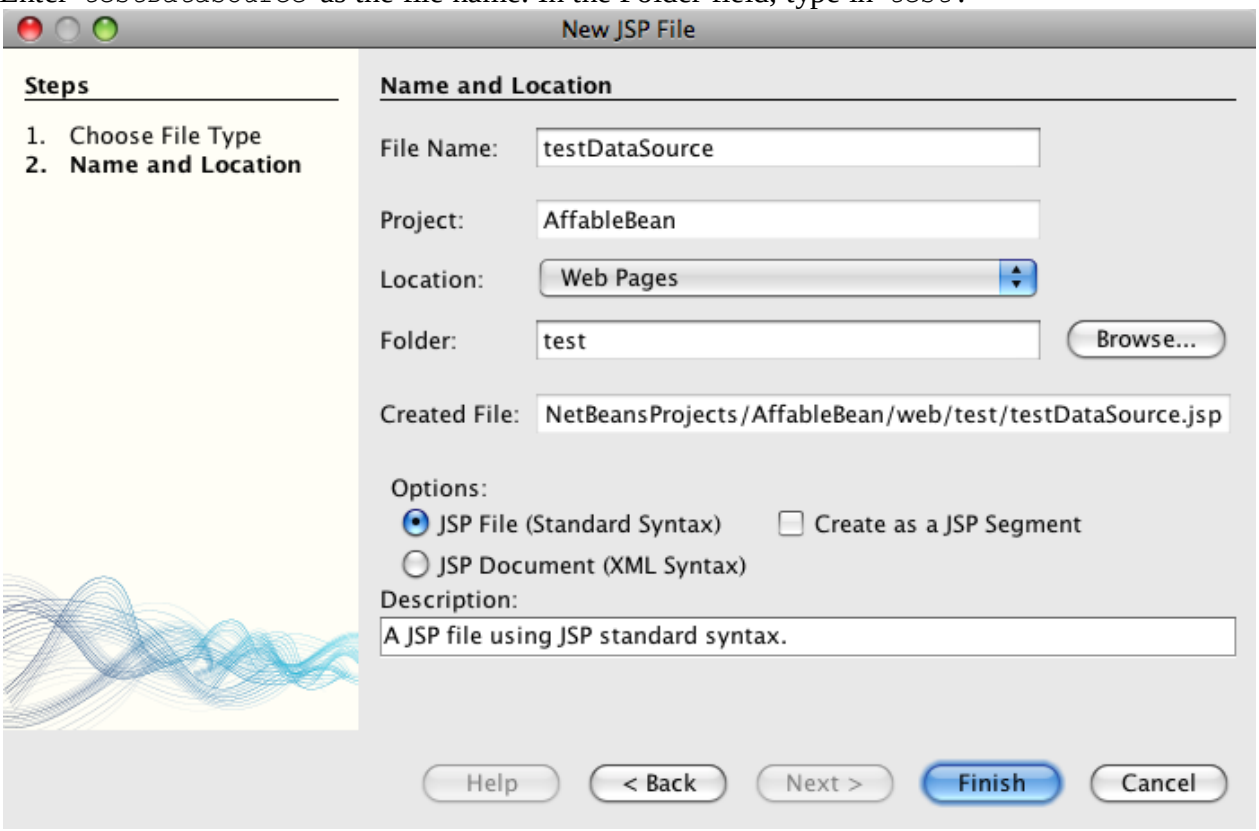
To verify that the resource is now added to the `web.xml` file, click the XML tab located along the top of the editor. Notice that the following `<resource-ref>` tags are now included:

```
<resource-ref>
    <description>Connects to database for AffableBean
application</description>
    <res-ref-name>jdbc/affablebean</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
      <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Querying the Database from a JSP Page

1. Create a new JSP page to test the data source. Click the New File ( ) button. (Alternatively, press Ctrl-N; ⌘-N on Mac.)
2. Select the Web category, then select the JSP file type and click Next.

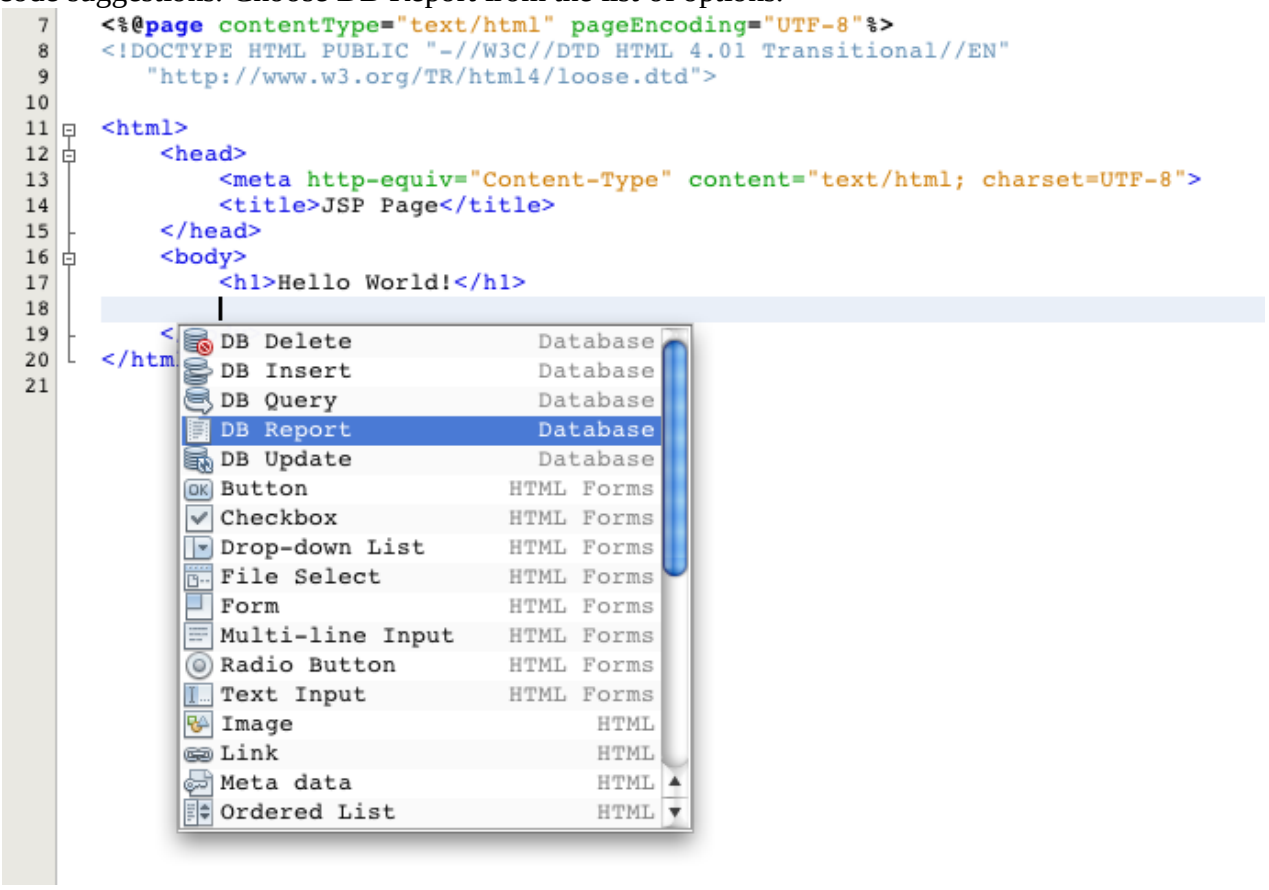
3. Enter '`testDataSource`' as the file name. In the Folder field, type in '`test`'.



The project does not yet have a folder named '`test`' within the Web Pages location (i.e., within the `web` folder). By entering '`test`' into the Folder field, you have the IDE create the folder upon completing the wizard.

4. Click finish. The IDE generates a new `testDataSource.jsp` file, and places it into the new `test` folder within the project.
5. In the new `testDataSource.jsp` file, in the editor, place your cursor at the end of the line containing the `<h1>` tags (line 17). Press Return, then press Ctrl-Space to invoke

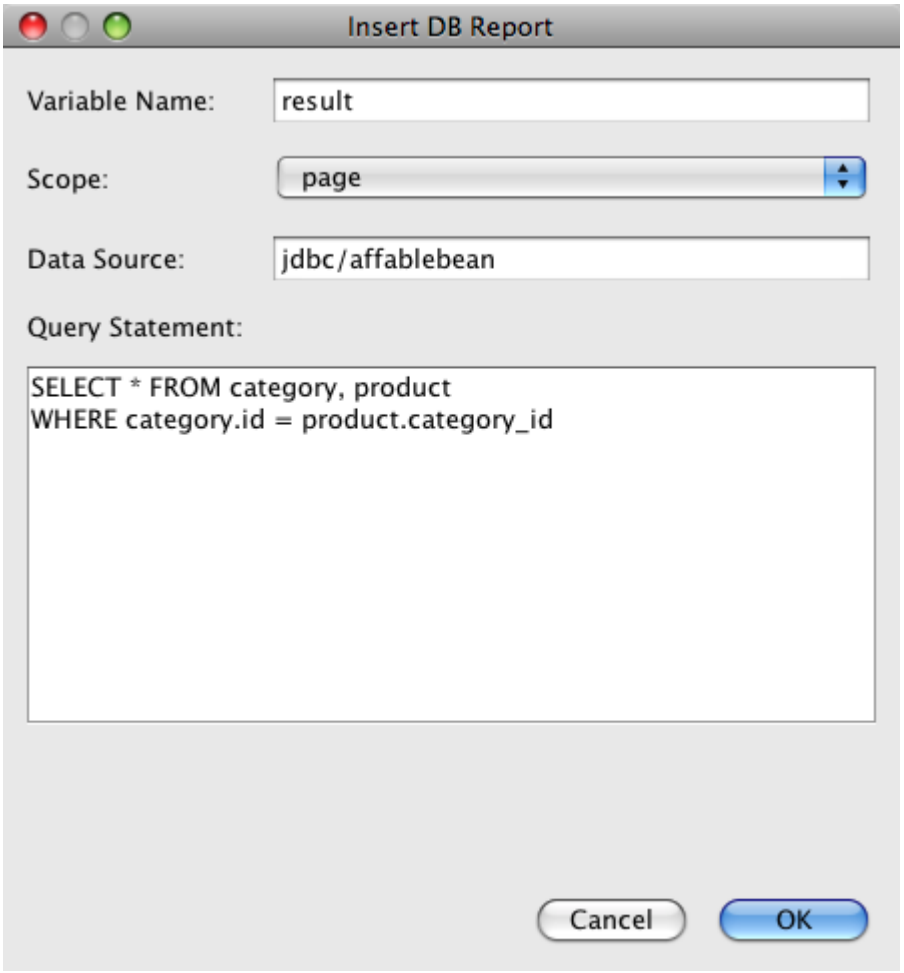
code suggestions. Choose DB Report from the list of options.



If line numbers do not display, right-click in the left margin of the editor and choose Show Line Numbers.

6. In the Insert DB Report dialog, specify the data source and modify the SQL query to be executed:

- **Data Source:** `jdbc/affablebean`
- **Query Statement:** `SELECT * FROM category, product WHERE category.id = product.category_id`



7. Click OK. The dialog adds the `taglib` directives for the JSTL `core` and `sql` libraries to the top of the file:

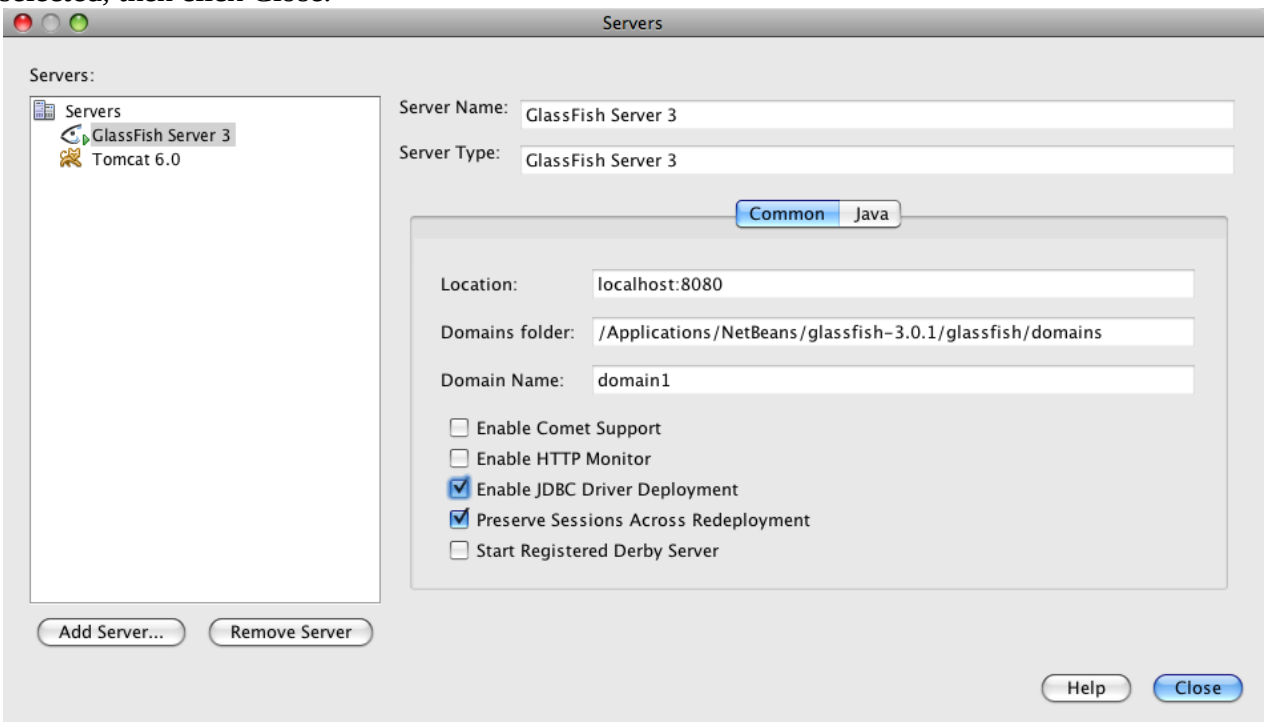
```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
```

The dialog also generates template code to display the query results in an HTML table:

```
<sql:query var="result" dataSource="jdbc/affablebean">
    SELECT * FROM category, product
    WHERE category.id = product.category_id
</sql:query>

<table border="1">
    <!-- column headers -->
    <tr>
        <c:forEach var="columnName" items="${result.columnNames}">
            <th><c:out value="${columnName}"/></th>
        </c:forEach>
    </tr>
    <!-- column data -->
    <c:forEach var="row" items="${result.rowsByIndex}">
        <tr>
            <c:forEach var="column" items="${row}">
                <td><c:out value="${column}"/></td>
            </c:forEach>
        </tr>
```

```
    </c:forEach>
</table>
```

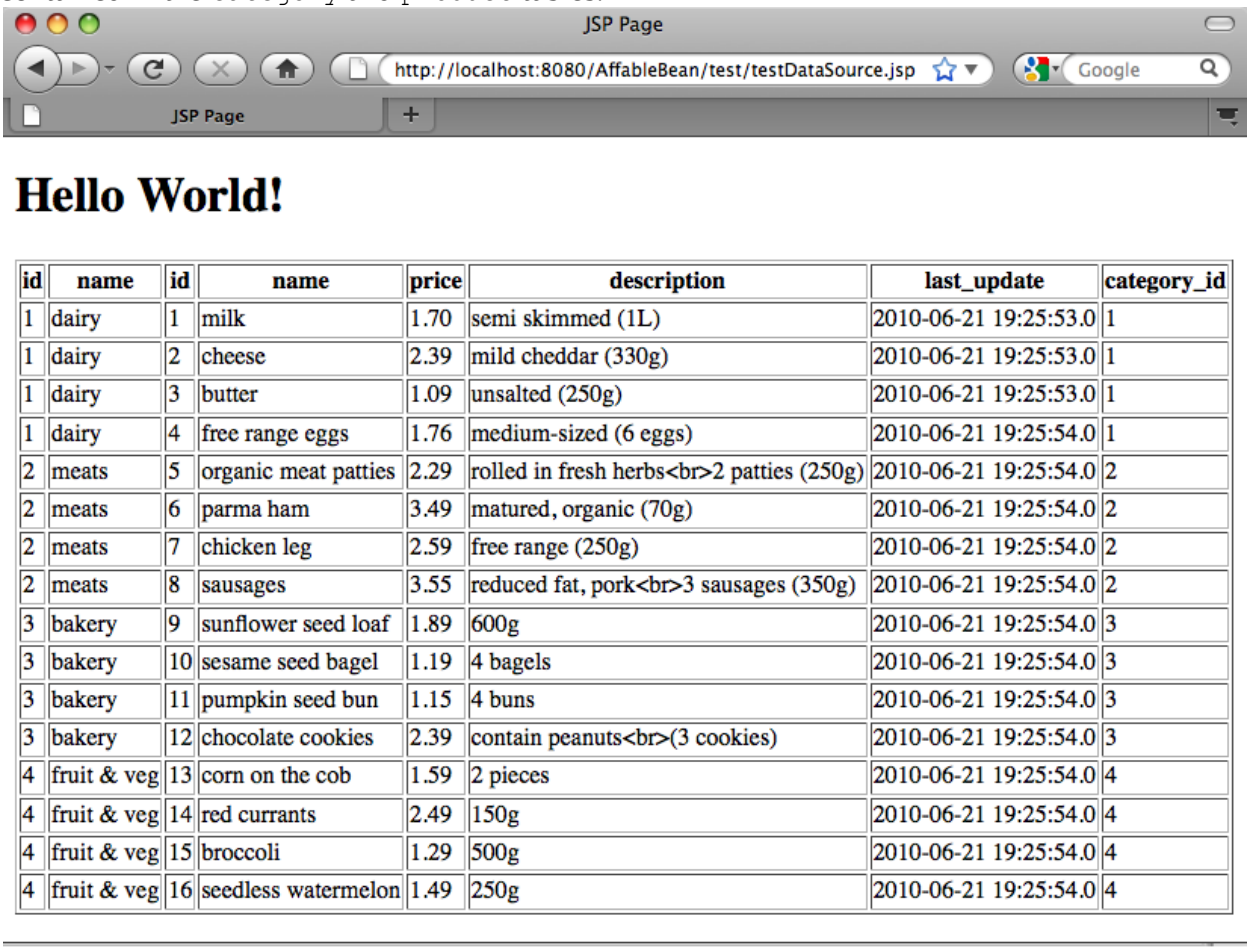
9. Before running the file in a browser, make sure you have enabled the JDBC driver deployment option in NetBeans' GlassFish support. Choose Tools > Servers to open the Servers window. In the left column, select the GlassFish server you are deploying to. In the main column, ensure that the 'Enable JDBC Driver Deployment' option is selected, then click Close.



   For Java applications that connect to a database, the server requires a JDBC driver to be able to create a communication bridge between the SQL and Java languages. In the case of MySQL, you use the Connector/J JDBC driver. Ordinarily you would need to manually place the driver JAR file into the server's `lib` directory. With the 'Enable JDBC Driver Deployment' option selected, the server performs a check to see whether a driver is needed, and if so, the IDE deploys the driver to the server.
10. Right-click in the editor and choose Run File (Shift-F6; fn-Shift-F6 on Mac). The `testDataSource.jsp` file is compiled into a servlet, deployed to the server, then runs in a browser.
11. Open the Output window (Ctrl-4; ⌘-4 on Mac) and click the 'AffableBean (run)' tab. The output indicates that the driver JAR file (`mysql-connector-java-5.1.6-`

`bin.jar`) is deployed.

12. Examine `testDataSource.jsp` in the browser. You see an HTML table listing data contained in the `category` and `product` tables.



id	name	id	name	price	description	last_update	category_id
1	dairy	1	milk	1.70	semi skimmed (1L)	2010-06-21 19:25:53.0	1
1	dairy	2	cheese	2.39	mild cheddar (330g)	2010-06-21 19:25:53.0	1
1	dairy	3	butter	1.09	unsalted (250g)	2010-06-21 19:25:53.0	1
1	dairy	4	free range eggs	1.76	medium-sized (6 eggs)	2010-06-21 19:25:54.0	1
2	meats	5	organic meat patties	2.29	rolled in fresh herbs 2 patties (250g)	2010-06-21 19:25:54.0	2
2	meats	6	parma ham	3.49	matured, organic (70g)	2010-06-21 19:25:54.0	2
2	meats	7	chicken leg	2.59	free range (250g)	2010-06-21 19:25:54.0	2
2	meats	8	sausages	3.55	reduced fat, pork 3 sausages (350g)	2010-06-21 19:25:54.0	2
3	bakery	9	sunflower seed loaf	1.89	600g	2010-06-21 19:25:54.0	3
3	bakery	10	sesame seed bagel	1.19	4 bagels	2010-06-21 19:25:54.0	3
3	bakery	11	pumpkin seed bun	1.15	4 buns	2010-06-21 19:25:54.0	3
3	bakery	12	chocolate cookies	2.39	contain peanuts (3 cookies)	2010-06-21 19:25:54.0	3
4	fruit & veg	13	corn on the cob	1.59	2 pieces	2010-06-21 19:25:54.0	4
4	fruit & veg	14	red currants	2.49	150g	2010-06-21 19:25:54.0	4
4	fruit & veg	15	broccoli	1.29	500g	2010-06-21 19:25:54.0	4
4	fruit & veg	16	seedless watermelon	1.49	250g	2010-06-21 19:25:54.0	4

(If you receive a server error, see suggestions in the Troubleshooting section below.)

At this stage, we have set up a working data source and connection pool on the server, and demonstrated that the application can access data contained in the `affablebean` database.

Setting Context Parameters

This section demonstrates how to configure context parameters for the application, and how to access parameter values from JSP pages. The owner of an application may want to be able to change certain settings without the need to make intrusive changes to source code. Context parameters enable you application-wide access to parameter values, and provide a convenient way to change parameter values from a single location, should the need arise.

Setting up context parameters can be accomplished in two steps:

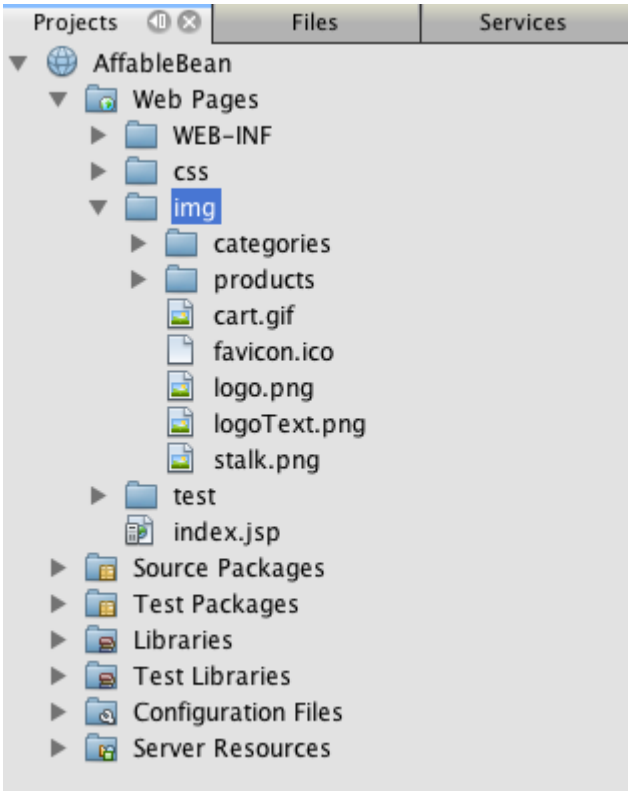
- Listing parameter names and values in the web deployment descriptor
- Calling the parameters in JSP pages using the `initParam` object

The JSP Expression Language (EL) defines *implicit objects*, which `initParam` is an example of. When working in JSP pages, you can utilize implicit objects using dot notation and placing expressions within EL delimiters (`${...}`). For example, if you have an initialization parameter named `myParam`, you can access it from a JSP page with the expression `${initParam.myParam}`.

For more information on the JSP Expression Language and implicit objects, see the Java EE 5 Tutorial: JavaServer Pages Technology > Unified Expression Language.

By way of demonstration, you create context parameters for the image paths to category and product images used in the `AffableBean` project. Begin by adding the provided image resources to the project, then perform the two steps outlined above.

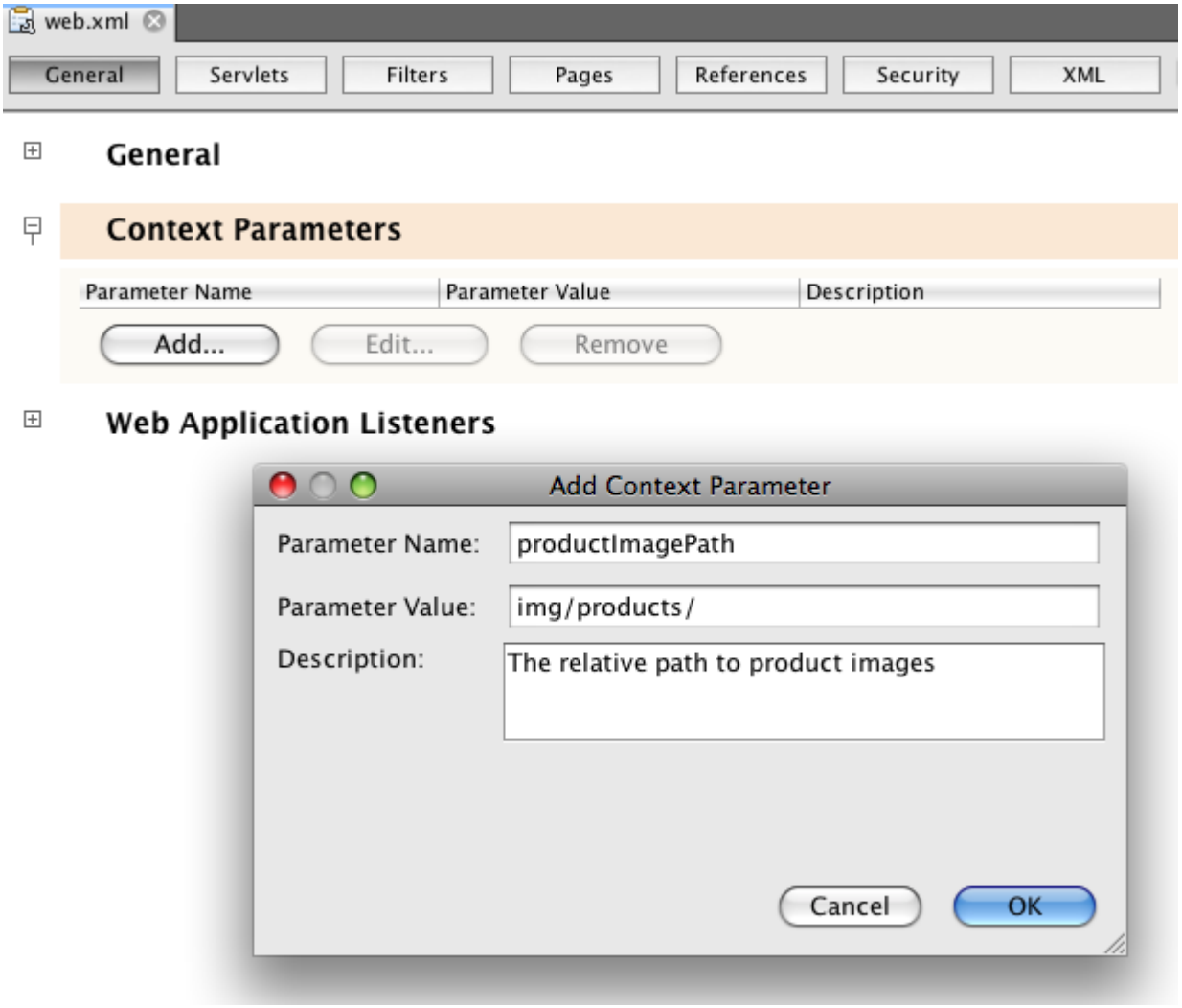
1. Download the website sample images, and unzip the file to a location on your computer. The unzipped file is an `img` folder that contains all of the image resources required for the `AffableBean` application.
2. Import the `img` folder into the `AffableBean` project. Copy (Ctrl-C; ⌘-C on Mac) the `img` folder, then in the IDE's Projects window, paste (Ctrl-V; ⌘-V on Mac) the folder into the project's Web Pages node.



The `categories` and `products` folders contain the images that will be displayed in the index and category pages, respectively.

3. Open the project's web deployment descriptor. In the Projects window, expand the Configuration Files node and double-click `web.xml`.
4. Click the General tab, then expand Context Parameters and click the Add button.
5. In the Add Context Parameter dialog, enter the following details:

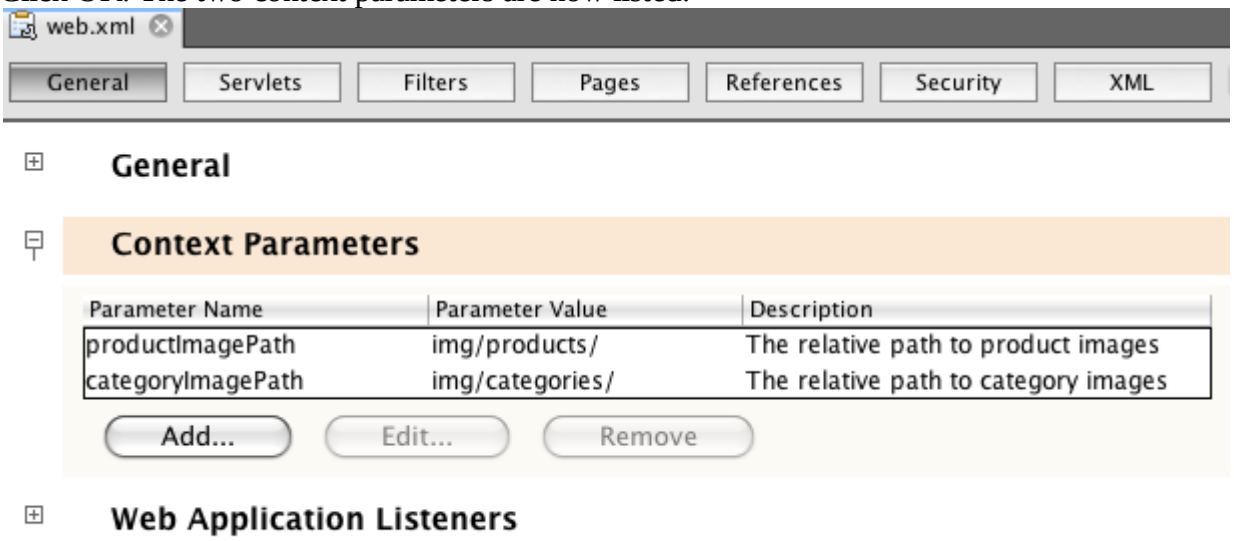
- **Parameter Name:** `productImagePath`
- **Parameter Value:** `img/products/`
- **Description:** *(Optional)* `The relative path to product images`



6. Click OK.
7. Click the Add button again and enter the following details:

- **Parameter Name:** `categoryImagePath`
- **Parameter Value:** `img/categories/`
- **Description:** *(Optional)* `The relative path to category images`

8. Click OK. The two context parameters are now listed:



9. Click the XML tab to view the XML content that has been added to the deployment descriptor. The following `<context-param>` entries have been added:

```
10.  <context-param>
11.      <description>The relative path to product images</description>
12.      <param-name>productImagePath</param-name>
13.      <param-value>img/products/</param-value>
14.  </context-param>
15.  <context-param>
16.      <description>The relative path to category images</description>
17.      <param-name>categoryImagePath</param-name>
18.      <param-value>img/categories/</param-value>
    </context-param>
```

19. To test whether the values for the context parameters are accessible to web pages, open any of the project's web pages in the editor and enter EL expressions using the `initParam` implicit object. For example, open `index.jsp` and enter the following (New code in **bold**):

```
20.  <div id="indexLeftColumn">
21.      <div id="welcomeText">
22.          <p>[ welcome text ]</p>
23.
24.          <!-- test to access context parameters -->
25.          categoryImagePath: ${initParam.categoryImagePath}
26.          productImagePath: ${initParam.productImagePath}
27.      </div>
    </div>
```

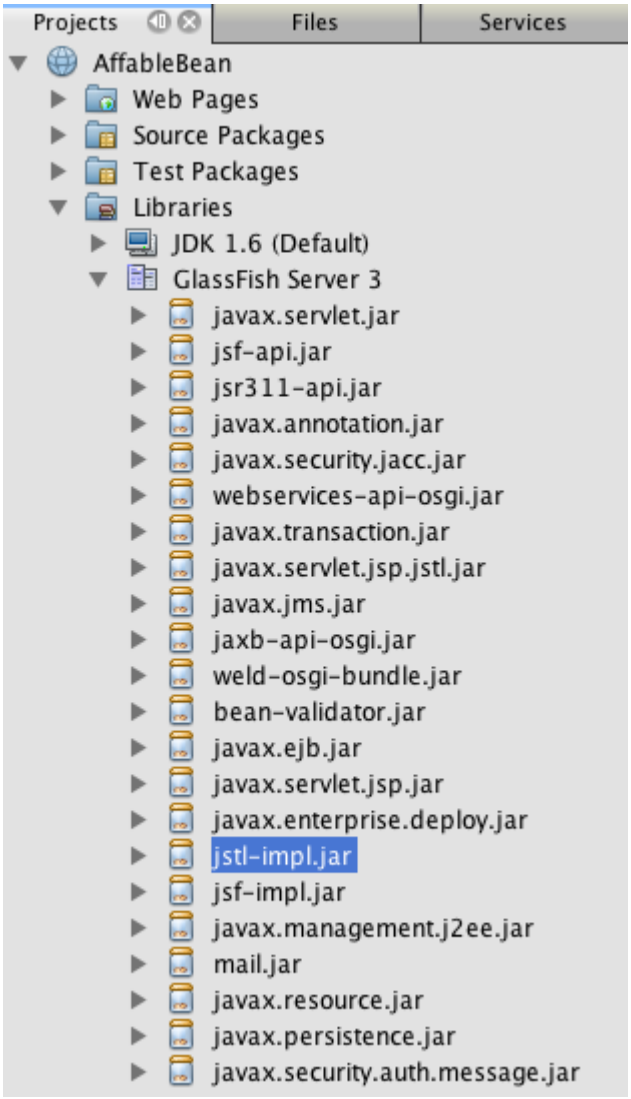
28. Run the project. Click the Run Project ( ) button. The project's index page opens in the browser, and you see the values for the `categoryImagePath` and

`productImagePath` context parameters displayed in the page.

Working with JSTL

So far in this tutorial unit, you've established how to access data from the `affablebean` database, add image resources to the project, and have set up several context parameters. In this final section, you combine these achievements to plug the product and category images into the application. In order to do so effectively, you need to begin taking advantage of the JavaServer Pages Standard Tag Library (JSTL).

Note that you do not have to worry about adding the JSTL JAR file (`jstl-impl.jar`) to your project's classpath because it already exists. When you created the `AffableBean` project and selected GlassFish as your development server, the libraries contained in the server were automatically added to your project's classpath. You can verify this in the Projects window by expanding the `AffableBean` project's Libraries > GlassFish Server 3 node to view all of the libraries provided by the server.



The `jstl-impl.jar` file is GlassFish' implementation of JSTL, version 1.2.

You can also download the GlassFish JSTL JAR file separately from:
https://jstl.dev.java.net/download.html

Before embarking upon an exercise involving JSTL, one implementation detail needs to first be clarified. Examine the files contained in the `categories` and `products` folders and note that the names of the provided image files match the names of the category and product entries found in the database. This enables us to leverage the database data to dynamically call image files within the page. So, for example, if the web page needs to access the image for the broccoli product entry, you can make this happen using the following statement.

```
${initParam.productImagePath}broccoli.png
```

After implementing a JSTL `forEach` loop, you'll be able to replace the hard-coded name of the product with an EL expression that dynamically extracts the name of the product from the database, and inserts it into the page.

```
${initParam.productImagePath}${product.name}.png
```

Begin by integrating the category images into the index page, then work within the category page so that data pertaining to the selected category is dynamically handled.

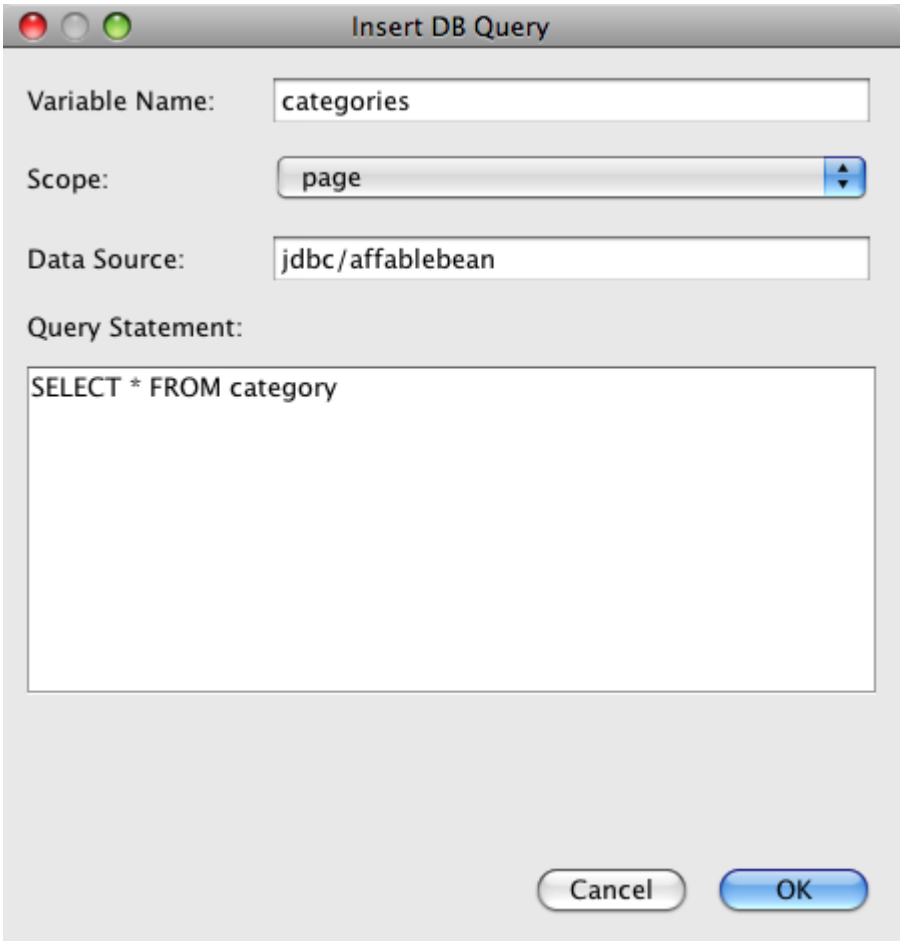
- index page
- category page

index page

1. In the Projects window, double-click the `index.jsp` node to open it in the editor. (If already opened, press Ctrl-Tab to select it in the editor.)
2. At the top of the file, before the first `<div>` tag, place your cursor on a blank line, then type '`db`' and press Ctrl-Space. In the code-completion pop-up window that displays, choose DB Query.

3. In the Insert DB Query dialog, enter the following details:

- **Variable Name:** `categories`
- **Scope:** `page`
- **Data Source:** `jdbc/affablebean`
- **Query Statement:** `SELECT * FROM category`



4. Click OK. The dialog generates an SQL query using JSTL `<sql:query>` tags. Also, note that the required reference to the `sql taglib` directive has been automatically inserted at the top of the page. (Changes displayed in **bold**.)

```
5. <%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
6. <%--
7.     Document   : index
8.     Created on : Sep 5, 2009, 4:32:42 PM
9.     Author     : nbuser
10.  --%>
11.
12.  <sql:query var="categories" dataSource="jdbc/affablebean">
13.      SELECT * FROM category
14.  </sql:query>
15.
16.              <div id="indexLeftColumn">
17.                  <div id="welcomeText">
18.                      <p>[ welcome text ]</p>
```

The SQL query creates a result set which is stored in the `categories` variable. You can then access the result set using EL syntax, e.g., `${categories}` (demonstrated below).

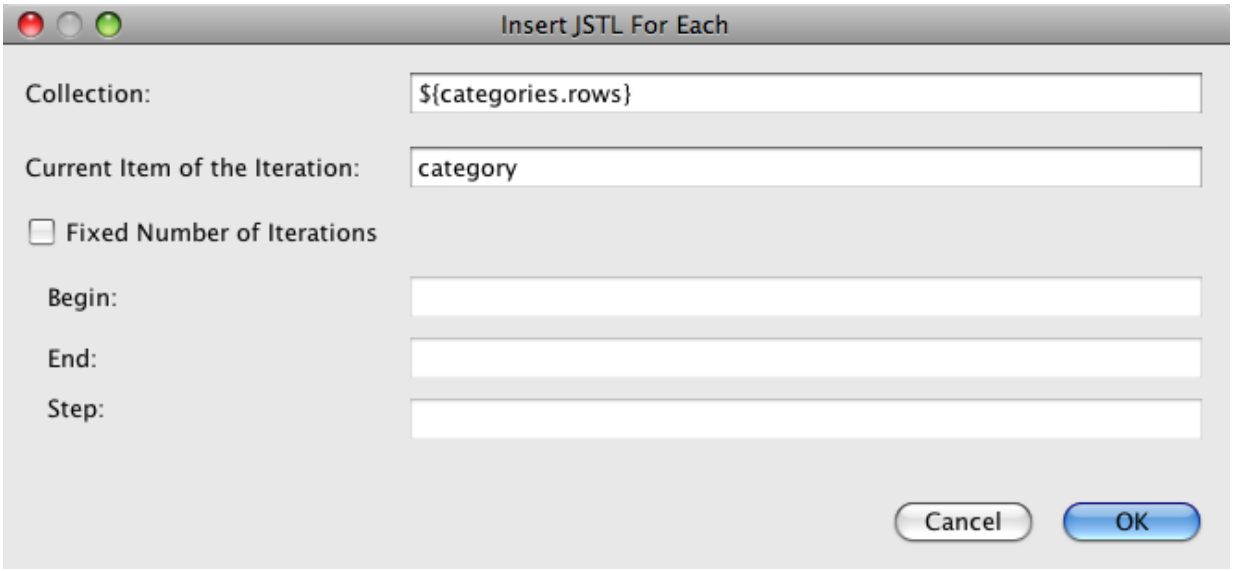
19. Place your cursor at the end of '`<div id="indexRightColumn">`' (line 22), hit return, type '`jstl`' then press Ctrl-Space and choose JSTL For Each.

```
 8  <sql:query var="categories" dataSource="jdbc/affablebean">
 9      SELECT * FROM category
10  </sql:query>
11
12              <div id="indexLeftColumn">
13                  <div id="welcomeText">
14                      <p>[ welcome text ]</p>
15
16                      <!-- test to access context parameters -->
17                      categoryImagePath: ${initParam.categoryImagePath}
18                      productImagePath: ${initParam.productImagePath}
19                  </div>
20              </div>
21
22              <div id="indexRightColumn">
23                  jstl
24                  <                                    ox">
25                  <
26                                                   ategoryLabelText">dairy</span>
27                  <
28                  </div>
29                  <div class="categoryBox">
30                      <a href="#">
```

JSTL Choose JSP
JSTL For Each JSP
JSTL If JSP

20. In the Insert JSTL For Each dialog, enter the following details:

- **Collection:** `${categories.rows}`
- **Current Item of the Iteration:** `category`



21. Click OK. The dialog sets up syntax for a JSTL `forEach` loop using `<c:forEach>` tags. Also, note that the required reference to the `core taglib` directive has been automatically inserted at the top of the page. (Changes displayed in **bold**.)

```
22.  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
23.  <%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
24.
25.      ...
26.
27.      <div id="indexRightColumn">
28.          <c:forEach var="category" items="categories.rows">
29.          </c:forEach>
```

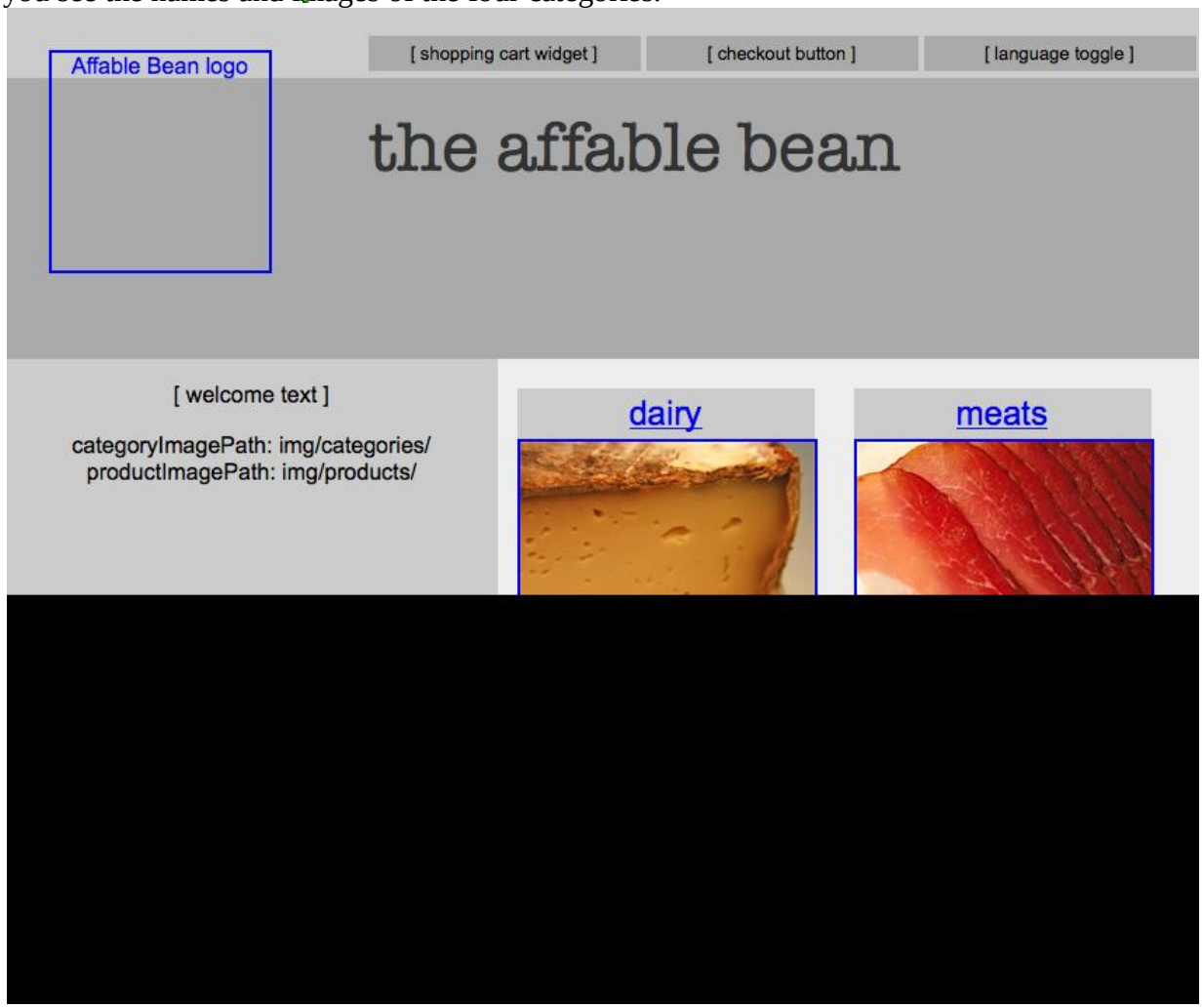
```
<div class="categoryBox">
```

If you are wondering what '`rows`' refers to in the generated code, recall that the `categories` variable represents a result set. More specifically, `categories` refers to an object that implements the `javax.servlet.jsp.jstl.sql.Result` interface. This object provides properties for accessing the rows, column names, and size of the query's result set. When using dot notation as in the above example, '`categories.rows`' is translated in Java to '`categories.getRows()`'.

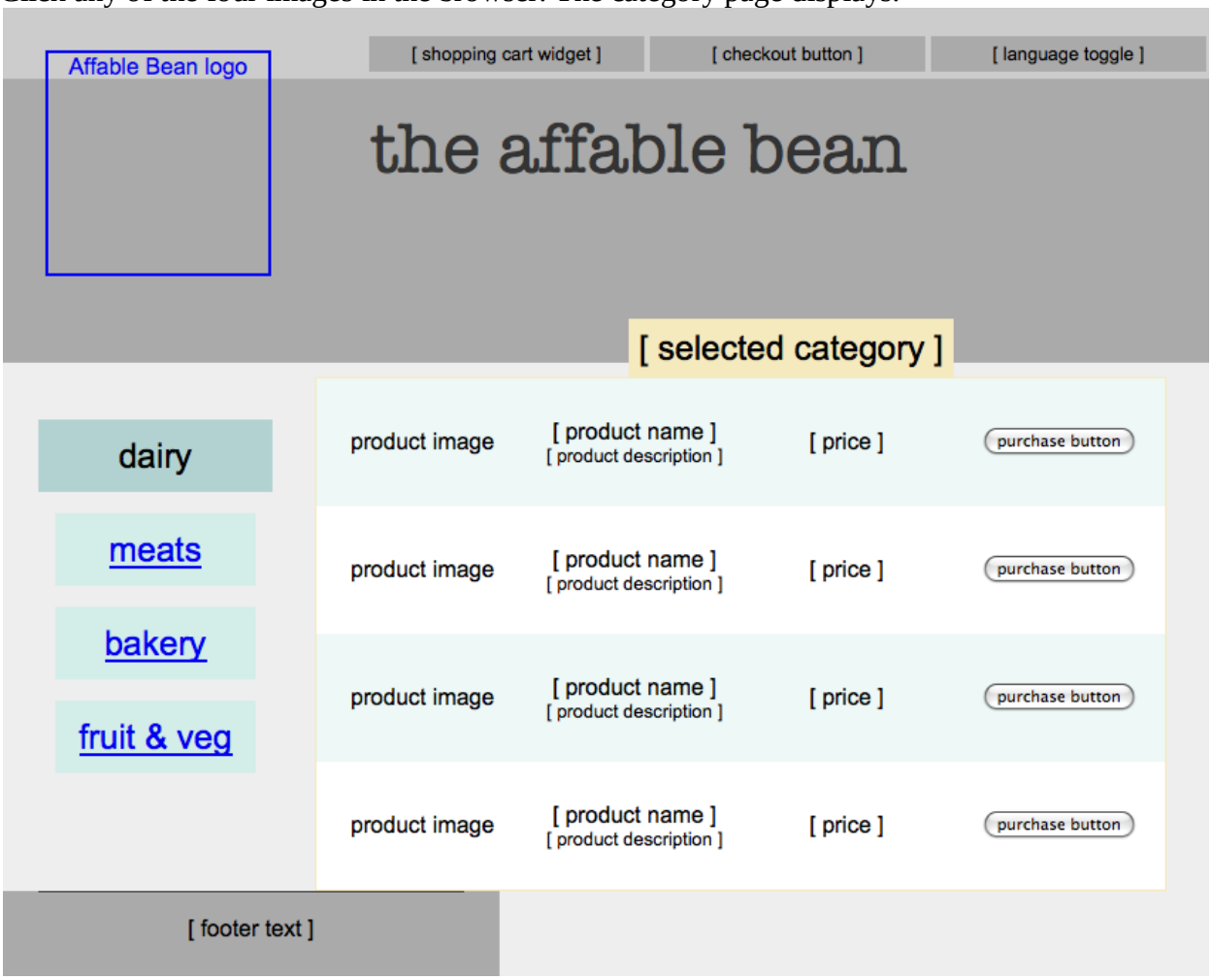
30. Integrate the `<c:forEach>` tags into the page. You can nest the `<div class="categoryBox">` tags within the `forEach` loop so that HTML markup is generated for each of the four categories. Use EL syntax to extract the `category` table's `id` and `name` column values for each of the four records. Make sure to delete the other `<div class="categoryBox">` tags which exist outside the `forEach` loop. When you finish, the complete `index.jsp` file will look as follows. (`<c:forEach>` tags and contents are displayed in **bold**.)

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
<%--
    Document   : index
    Created on : Sep 5, 2009, 4:32:42 PM
    Author     : nbuser
--%>

<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

            <div id="indexLeftColumn">
                <div id="welcomeText">
                    <p>[ welcome text ]</p>

                    <!-- test to access context parameters -->
                    categoryImagePath:
${initParam.categoryImagePath}
                    productImagePath: ${initParam.productImagePath}
                </div>
            </div>

            <div id="indexRightColumn">
                <c:forEach var="category"
items="${categories.rows}">
                    <div class="categoryBox">
                        <a href="category?${category.id}">

                            <span
class="categoryLabelText">${category.name}</span>

                            <img
src="${initParam.categoryImagePath}${category.name}.jpg"
                                 alt="${category.name}">
                        </a>
                    </div>
                </c:forEach>
            </div>
```

65. Click the Run Project ( ) button. The project's index page opens in the browser, and you see the names and images of the four categories.



66. Click any of the four images in the browser. The category page displays.



To understand how linking takes place between the index and category pages, reexamine the HTML anchor tags within the `forEach` loop:

```
<a href="category?${category.id}">
```

When a user clicks the image link in the browser, a request for '`category`' is sent to the application's context root on the server. In your development environment, the URL is:

```
http://localhost:8080/AffableBean/category
```

which can be explained as follows:

- `http://localhost:8080`: The default location of the GlassFish server on your computer
- `/AffableBean`: The context root of your deployed application
- `/category`: The path to the request

Recall that in Working with the Deployment Descriptor, you mapped a request for '`/category`' to the `ControllerServlet`. Currently, the `ControllerServlet`

internally forwards the request to `/WEB-INF/view/category.jsp`, which is why the category page displays upon clicking an image link.

You can verify the application's context root by expanding the Configuration Files node in the Projects window, and opening the `sun-web.xml` file. The `sun-web.xml` file is a deployment descriptor specific to GlassFish.

Also, note that a question mark (`?`) and category ID are appended to the requested URL.

```
<a href="category?${category.id}">
```

This forms the *query string*. As is demonstrated in the next section, you can apply `(pageContext.request.queryString}` to extract the value of the query string from the request. You can then use the category ID from the query string to determine which category details need to be included in the response.

category page

Three aspects of the category page need to be handled dynamically. The left column must indicate which category is selected, the table heading must display the name of the selected category, and the table must list product details pertaining to the selected category. In order to implement these aspects using JSTL, you can follow a simple, 2-step pattern:

1. Retrieve data from the database using the JSTL `sql` tag library.
2. Display the data using the JSTL `core` library and EL syntax.

Tackle each of the three tasks individually.

Display selected category in left column

1. In the Projects window, double-click the `category.jsp` node to open it in the editor. (If already opened, press Ctrl-Tab to select it in the editor.)
2. Add the following SQL query to the top of the file.

```
<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>
```

Either use the Insert DB Query dialog as described above, or use the editor's code suggestion and completion facilities by pressing Ctrl-Space while typing.

5. Between the `<div id="categoryLeftColumn">` tags, replace the existing static placeholder content with the following `<c:forEach>` loop.

```
<div id="categoryLeftColumn">

    <c:forEach var="category" items="${categories.rows}">

        <c:choose>
            <c:when test="${category.id ==
pageContext.request.queryString}">
                <div class="categoryButton" id="selectedCategory">
```

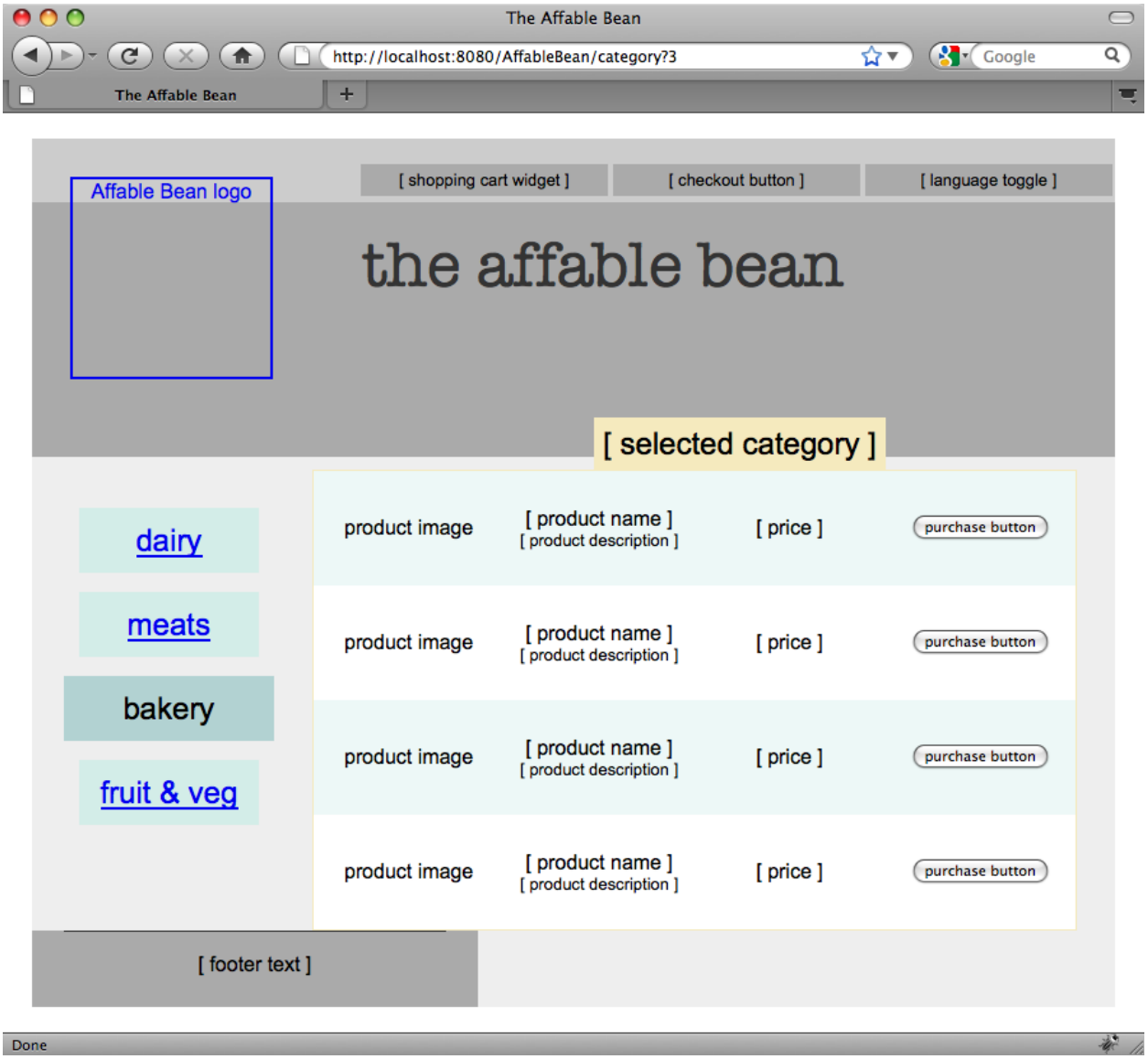
```
                        <span class="categoryText">
                            ${category.name}
                        </span>
                    </div>
                </c:when>
                <c:otherwise>
                    <a href="category?${category.id}"
class="categoryButton">
                        <div class="categoryText">
                            ${category.name}
                        </div>
                    </a>
                </c:otherwise>
            </c:choose>

        </c:forEach>

</div>
```

In the above snippet, you access the request's query string using '`pageContext.request.queryString`'. `pageContext` is another implicit object defined by the JSP Expression Language. The EL expression uses the `PageContext` to access the current request (an `HttpServletRequest` object). From `HttpServletRequest`, the `getQueryString()` method is called to obtain the value of the request's query string.

29. Make sure to add the JSTL `core` and `sql taglib` directives to the top of the page. (This is done automatically when using the editor's code suggestion and completion facilities.)

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
```

31. Run the project. In the browser, navigate to the category page and click the category buttons in the left column. Each time you click, the page refreshes highlighting the selected category.



Also, note that the ID of the selected category is displayed in the page's URL. (In the above image, the bakery category is selected, and '`3`' is appended to the URL in the browser's navigation toolbar.)

Your servlet container (i.e., GlassFish) converts JSP pages into servlets before running them as part of a project. You can view the generated servlet for a JSP page by right-clicking the page node in the Projects window and choosing View Servlet. Of course, you first need to run the project so that the servlet is generated. Taking the `index.jsp` file as an example, when you choose View Servlet, the IDE displays a read-only copy of the generated servlet, `index_jsp.java`, in the editor. The servlet exists on the server at: *`<gf-install-dir>`*`/glassfish/domains/domain1/generated/jsp/AffableBean/org/apache/jsp/index_jsp.java`.

Examining Implicit Object Values using the IDE's Debugger

You can use the IDE's Java debugger to examine values for implicit objects. To do so, set a breakpoint on a line containing JSP or JSTL syntax in a JSP page, then run the debugger.

When the debugger suspends on the breakpoint, you can open the Variables window (Window > Debugging > Variables) to inspect values currently held by the application.

Taking your current implementation of `category.jsp` as an example, perform the following steps:

1. Set a breakpoint on the line containing:

   ```
   <c:when test="${category.id == pageContext.request.queryString}">
   ```

   (To set a breakpoint, click in the left margin of the line. A breakpoint ( ) icon displays.)

2. In the IDE's main toolbar, click the Debug Project ( ) button. A debugging session is activated for the project, and the application's index page opens in the browser.
3. Click the bakery category in the index page. (You know that the ID for the bakery category is '3').
4. Return to the IDE, and note that the debugger is suspended on the line containing the breakpoint. When suspended, the margin shows a green arrow on the breakpoint ( ), and the line displays with green background.
5. Open the Variables window (Ctrl-Shift-1) and expand the Implicit Objects > pageContext > request > queryString node. Inspect the variable value and note that the value is '3', corresponding to the category ID from your selection.
6. Press the Finish Debugger Session ( ) button to terminate the debugger session.

Display title heading above product table

1. Add the following SQL query to the top of the file, underneath the query you just implemented. (New query is shown in **bold**.)

```
<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

<sql:query var="selectedCategory" dataSource="jdbc/affablebean">
    SELECT name FROM category WHERE id = ?
    <sql:param value="${pageContext.request.queryString}"/>
</sql:query>
```

9. Use JSP EL syntax to extract the category name from the query and display it in the page. Make the following change to the `<p id="categoryTitle">` element. (Displayed in **bold**.)

```
 <p id="categoryTitle">
     <span style="background-color: #f5eabe; padding:
7px;">${selectedCategory.rows[0].name}</span>
</p>
```

   Since the result from the `selectedCategory` query contains only one item (i.e., user can select only one category), you can retrieve the first row of the result set using '`selectedCategory.rows[0]`'. If a user selects the 'meats' category for example, the

returned expression would be '`{name=meats}`'. You could then access the category name with '`${selectedCategory.rows[0].`**`name`**`}`'.

12. Save (Ctrl-S; ⌘-S on Mac) changes made to the file.
13. Return to the browser and refresh the category page. The name of the selected category now displays above the product table.

Note: As demonstrated in this and the previous step, you do not need to explicitly recompile, deploy, and run the project with each change to your code base. The IDE provides a Deploy on on Save feature, which is enabled for Java web projects by default. To verify that the feature is activated, right-click your project node in the Projects window and choose Properties. In the Project Properties window, click the Run category and examine the 'Deploy on Save' option.

Display product details within the table

1. Add the following SQL query to the top of the file, underneath the previous queries you implemented. (New query is shown in **bold**.)

```
<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

<sql:query var="selectedCategory" dataSource="jdbc/affablebean">
    SELECT name FROM category WHERE id = ?
    <sql:param value="${pageContext.request.queryString}"/>
</sql:query>

 <sql:query var="categoryProducts" dataSource="jdbc/affablebean">
     SELECT * FROM product WHERE category_id = ?
     <sql:param value="${pageContext.request.queryString}"/>
</sql:query>
```

14. Between the `<table id="productTable">` tags, replace the existing static table row placeholders (`<tr>` tags) with the following `<c:forEach>` loop. (Changes are displayed in **bold**.)

```
 <table id="productTable">

     <c:forEach var="product" items="${categoryProducts.rows}"
varStatus="iter">

         <tr class="${((iter.index % 2) == 0) ? 'lightBlue' :
'white'}">
             <td>
                 <img
src="${initParam.productImagePath}${product.name}.png"
                      alt="image of ${product.name}">
             </td>
             <td>
                 ${product.name}
                 <br>
                 <span
class="smallText">${product.description}</span>
             </td>
             <td>
                 &euro; ${product.price} / unit
             </td>
             <td>
                 <form action="addToCart" method="post">
                     <input type="hidden"
                            name="productId"
                            value="${product.id}">
                     <input type="submit"
                            value="add to cart">
                 </form>
             </td>
         </tr>

     </c:forEach>

</table>
```

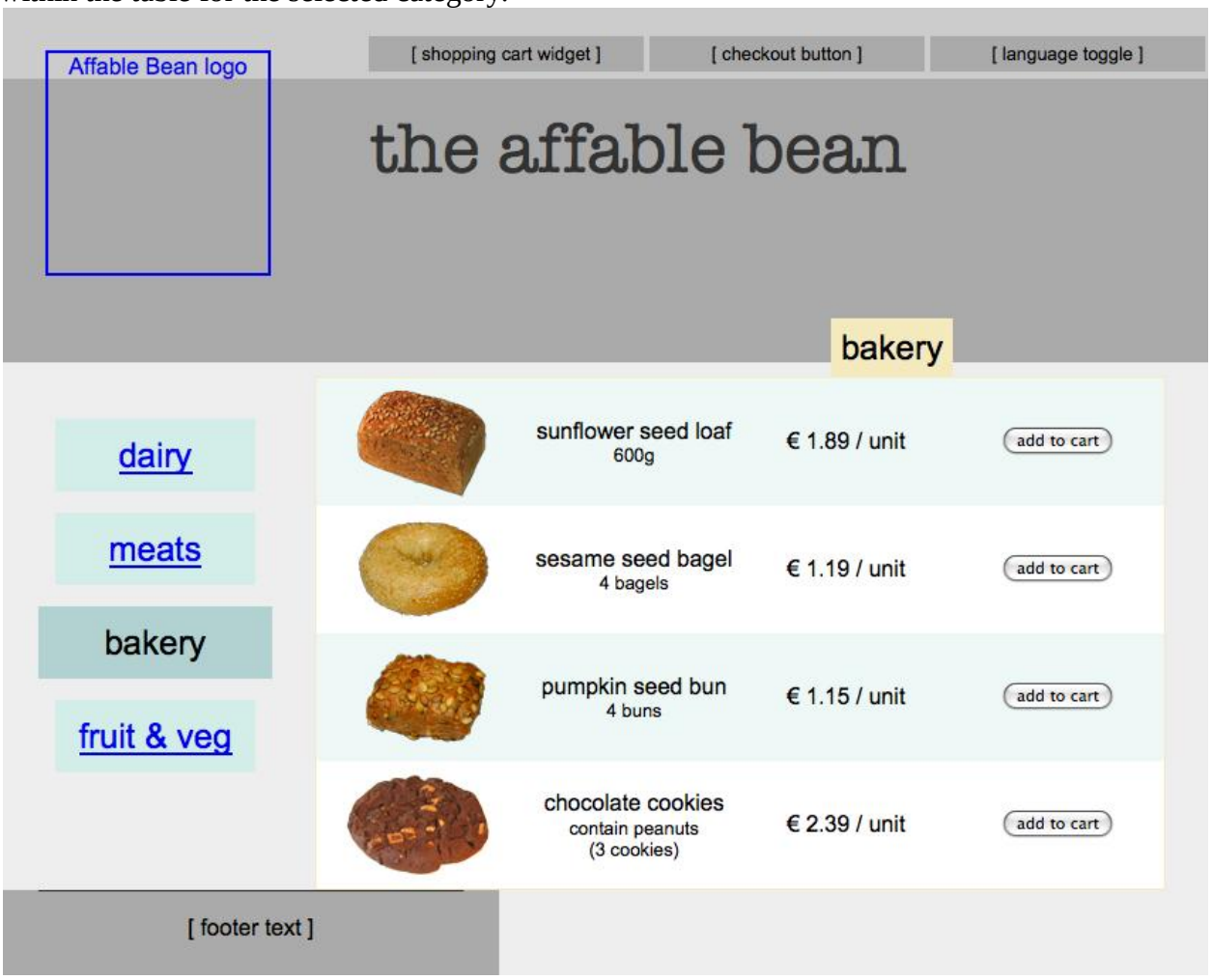
Note that in the above snippet an EL expression is used to determine the background color for table rows:

```
class="${((iter.index % 2) == 0) ? 'lightBlue' : 'white'}"
```

The API documentation for the `<c:forEach>` tag indicates that the `varStatus` attribute represents an object that implements the `LoopTagStatus` interface. Therefore, `iter.index` retrieves the index of the current round of the iteration. Continuing with the expression, `(iter.index % 2) == 0)` evaluates the remainder when `iter.index` is divided by `2`, and returns a boolean value based on the outcome. Finally, an EL conditional operator (`? :`) is used to set the returned value to '`lightBlue`' if true, '`white`' otherwise.

For a description of JSP Expression Language operators, see the Java EE 5 Tutorial: JavaServer Pages Technology > Unified Expression Language > Operators.

45. Save (Ctrl-S; ⌘-S on Mac) changes made to the file.
46. Return to the browser and refresh the category page. Product details now display within the table for the selected category.



You have now completed this tutorial unit. In it, you explored how to connect your application to the database by setting up a connection pool and data source on the server, then referenced the data source from the application. You also created several context parameters, and learned how to access them from JSP pages. Finally, you implemented JSTL tags into the application's web pages in order to dynamically retrieve and display database data.

You can download and examine AffableBean_snapshot3.zip if you'd like to compare your work with the solution project. The solution project contains enhancements to the HTML

markup and stylesheet in order to properly display all provided images. It also provides welcome page text, and a basic implementation for the page footer.

Send Us Your Feedback

Troubleshooting

If you are having problems, see the troubleshooting tips below. If you continue to have difficulty, or would like to provide constructive feedback, use the Send us Your Feedback link.

- You receive the following exception:

  ```
  org.apache.jasper.JasperException: PWC6188: The absolute uri:
  http://java.sun.com/jsp/jstl/core cannot be resolved in either
  web.xml or the jar files deployed with this application
  ```

  This is a known issue for NetBeans IDE 6.9. Try to deploy the project, then access the file by typing its URL in the browser. For example, if you are trying to view `testDataSource.jsp` in a browser, enter '`http://localhost:8080/AffableBean/test/testDataSource.jsp`' in the browser's URL field directly. Otherwise, add the IDE's JSTL 1.1 library to the project. In the Projects window, right-click the Libraries node and choose Add Library. Select JSTL 1.1. For more information, see: http://forums.netbeans.org/topic28571.html.

- You receive the following exception:

  ```
  javax.servlet.ServletException: javax.servlet.jsp.JspException:
  Unable to get connection, DataSource invalid: "java.sql.SQLException:
  Error in allocating a connection. Cause: Class name is wrong or
  classpath is not set for :
  com.mysql.jdbc.jdbc2.optional.MysqlDataSource"
  ```

  This can occur when the MySQL driver has not been added to the domain `lib` folder. (Note that after adding, it is necessary to restart the server if it is already running.)

- You receive the following exception:

  ```
  javax.servlet.ServletException: javax.servlet.jsp.JspException:
  Unable to get connection, DataSource invalid: "java.sql.SQLException:
  No suitable driver found for jdbc/affablebean"
  ```

  This can occur when the `jdbc/affablebean` resource reference hasn't been added to the `web.xml` deployment descriptor.

- You receive the following exception:

  ```
  javax.servlet.ServletException: javax.servlet.jsp.JspException:
  Unable to get connection, DataSource invalid: "java.sql.SQLException:
  Error in allocating a connection. Cause: Connection could not be
  ```

```
allocated because: Access denied for user 'root'@'localhost' (using
password: YES)"
```

This can occur when you are using an incorrect username/password combination. Make sure the username and password you use to connect to the MySQL server are correctly set for your connection pool in the `sun-resources.xml` file. Also, check that the username and password are

correctly set for the connection pool in the GlassFish Administration Console.

The NetBeans E-commerce Tutorial - Adding Entity Classes and Session Beans

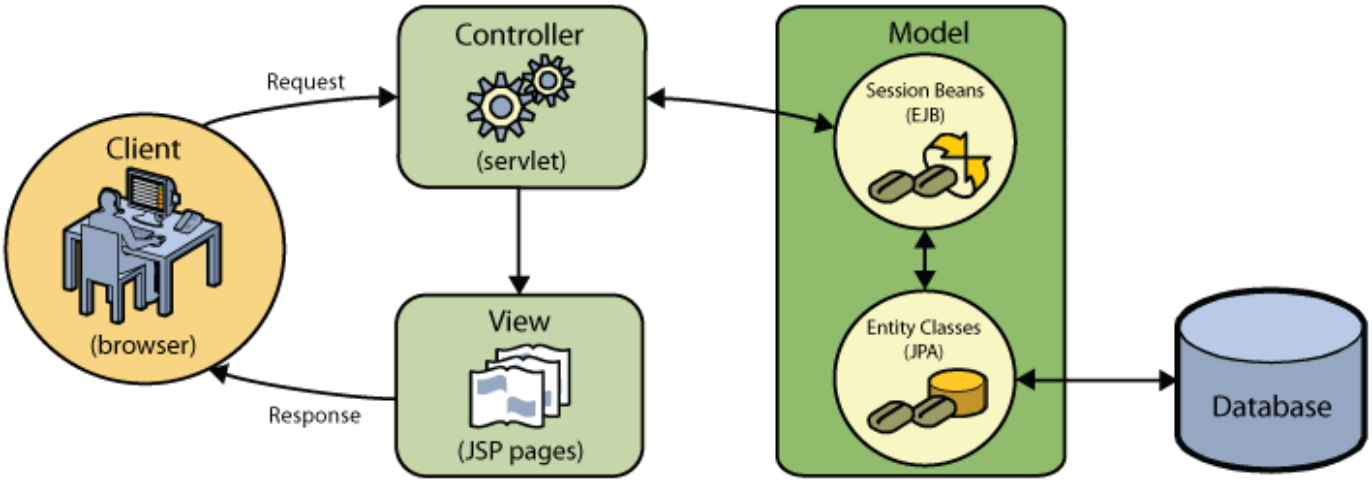
Tutorial Contents

This tutorial unit introduces the Enterprise JavaBeans (EJB) and Java Persistence (JPA) technologies. In it, you use two of the IDE's wizards that are essential to Java EE development. These are:

- **Entity Classes from Database wizard:** Creates a Java Persistence API entity class for each selected database table, complete with named query annotations, fields representing columns, and relationships representing foreign keys.

- **Session Beans for Entity Classes wizard:** Creates an EJB session facade for each entity class with basic access methods.

These two wizards provide an efficient way for you to quickly set up the model for your application. If you reexamine the MVC diagram for the application you are building, you can see where EJB session beans and JPA entity classes fit into its structure.



In this unit, the entity classes you create form a Java-based representation of the `affablebean` database. While each entity class represents a database table, instances of entity classes correspond to records that can be saved (i.e., *persisted*) to the database. The business logic of the application is encapsulated by session beans, which can either be used as *facade* classes that enable CRUD (Create-Read-Update-Delete) access to entities (as demonstrated here), or they can contain code that implements actions specific to your application. (An example of this is provided in Unit 9: Integrating Transactional Business Logic).

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
GlassFish server	v3 or Open Source Edition 3.0.1
AffableBean project	snapshot 3

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.

- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could download the GlassFish server independently, but the version provided with the NetBeans download has the added benefit of being automatically registered with the IDE.
- You can follow this tutorial unit without having completed previous units. To do so, see the setup instructions, which describe how to prepare the database and establish connectivity between the IDE, GlassFish, and MySQL.
- Snapshot 4 of the `AffableBean` project is available for download and corresponds to state the project after completing this unit using NetBeans IDE 6.9.

What are EJB and JPA Technologies?

Up until now, the project that you've been developing in this tutorial could be run in a web server with a servlet container, such as Apache Tomcat. After all, you've so far only made use of JSTL and servlet technologies, and are connecting to the database directly using JDBC. In fact, you could theoretically continue to develop the application using just these technologies, while manually coding for all aspects of your application, including thread-safety, transactions, and security. However, using Enterprise beans with JPA entity classes allows you focus on the business logic of your application while relying on solutions that have already been tried and tested. The following sections introduce the two technologies and define their role in EE development.

Enterprise JavaBeans

The official EJB product page describes EnterPrise JavaBeans technology as a "server-side component architecture" that "enables rapid and simplified development of distributed, transactional, secure and portable applications." You can apply EJBs (i.e., Enterprise beans) to your projects, and the services provided by the technology remain transparent to you as a developer, thus eliminating the tedious and often error-prone task of adding a lot of boiler plate code which would otherwise be required. If you are new to EE development, you may question the need for EJBs in your Java web application. The book EJB 3 In Action, by Debu Panda, Reza Rahman and Derek Lane, paraphrases the role of EJB technology nicely:

Although many people think EJBs are overkill for developing relatively simple web applications of moderate size, nothing could be further from the truth. When you build a house, you don't build everything from scratch. Instead, you buy materials or even the services of a contractor as you need it. It isn't too practical to build an enterprise application from scratch either. Most server-side applications have a lot in common, including churning business logic, managing application state, storing and retrieving information from a relational database, managing transactions, implementing security, performing asynchronous processing, integrating systems, and so on.

As a framework, the EJB container provides these kinds of common functionality as out-of-

the-box services so that your EJB components can use them in your applications without reinventing the wheel. For instance, let's say that when you build a credit card module in your web application, you write a lot of complex and error-prone code to manage transactions and security access control. You could have avoided that by using the declarative transaction and security services provided by the EJB container. These services as well as many others are available to EJB components when they are deployed in an EJB container. This means writing high-quality, feature-rich applications much faster than you might think.[1]

You can think of EJB both as components, or Java classes that are incorporated in your project, as well as a *framework* that provides numerous enterprise-related services. Some of the services that we take advantage of in this tutorial are described in EJB 3 In Action as follows:

- **Pooling:** For each EJB component, the EJB platform creates a pool of component instances that are shared by clients. At any point in time, each pooled instance is only allowed to be used by a single client. As soon as an instance is finished servicing a client, it is returned to the pool for reuse instead of being frivolously discarded for the garbage collector to reclaim.
- **Thread Safety:** EJB makes all components thread-safe and highly performant in ways athat are completely invisible. This means that you can write your server components as if you were developing a single-threaded desktop application. It doesn't matter how complex the component itself is; EJB will make sure it is thread-safe.
- **Transactions:** EJB supports declarative transaction management that helps you add transactional behavior to components using simple configuration instead of code. In effect, you can designate any component method to be transactional. If the method completes normally, EJB commits the transaction and makes the data changes made by the method permanent. Otherwise the transaction is rolled back.

Although the tutorial does not require the use of EJB security, this service is worth mentioning here:

- **Security:** EJB supports integration with the Java Authentication and Authorization Service (JAAS) API, so it is very easy to completely externalize security and secure an application using simple configuration instead of cluttering up your application with security code.[2]

Java Persistence

In the context of Java Enterprise, *persistence* refers to the act of automatically storing data contained in Java objects into a relational database. The Java Persistence API (JPA) is an object-relational mapping (ORM) technology that enables applications to manage data between Java objects and a relational database in a way that is transparent to the developer. This means that you can apply JPA to your projects by creating and configuring a set of Java classes (*entities*) that mirror your data model. Your application can then access these entities as though it were directly accessing the database.

There are various benefits to using JPA in your projects:

- JPA has its own rich, SQL-like query language for static and dynamic queries. Using the Java Persistence Query Language (JPQL), your applications remain portable across different database vendors.
- You can avoid the task of writing low-level, verbose and error-prone JDBC/SQL code.
- JPA transparently provides services for data caching and performance optimization.

What are Session Beans?

Enterprise session beans are invoked by a client in order to perform a specific business operation. The name *session* implies that a bean instance is available for the duration of a "unit of work". The EJB 3.1 specification describes a typical session object as having the following characteristics:

- Executes on behalf of a single client
- Can be transaction-aware
- Updates shared data in an underlying database
- Does not represent directly shared data in the database, although it may access and update such data
- Is relatively short-lived
- Is removed when the EJB container crashes. The client has to re-establish a new session object to continue computation.

EJB provides three types of session beans: *stateful*, *stateless*, and *singleton*. The following descriptions are adapted from the Java EE 6 Tutorial.

- **Stateful:** The state of the bean is maintained across multiple method calls. The "state" refers to the values of its instance variables. Because the client interacts with the bean, this state is often called the *conversational* state.
- **Stateless:** Stateless beans are used for operations that can occur in a single method call. When the method finishes processing, the client-specific state of the bean is not retained. A stateless session bean therefore does not maintain a conversational state with the client.
- **Singleton:** A singleton session bean is instantiated once per application, and exists for the lifecycle of the application. Singleton session beans are designed for circumstances where a single enterprise bean instance is shared across and concurrently accessed by clients.

For more information on EJB session beans, see the Java EE 6 Tutorial: What is a Session Bean?.

For purposes of developing the e-commerce application in this tutorial, we will only be working with stateless session beans.

About Specifications and Implementations

EJB and JPA technologies are defined by the following specifications:

- JSR 317: Java Persistence 2.0
- JSR 318: Enterprise JavaBeans 3.1

These specifications define the technologies. To apply a technology to your project however, you must use an *implementation* of the specification. When a specification becomes finalized, it includes a reference implementation, which is a free implementation of the technology. If you find this concept confusing, consider the following analogy: A musical composition (i.e., the notes on a page) defines a piece of music. When a musician learns the composition and records her performance, she provides an *interpretation* of the piece. In this manner the musical composition is likened to the technical specification, and the musician's recording corresponds to the specification's implementation.

See What is the Java Community Process? for an explanation of Java technical specifications, and how they are formally standardized.

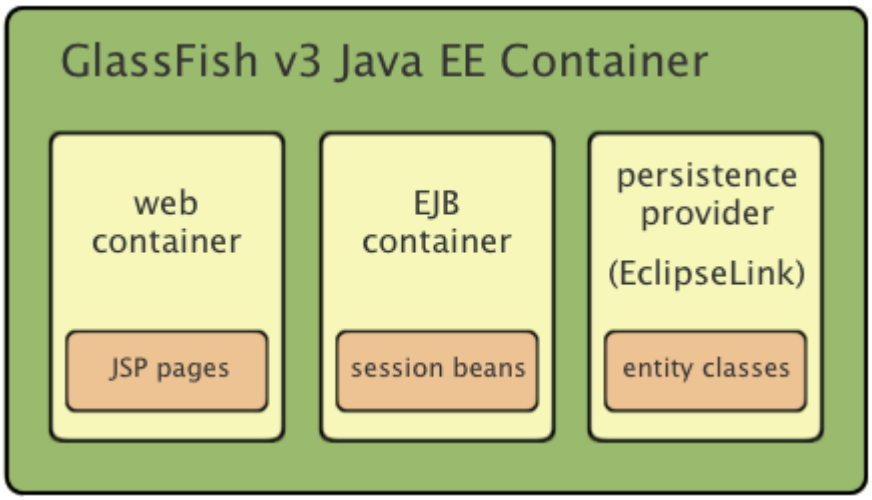
If you examine the download pages for the final releases of the EJB and JPA specifications, you'll find links to the following reference implementations:

- **JPA:** http://www.eclipse.org/eclipselink/downloads/ri.php
- **EJB:** http://glassfish.dev.java.net/downloads/ri

Implementations of the JPA specification are dubbed *persistence providers*, and the persistence provider which has been chosen as the reference implementation for the JPA 2.0 specification is EclipseLink.

If you examine the link for the EJB reference implementation, you'll come to a page that lists not only the implementation for EJB, but for all reference implementations provided by Project GlassFish. The reason for this is that Project GlassFish forms the reference implementation of the Java EE 6 platform specification (JSR 316). The GlassFish v3 application server (or the Open Source Edition), which you are using to build the e-commerce project in this tutorial, contains the reference implementations of all technologies developed under Project GlassFish. As such, it is referred to as a Java EE 6 *container*.

A Java EE container contains three essential components: a web (i.e., servlet) container, an EJB container, and a persistence provider. The deployment scenario for the e-commerce application is displayed in the diagram below. Entity classes that you create in this unit are managed by the persistence provider. The session beans that you create in this unit are managed by the EJB container. Views are rendered in JSP pages, which are managed by the web container.



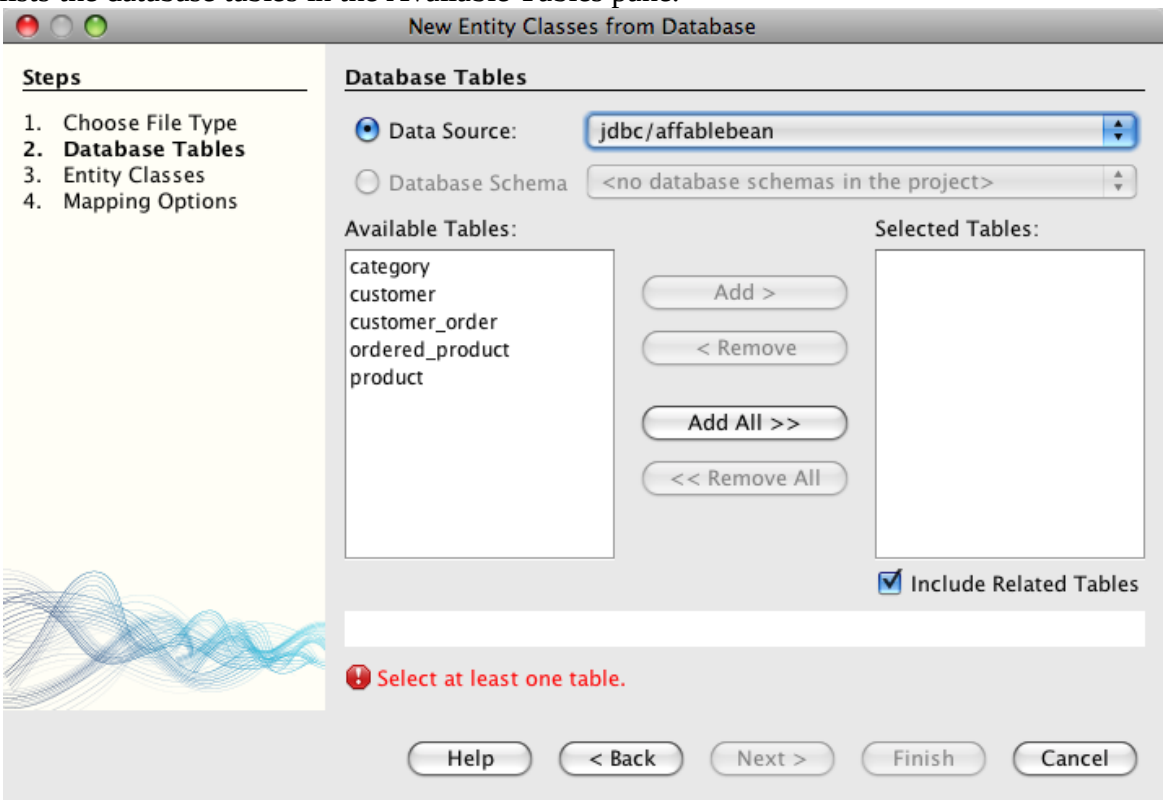
Adding Entity Classes

Begin by using the IDE's Entity Classes from Database wizard to generate entity classes based on the `affablebean` schema. The wizard relies on the underlying persistence provider to accomplish this task.

1. Open the project snapshot in the IDE. In the IDE, press Ctrl-Shift-O (⌘-Shift-O on Mac) and navigate to the location on your computer where you unzipped the downloaded file.
2. Press Ctrl-N (⌘-N on Mac) to open the File wizard.
3. Select the Persistence category, then select Entity Classes from Database. Click Next.
4. In Step 2: Database Tables, choose `jdbc/affablebean` from the Data Source drop-down list. The drop-down list is populated by data sources registered with the application server.

   When you choose the `jdbc/affablebean` data source, the IDE scans the database and

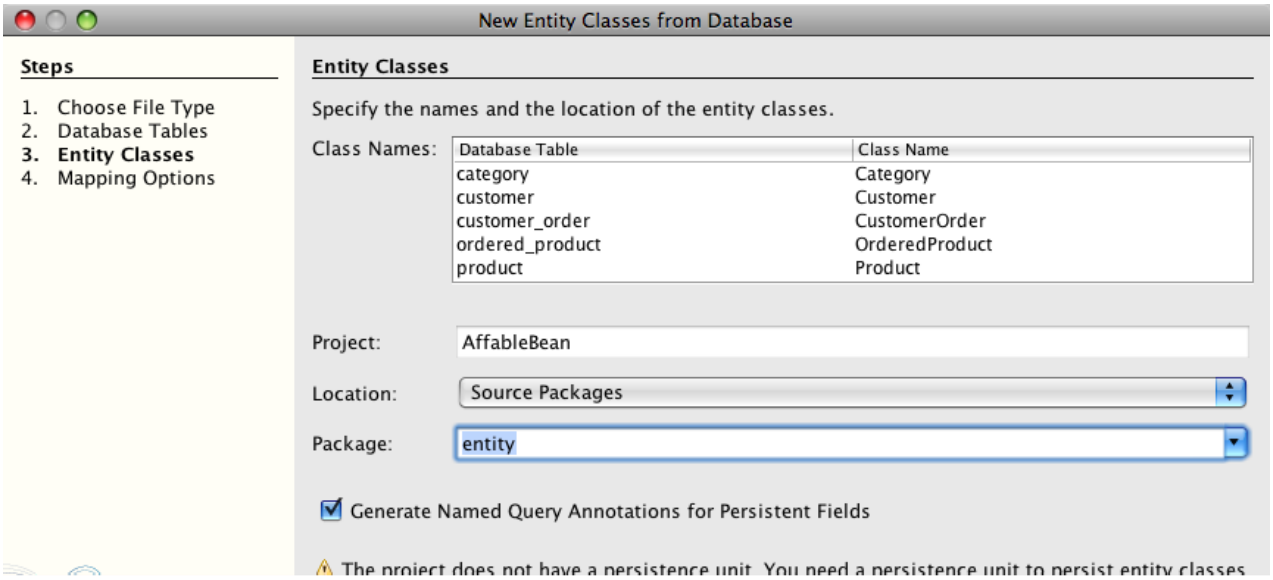
lists the database tables in the Available Tables pane.



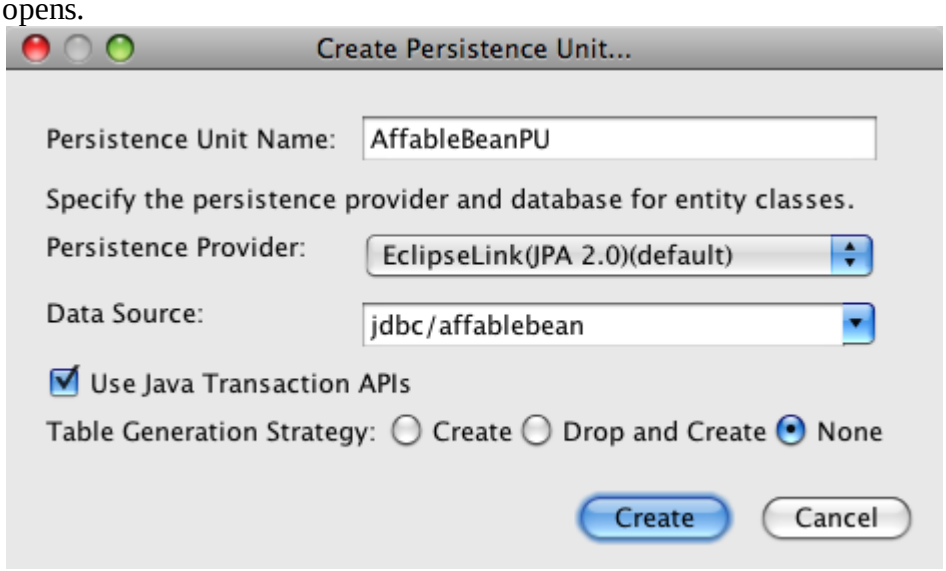
5. Click the Add All button, then click Next.
6. Step 3 of the Entity Classes from Database wizard differs slightly between NetBeans IDE 6.8 and 6.9. Depending on the version IDE you are using, perform the following steps.

- NetBeans IDE 6.8
- NetBeans IDE 6.9

NetBeans IDE 6.8



c. Type in **entity** in the Package field. The wizard will create a new package for the entity classes upon completing.
d. Click the Create Persistence Unit button. The Create Persistence Unit dialog opens.

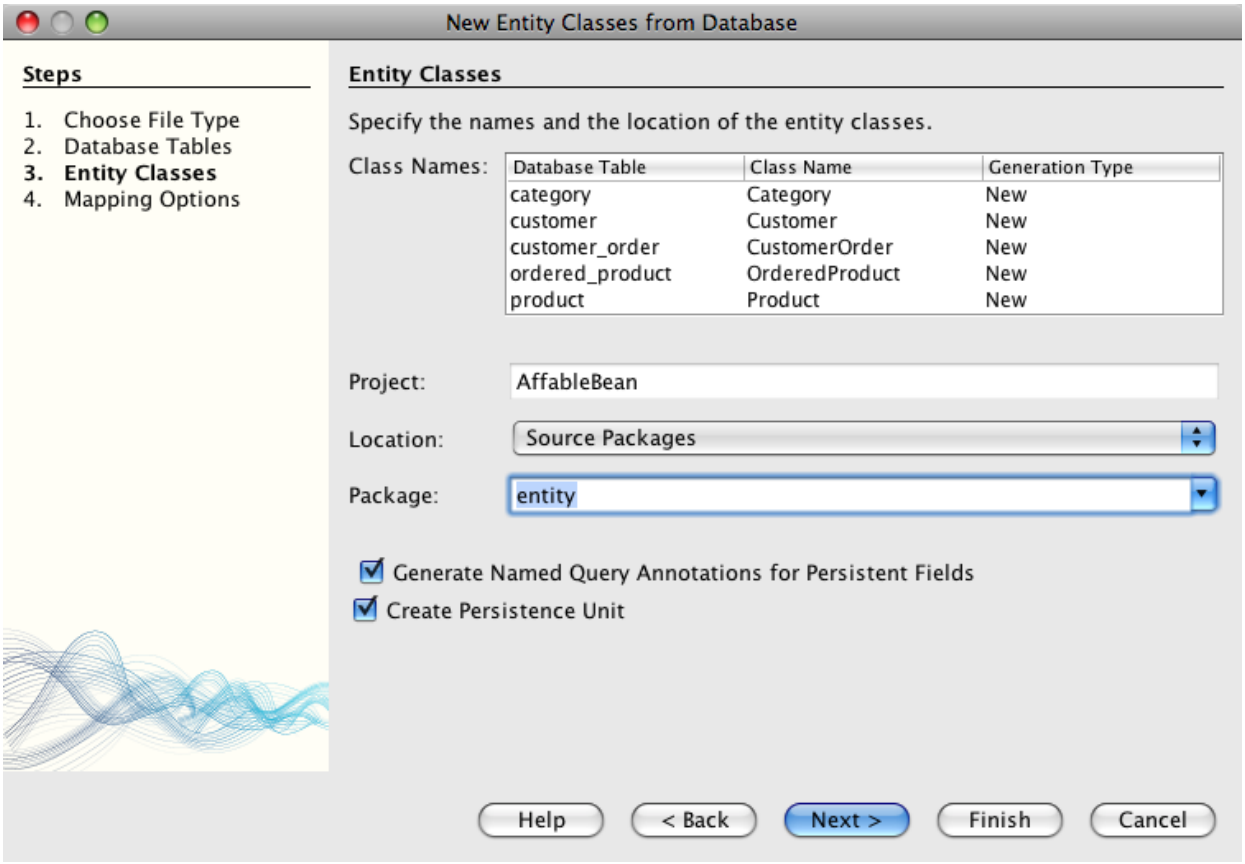


The dialog generates a `persistence.xml` file for your project, which is the configuration file used by your persistence provider. Note that 'EclipseLink (JPA 2.0)' is the default selection for the server associated with the project. Leave 'Table Generation Strategy' set to '`None`'. This prevents the persistence provider from affecting your database. (For example, if you want the persistence provider to delete then recreate the database based on the existing entity classes, you could set the strategy to '`Drop and Create`'. This action would then be taken each time the project was deployed.)

e. Click Create.

f. Back in Step 3: Entity Classes, note that the class names for the entities are based on database tables. For example, the `CustomerOrder` entity is mapped to the `customer_order` database table. Also note that the 'Generate Named Query Annotations for Persistent Fields' option is selected by default. We will be using various named queries later in the tutorial.
g. Continue to step 7 below.

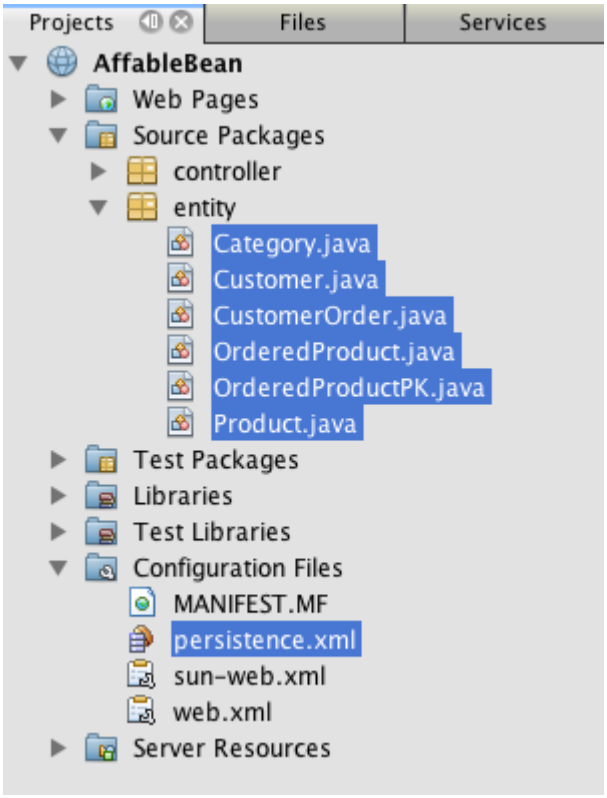
NetBeans IDE 6.9



h. Type in **entity** in the Package field. The wizard will create a new package for the entity classes upon completing.
i. Note the following:

- The class names for the entities are based on database tables. For example, the `CustomerOrder` entity will be mapped to the `customer_order` database table.
- The 'Generate Named Query Annotations for Persistent Fields' option is selected by default. We will be using various named queries later in the tutorial.
- The 'Create Persistence Unit' option is selected by default. This means that the wizard will also generate a `persistence.xml` file for your project, which is the configuration file used by your persistence provider.

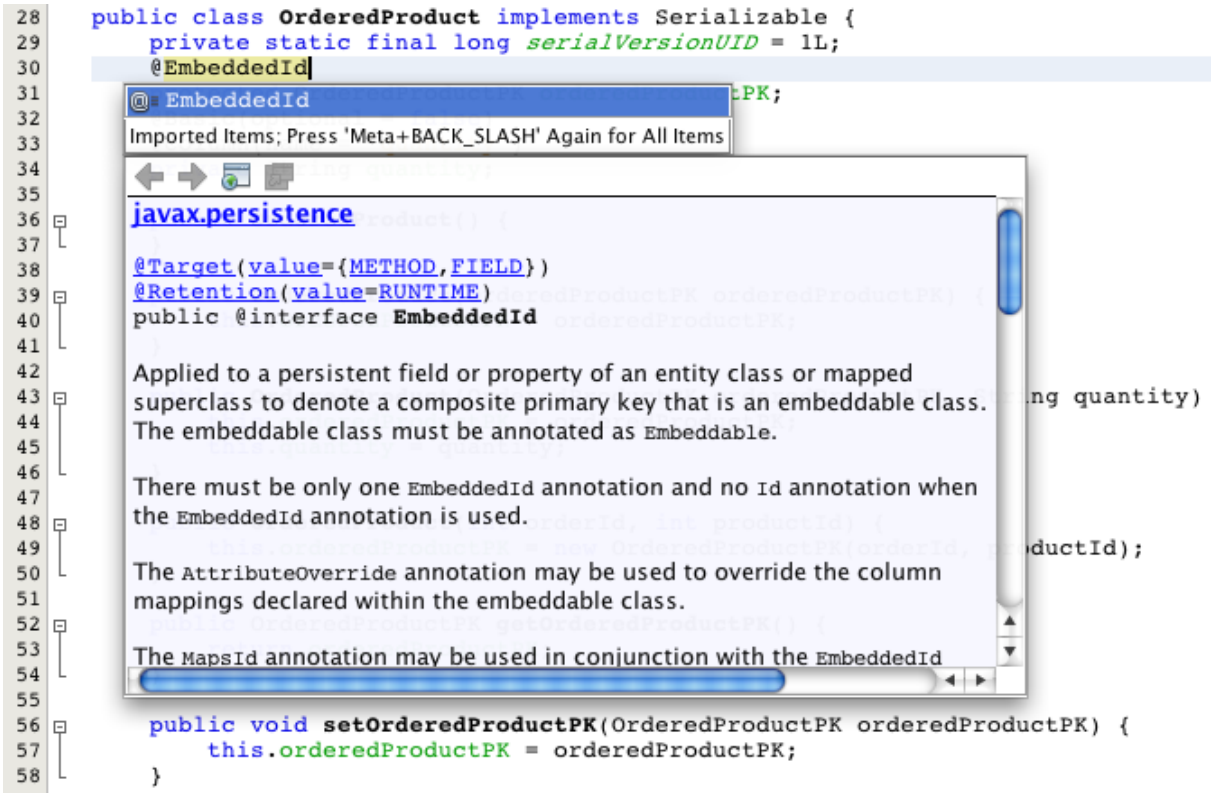
Click Finish. The JPA entity classes are generated, based on the `affablebean` database tables. You can examine the entity classes in the Projects window by expanding the newly created `entity` package. Also, note that the new persistence unit exists under the Configuration Files node.



Note that the wizard generated an additional entity class, `OrderedProductPK`. Recall that the data model's `ordered_product` table uses a composite primary key that comprises the primary keys of both the `customer_order` and `product` tables. (See Designing the Data Model - Creating Many-To-Many Relationships.) Because of this, the persistence provider creates a separate entity class for the composite key, and *embeds* it into the `OrderedProduct` entity. You can open `OrderedProduct` in the editor to inspect it. JPA uses the `@EmbeddedId` annotation to signify that the embeddable class is a composite primary key.

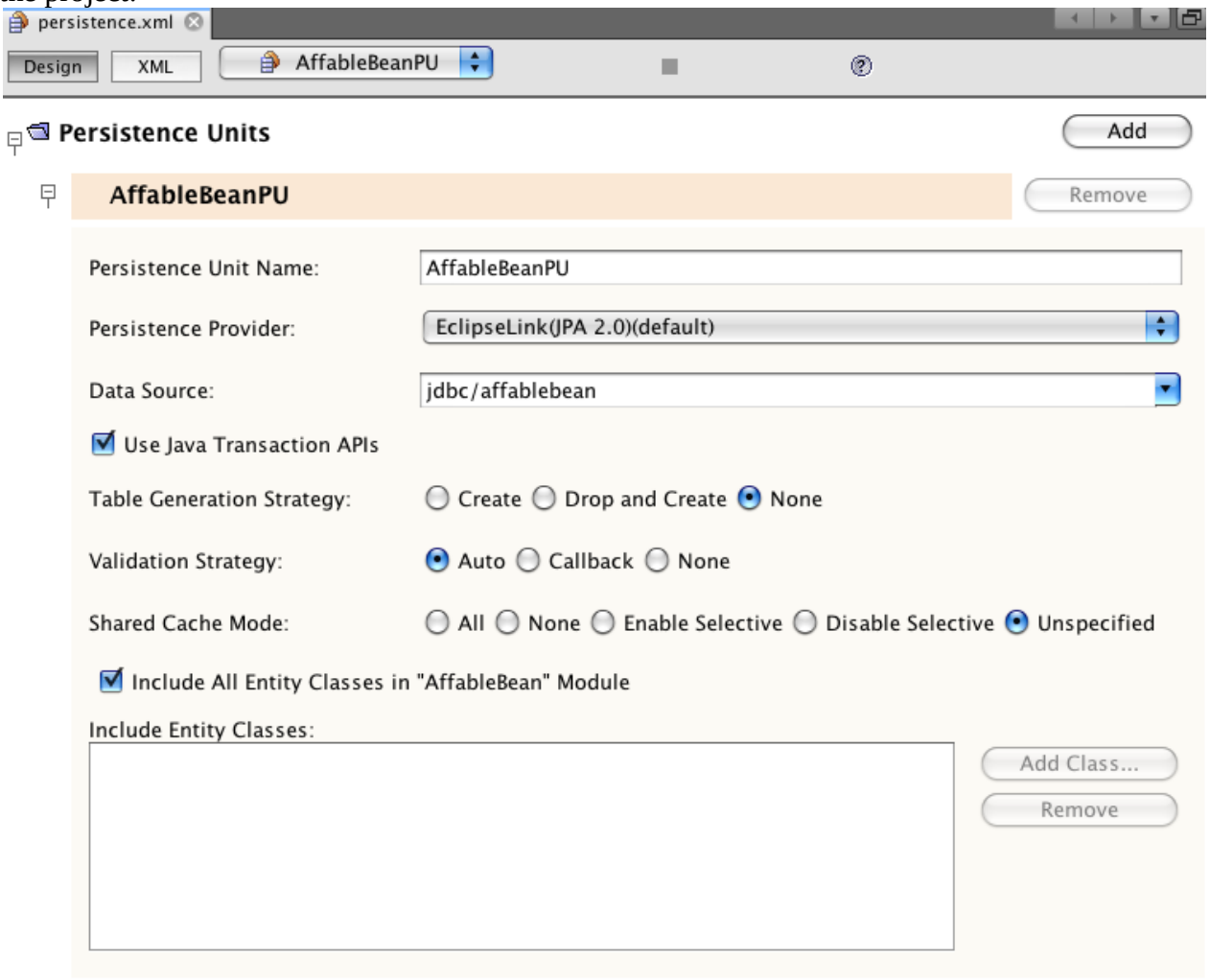
```
public class OrderedProduct implements Serializable {
    private static final long serialVersionUID = 1L;
    @EmbeddedId
    protected OrderedProductPK orderedProductPK;
```

Press Ctrl-Space on the `@EmbeddedId` annotation to invoke the API documentation.



Open the persistence unit (`persistence.xml`) in the editor. The IDE provides a Design view for persistence units, in addition to the XML view. The Design view provides a convenient way to make configuration changes to the persistence provider's management of

the project.



Click the XML tab at the top of the `AffableBeanPU` persistence unit to open the XML view. Add the following property to the file.

```
  <persistence-unit name="AffableBeanPU" transaction-type="JTA">
    <jta-data-source>jdbc/affablebean</jta-data-source>
    <properties>
      <property name="eclipselink.logging.level" value="FINEST"/>
    </properties>
</persistence-unit>
```

You set the logging level property to `FINEST` so that you can view all possible output produced by the persistence provider when the application runs. This enables you to see the SQL that the persistence provider is using on the database, and can facilitate in any required debugging.

See the official EclipseLink documentation for an explanation of logging and a list of all logging values: How To Configure Logging

Adding Session Beans

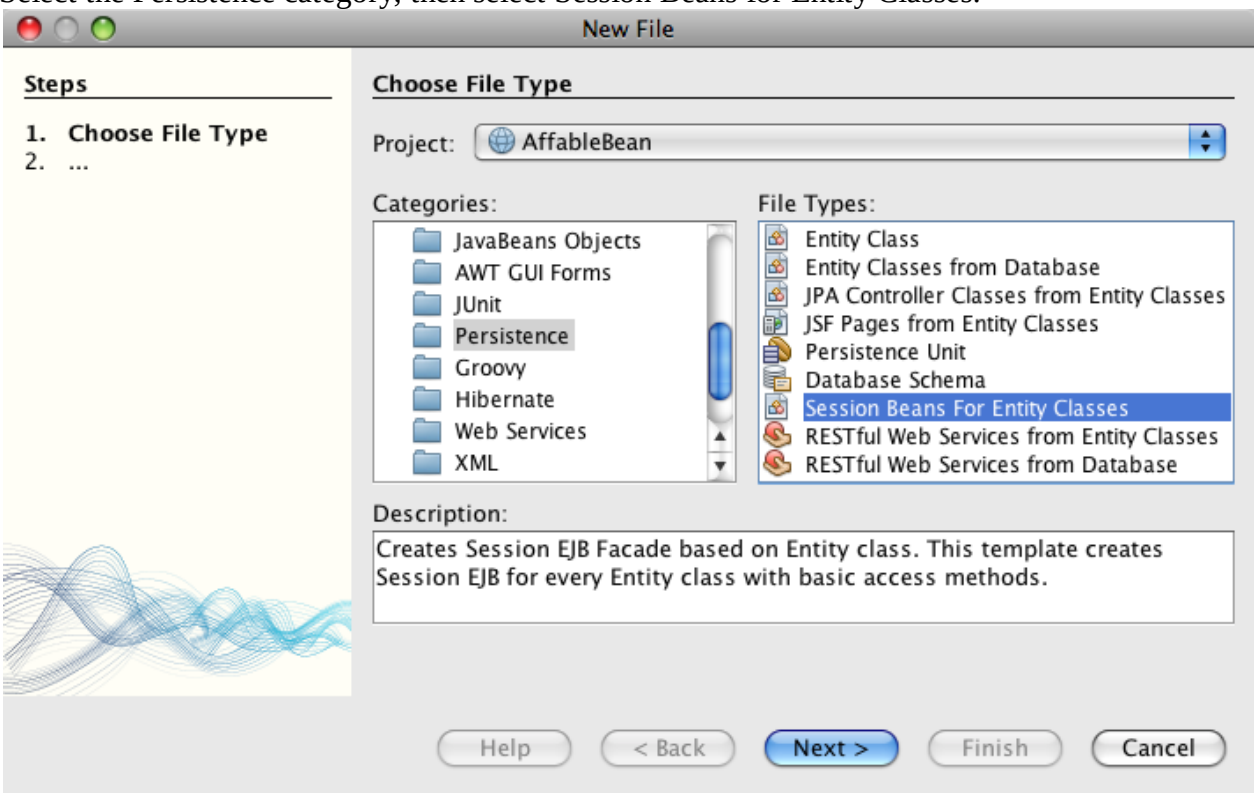
In this section, we use the IDE's Session Beans for Entity Classes wizard to generate an EJB *session facade* for each of the entity classes that you just created. Each session bean will contain basic access methods for its respective entity class.

A *session facade* is a design pattern advertised in the Enterprise BluePrints program. As stated in the Core J2EE Pattern Catalog, it attempts to resolve common problems that arise in a multi-tiered application environment, such as:

- Tight coupling, which leads to direct dependence between clients and business objects
- Too many method invocations between client and server, leading to network performance problems
- Lack of a uniform client access strategy, exposing business objects to misuse

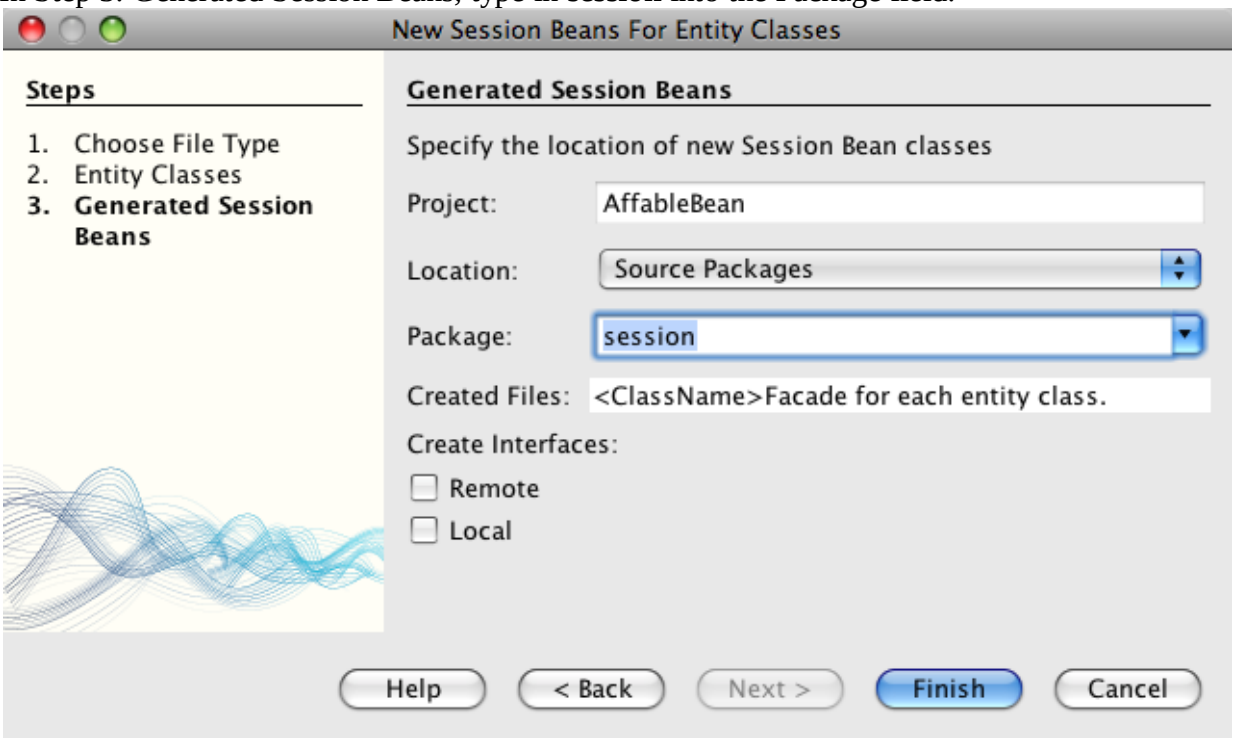
A session facade abstracts the underlying business object interactions and provides a service layer that exposes only the required functionality. Thus, it hides from the client's view the complex interactions between the participants. Thus, the session bean (representing the session facade) manages the relationships between business objects. The session bean also manages the life cycle of these participants by creating, locating, modifying, and deleting them as required by the workflow.

1. Press Ctrl-N (⌘-N on Mac) to open the File wizard.
2. Select the Persistence category, then select Session Beans for Entity Classes.



3. Click Next.
4. In Step 2: Entity Classes, note that all entity classes contained in your project are listed on the left, under Available Entity Classes. Click Add All. All entity classes are moved to the right, under Selected Entity Classes.
5. Click Next.

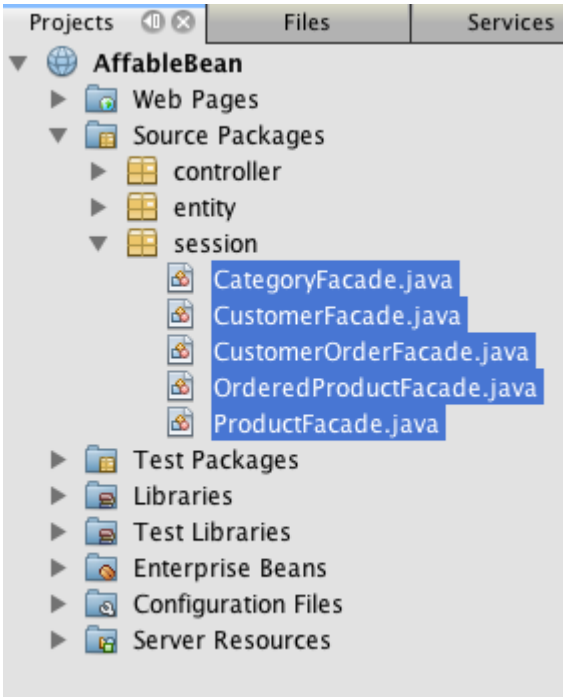
6. In Step 3: Generated Session Beans, type in **session** into the Package field.

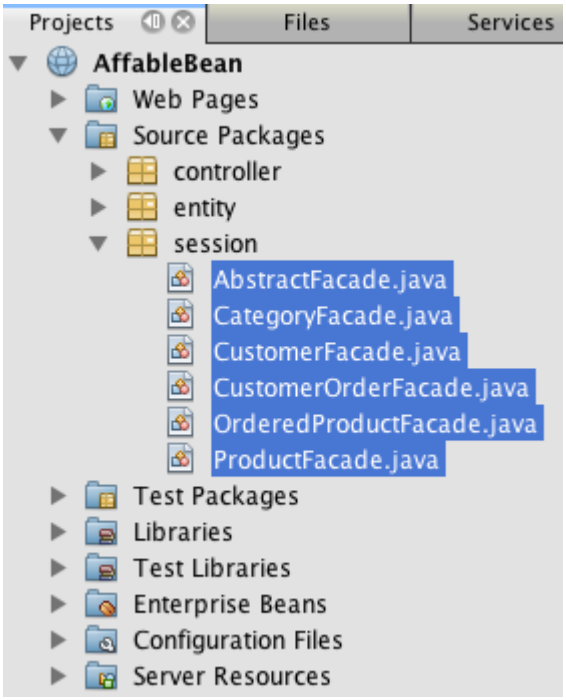


Note: You can use the wizard to generate local and remote interfaces for the session beans. While there is benefit to programming session beans to interfaces (For example, hiding business object interactions behind an interface enables you to further decouple the client from your business logic. This also means that you can code multiple implementations of the interface for your application, should the need arise.), this lies outside the scope of the tutorial. Note that EJB versions prior to 3.1 *require* that you implement an interface for each session bean.

7. Click Finish. The IDE generates session beans for each of the entity classes contained in your project. In the Projects window, expand the new `session` package to examine the session beans.

NetBeans 6.8 **NetBeans 6.9**





8. **Note:** As shown above, NetBeans IDE 6.9 provides slight improvements in the way the Session Beans for Entity Classes wizard generates facade classes. Namely, boiler-plate code that is common to all classes is factored out into an abstract class named `AbstractFacade`. If you are working in version 6.9, open any of the facade classes that have been generated (aside from `AbstractFacade`). You'll see that the class extends `AbstractFacade`.

Your application now contains a persistence model of the `affablebean` database in the form of JPA entity classes. It also contains a session facade consisting of Enterprise beans that can be used to access the entity classes. The next section demonstrates how you can access the database using the session beans and entity classes.

Accessing Data with EJBs

In the previous tutorial unit, you learned how to access the database from the application by configuring a data source on GlassFish, adding a resource reference to the application's deployment descriptor, and using JSTL `<sql>` tags in the application's JSP pages. This is a valuable technique, as it allows you to quickly set up prototypes that include data from the database. However, this is not a realistic scenario for medium to large-sized applications, or applications managed by a team of developers, as it would prove difficult to maintain or scale. Furthermore, if you are developing the application into multiple tiers or are adhering to the MVC pattern, you would not want to keep data-access code in your front-end. Using Enterprise beans with a persistence model enables you better conform to the MVC pattern by effectively decoupling the presentation and model components.

The following instructions demonstrate how to begin using the session and entity beans in the `AffableBean` project. You are going to remove the JSTL data access logic that you previously set up for the index and category pages. In its place, you'll utilize the data access methods provided by the session beans, and store the data in scoped variables so that it can be retrieved from front-end page views. We'll tackle the index page first, then move on to the more complicated category page.

index page

The index page requires data for the four product categories. In our current setup, the JSTL `<sql>` tags query the database for category details each time the index page is requested. Since this information is rarely modified, it makes more sense from a performance standpoint to perform the query only once after the application has been deployed, and store the data in an application-scoped variable. We can accomplish this by adding this code to the `ControllerServlet`'s `init` method.

1. In the Projects window, double-click the Source Packages > `controller` > `ControllerServlet` node to open it in the editor.
2. Declare an instance of `CategoryFacade`, and apply the `@EJB` annotation to the instance.

```
3. public class ControllerServlet extends HttpServlet {
4.
5.     @EJB
6.     private CategoryFacade categoryFacade;
7.
8.     ...
   }
```

   The `@EJB` annotation instructs the EJB container to instantiate the `categoryFacade` variable with the EJB named `CategoryFacade`.

9. Use the IDE's hints to add import statements for:
   - `javax.ejb.EJB`
   - `session.CategoryFacade`

   Pressing Ctrl-Shift-I (⌘-Shift-I on Mac) automatically adds required imports to your class.

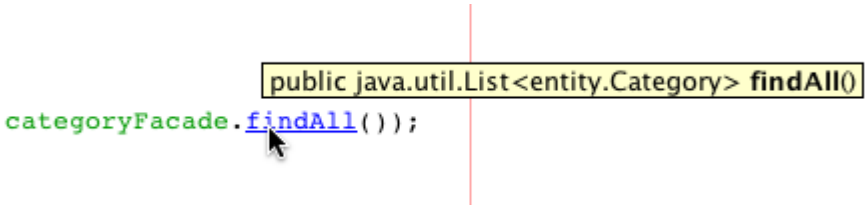
10. Add the following `init` method to the class. The web container initializes the servlet by calling its `init` method. This occurs only once, after the servlet is loaded and before it begins servicing requests.

```
11. public class ControllerServlet extends HttpServlet {
12.
13.     @EJB
14.     private CategoryFacade categoryFacade;
15.
16.     public void init() throws ServletException {
17.
18.         // store category list in servlet context
```

```
19.          getServletContext().setAttribute("categories",
   categoryFacade.findAll());
20.      }
21.
22.      ...
   }
```

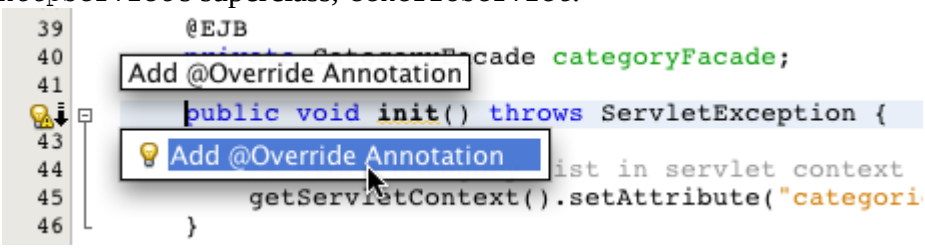
Here, you apply the facade class' `findAll` method to query the database for all records of `Category`. You then set the resulting `List` of `Category` objects as an attribute that can be referenced by the "`categories`" string. Placing the reference in the `ServletContext` means that the reference exists in a scope that is application-wide.

To quickly determine the method signature of the `findAll` method, hover your mouse over the method while holding down the Ctrl key (⌘ on Mac). (The image below displays the popup that appears using NetBeans IDE 6.8.)



Clicking the hyperlink enables you to navigate directly to the method.

23. Use the IDE's hint to add the `@Overrides` annotation. The `init` method is defined by `HttpServlet`'s superclass, `GenericServlet`.



Adding the annotation is not required, however it does provide several advantages:

- It enables you to use compiler checking to ensure that you are actually overriding a method that you assume you are overriding.
- It improves readability, as it becomes clear when methods in your source code are being overridden.

For more information on annotations, see the Java Tutorials: Annotations.

24. Now that you have set up an application-scoped attribute that contains a list of categories, modify the index page to access the newly created attribute.

Double-click the Web Pages > `index.jsp` node in the Projects window to open the file in the editor.

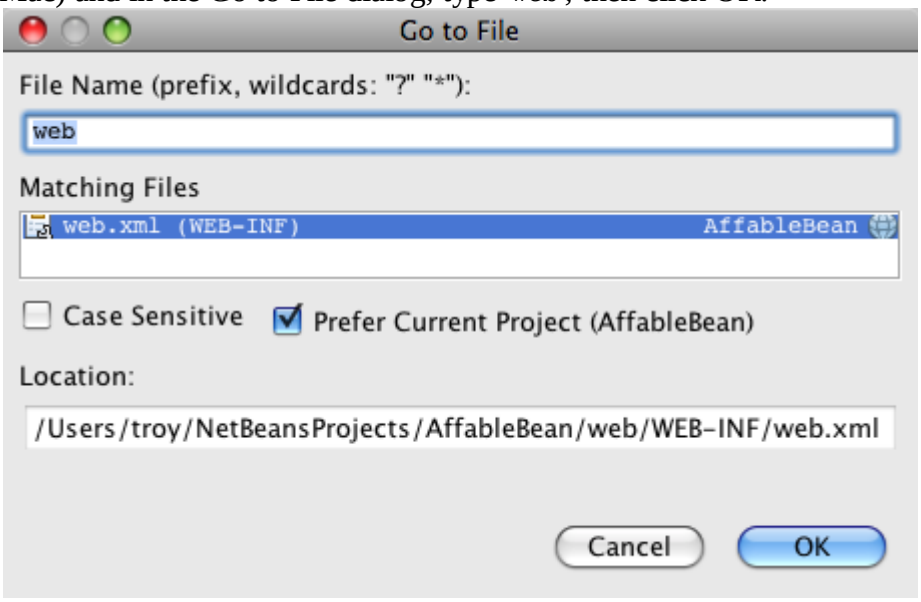
25. Comment out (or delete) the `<sql:query>` statement that is listed at the top of the file. To comment out code in the editor, highlight the code, then press Ctrl-/ (⌘-/ on Mac).

```
7   <%--<sql:query var="categories" dataSource="jdbc/affablebean">
8       SELECT * FROM category
9   </sql:query>--%>
10
```

26. Modify the opening `<c:forEach>` tag so that its `items` attribute references the new application-scoped `categories` attribute.

```
<c:forEach var="category" items="${categories}">
```

27. Open the project's web deployment descriptor. Press Alt-Shift-O (Ctrl-Shift-O on Mac) and in the Go to File dialog, type '`web`', then click OK.



28. Comment out (or delete) the `<resource-ref>` entry. The entry was required for the `<sql>` tags in order to identify the data source registered on the server. We are now relying on JPA to access the database, and the `jdbc/affablebean` data source has already been specified in the persistence unit. (Refer to the Design view of the project's persistence unit above.)

    Highlight the entire `<resource-ref>` entry, then press Ctrl-/ (⌘-/ on Mac).

```
29.  <!-- <resource-ref>
30.          <description>Connects to database for AffableBean
     application</description>
31.          <res-ref-name>jdbc/affablebean</res-ref-name>
32.          <res-type>javax.sql.DataSource</res-type>
33.          <res-auth>Container</res-auth>
34.          <res-sharing-scope>Shareable</res-sharing-scope>
       </resource-ref> -->
```

35. Run the project. Click the Run Project ( ) button. The project's index page opens in the browser, and you see that all four category names and images display.

category page

The category page requires three pieces of data in order to render properly:

1. **category data:** for left column category buttons
2. **selected category:** the selected category is highlighted in the left column, and the name of the selected category displays above the product table
3. **product data for selected category:** for products displayed in the product table

Let's approach each of the three pieces of data individually.

- category data
- selected category
- product data for selected category

category data

To account for category data, we can reuse the application-scoped `categories` attribute that we created for the index page.

1. Open `category.jsp` in the editor, and comment out (Ctrl-/; ⌘-/ on Mac) the JSTL `<sql>` statements that are listed at the top of the file.

```
<%--<sql:query var="categories" dataSource="jdbc/affablebean">
    SELECT * FROM category
</sql:query>

<sql:query var="selectedCategory" dataSource="jdbc/affablebean">
    SELECT name FROM category WHERE id = ?
    <sql:param value="${pageContext.request.queryString}"/>
</sql:query>

<sql:query var="categoryProducts" dataSource="jdbc/affablebean">
    SELECT * FROM product WHERE category_id = ?
    <sql:param value="${pageContext.request.queryString}"/>
</sql:query>--%>
```

2. Modify the opening `<c:forEach>` tag so that its `items` attribute references the application-scoped `categories` attribute. (This is identical to what you did above for `index.jsp`.)

```
<c:forEach var="category" items="${categories}">
```

3. Run the project to examine the current state of the category page. Click the Run Project ( ) button. When the project's index page opens in the browser, click any of the four categories. The category buttons in the left column display and function as expected.

selected category

To retrieve the selected category, we can use the `categoryFacade` that we already created to find the `Category` whose ID matches the request query string.

1. Open the `ControllerServlet` in the editor. (If already opened, press Ctrl-Tab and choose from the pop-up list.)
2. Start implementing functionality to acquire the selected category. Locate the `TODO: Implement category request` comment, delete it and add the following code (in **bold**).

```
// if category page is requested
if (userPath.equals("/category")) {

    // get categoryId from request
    String categoryId = request.getQueryString();

    if (categoryId != null) {

    }

// if cart page is requested
} else if (userPath.equals("/viewCart")) {
```

You retrieve the requested category ID by calling `getQueryString()` on the request.

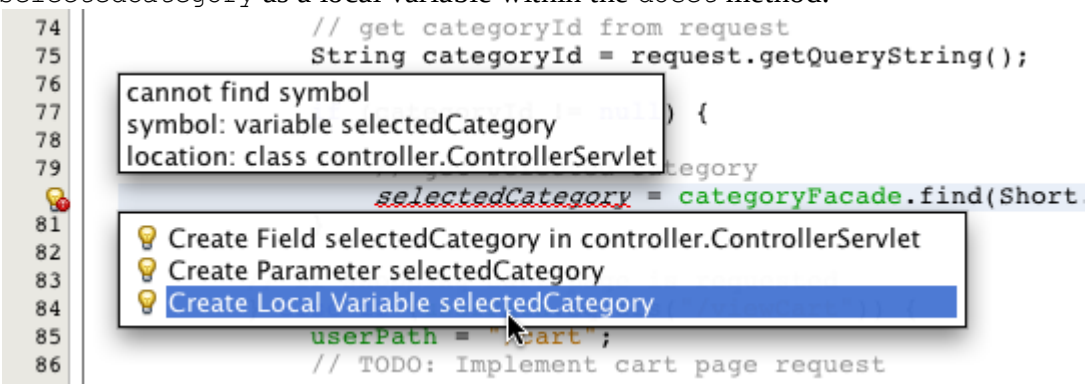
Note: The logic to determine the selected category within the left column category buttons is already implemented in `category.jsp` using an EL expression, which is comparable to calling `getQueryString()` in the servlet. The EL expression is: `pageContext.request.queryString`.

14. Add the following line of code within the `if` statement.

```
// get categoryId from request
String categoryId = request.getQueryString();

if (categoryId != null) {

    // get selected category
    selectedCategory =
categoryFacade.find(Short.parseShort(categoryId));
}
```

You use the `CategoryFacade`'s `find` method to retrieve the `Category` object based on the requested category ID. Note that you must cast `categoryId` to a `Short`, as this is the type used for the `id` field in the `Category` entity class.

22. Click the badge ( ) in the left margin to use the editor's hint to declare `selectedCategory` as a local variable within the `doGet` method.



23. Add the following line to place the retrieved `Category` object in the request scope.

```
24.  // get categoryId from request
25.  String categoryId = request.getQueryString();
26.
27.  if (categoryId != null) {
28.
29.      // get selected category
30.      selectedCategory =
   categoryFacade.find(Short.parseShort(categoryId));
31.
32.      // place selected category in request scope
33.      request.setAttribute("selectedCategory", selectedCategory);
   }
```

34. In the editor, switch to `category.jsp`. (Press Ctrl-Tab and choose from the pop-up list.)
35. Locate `<p id="categoryTitle">` and make the following change.

```
36.  <p id="categoryTitle">
37.      <span style="background-color: #f5eabe; padding:
   7px;">${selectedCategory.name}</span>
   </p>
```

You are now using the `selectedCategory` attribute, which you just placed in the request scope from the `ControllerServlet`. Using '`.name`' within the EL expression calls the `getName` method on the given `Category` object.

38. Run the project to examine the current state of the category page. Click the Run Project ( ) button. When the project's index page opens in the browser, click any of the four categories. The name of the selected category now displays in the page.

product data for selected category

In order to retrieve all products for a selected category, we'll make use of the `ProductFacade` EJB. Unfortunately, the class currently does not provide a method that allows us to search for products based on category ID. Start by creating a method that performs this action. Then, call the method in the `ControllerServlet`, place the retrieved list of products in a scoped variable, and finally retrieve the list from the category JSP page.

1. Open `ProductFacade` in the editor and, depending on your IDE version, add the following method to the class.

   NetBeans 6.8

   ```
   public List<Product> findForCategory(Category category) {
       return em.createQuery("SELECT p FROM Product p WHERE p.categoryId
   = :categoryId").
               setParameter("categoryId", category).getResultList();
   }
   ```

   NetBeans 6.9

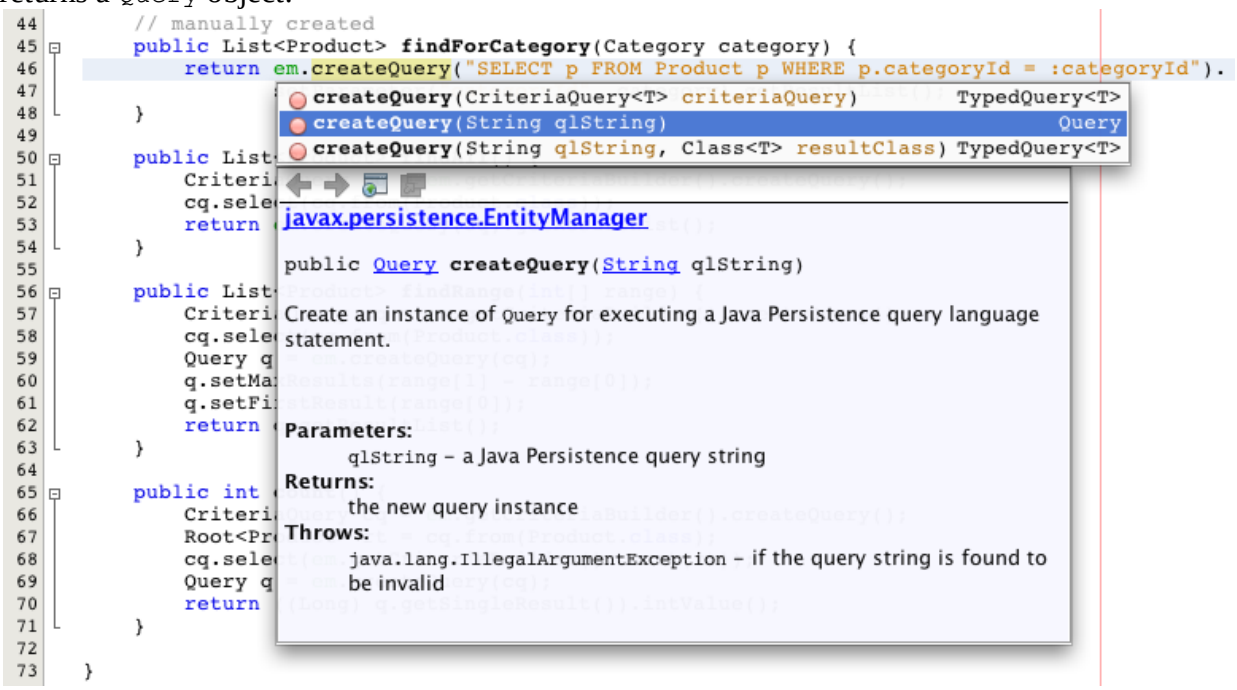
   ```
   public List<Product> findForCategory(Category category) {
       return em.createQuery("SELECT p FROM Product p WHERE p.category =
   :category").
               setParameter("category", category).getResultList();
   }
   ```

   `ProductFacade`, like all of the generated facade classes, instantiates an `EntityManager`. The `EntityManager` is an integral component of the Java Persistence API, and is responsible for performing persistence actions on the database. The book EJB 3 In Action describes the `EntityManager` as follows:

   The JPA `EntityManager` interface manages entities in terms of actually providing persistence services. While entities tell a JPA provider how they map to the database, they do not persist themselves. The `EntityManager` interface reads the ORM metadata for an entity and performs persistence operations.

   In the above example, the `EntityManager`'s `createQuery` method is called to perform a query on the database. If you examine the `createQuery` API documentation (press Ctrl-Space on the method in the editor), you see that the method takes a Java Persistence Query Language (JPQL) query string as an argument, and

returns a `Query` object.



For more information on the JPQL, including terminology, syntax, and example queries, see the Java EE 6 Tutorial, Chapter 21: The Java Persistence Query Language.

The `Query` object in turn calls `setParameter` to bind the `categoryId` parameter used in the query string with the the `Category` object that is passed into the method. Finally, `getResultList()` is called to execute a `SELECT` query and return the query results as a `List` of `Product`s.

2. Press Ctrl-Shift-I (⌘-Shift-I on Mac) to fix imports. The Fix All Imports dialog opens to enable you to add an import for `entity.Category` (as well as `java.util.List` if you are using NetBeans 6.9). Click OK.
3. Open the `ControllerServlet` in the editor (Use Ctrl-Tab if the class is already opened, otherwise press Alt-Shift-O (Ctrl-Shift-O on Mac) and use the Go to File dialog.
4. Declare an instance of `ProductFacade`, and apply the `@EJB` annotation to the instance.

```
public class ControllerServlet extends HttpServlet {

    @EJB
    private CategoryFacade categoryFacade;
    @EJB
     private ProductFacade productFacade;

     ...
}
```

The `@EJB` annotation instructs the EJB container to instantiate the `productFacade` variable with the EJB named `ProductFacade`.

13. Use the IDE's hint to add an import statement for `session.ProductFacade`.

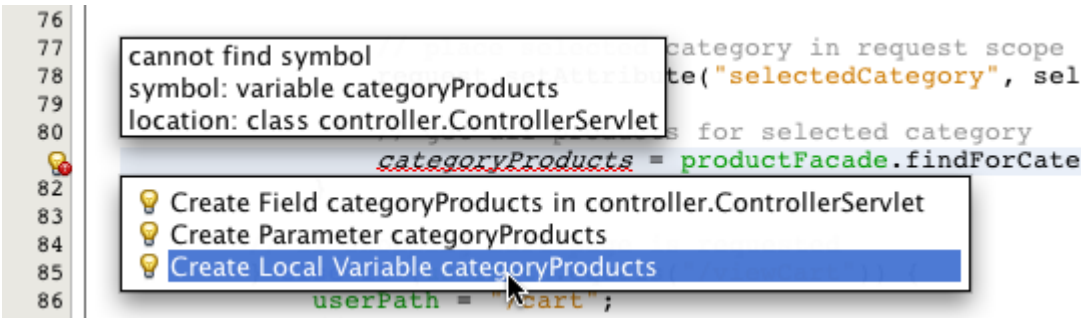
Pressing Ctrl-Shift-I (⌘-Shift-I on Mac) automatically adds required import to your class.

14. Add the following statement to the code that manages the category request.

```
15.  // if category page is requested
16.  if (userPath.equals("/category")) {
17.
18.      // get categoryId from request
19.      String categoryId = request.getQueryString();
20.
21.      if (categoryId != null) {
22.
23.          // get selected category
24.          selectedCategory =
   categoryFacade.find(Short.parseShort(categoryId));
25.
26.          // place selected category in request scope
27.          request.setAttribute("selectedCategory", selectedCategory);
28.
29.          // get all products for selected category
30.          categoryProducts =
   productFacade.findForCategory(selectedCategory);
      }
```

You use the `productFacade` to call the newly created `findForCategory` method. Because the method requires a `Category` object as an argument, you pass the previously created `selectedCategory` into the method.

31. Use the editor's hint to define `categoryProducts` as a local variable for the `doGet` method.



32. Place the list of `Products` in the request scope, so that it can be retrieved from the application's front-end.

```
33.  // if category page is requested
34.  if (userPath.equals("/category")) {
35.
36.      // get categoryId from request
37.      String categoryId = request.getQueryString();
38.
39.      if (categoryId != null) {
40.
41.          // get selected category
42.          selectedCategory =
   categoryFacade.find(Short.parseShort(categoryId));
43.
44.          // place selected category in request scope
45.          request.setAttribute("selectedCategory", selectedCategory);
46.
47.          // get all products for selected category
```

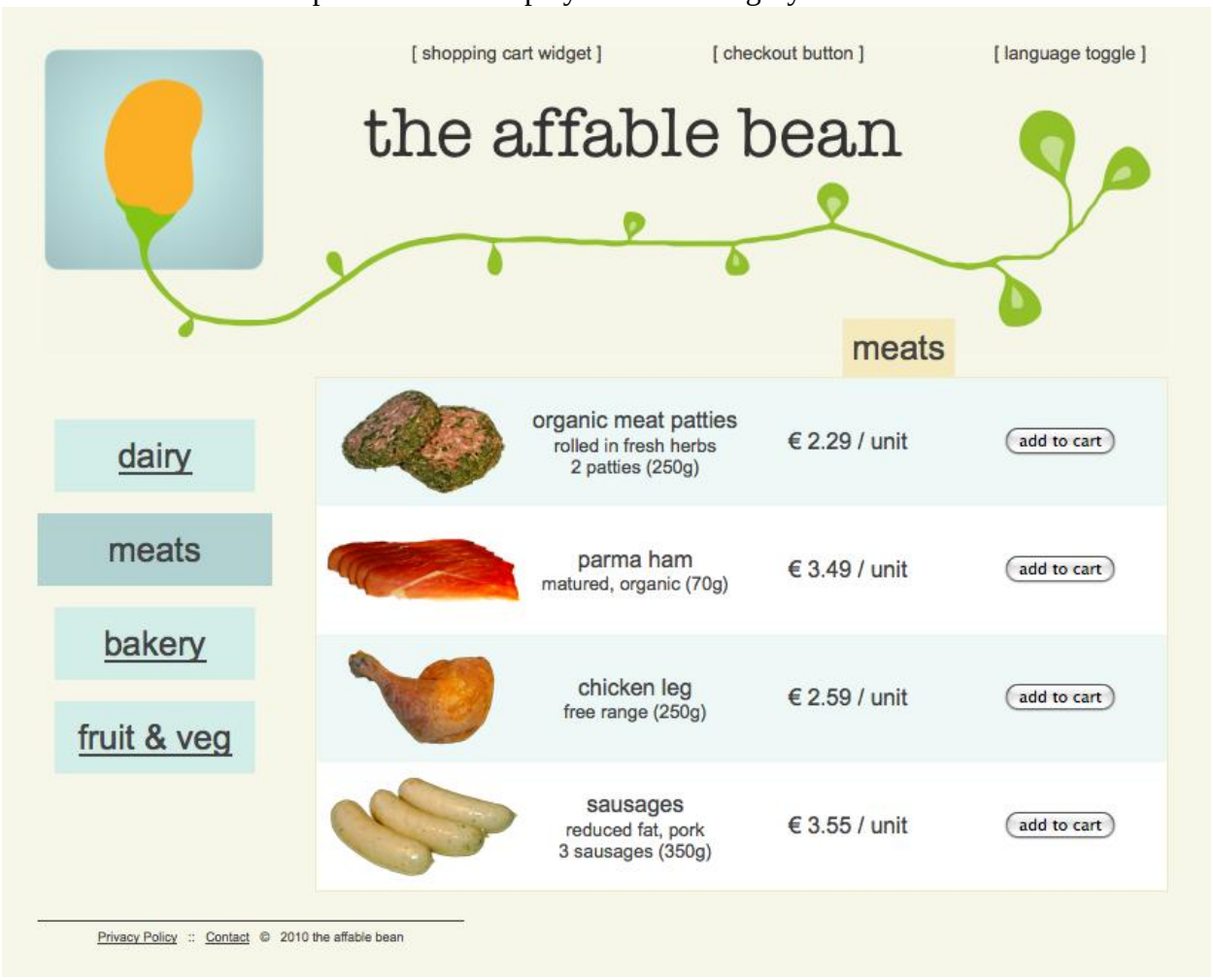
```
48.          categoryProducts =
   productFacade.findForCategory(selectedCategory);
49.
50.          // place category products in request scope
51.          request.setAttribute("categoryProducts", categoryProducts);
       }
```

52. Open the `category.jsp` file in the editor and make the following change to the product table.

```
53.  <table id="productTable">
54.
       <c:forEach var="product" items="${categoryProducts}"
   varStatus="iter">
```

The `<c:forEach>` tag now references the `categoryProducts` list. The `c:forEach` loop will now iterate over each `Product` object contained in the list, and extract data accordingly.

55. Press F6 (fn-F6 on Mac) to run the project. Navigate to the category page in the browser and note that all products now display for each category.



This tutorial unit provided a brief introduction to JPA and EJB technologies. It also described the role of Java specifications, and how their reference implementations are used by the GlassFish application server. It then demonstrated how to create a set of JPA entity classes that provide a Java implementation of the project database. Then, following the *session*

facade pattern, it showed how to create a set of EJB session beans that exist on top of the entity classes and enable convenient access to them. Finally, you modified the `AffableBean` project to utilize the new session beans and entities for database access required in the index and category pages.

You can download snapshot 4 of the `AffableBean` project, which corresponds to state the project after completing this unit using NetBeans IDE 6.9.

In the next unit you explore session management, and how to enable the application to remember a user's actions as he or she clicks through the site. This is key to implementing a shopping cart mechanism in an e-commerce application.

The NetBeans E-commerce Tutorial - Managing Sessions

Tutorial Contents

Every e-commerce application that offers some form of shopping cart functionality needs to be able to remember user-specific data as users click through the website. Unfortunately for you the developer, the HTTP protocol, over which communication on the Internet takes

place, is a *stateless* protocol. Each request received by your server is an independent piece of information that has no relation to previously received requests. Therefore, if a customer clicks a button to add an item to his or her shopping cart, your application must take measures to ensure not only that the state of the user's cart is updated, but that the action doesn't affect the cart of another user who happens to be browsing the site at the same time.

In order to properly handle the above-described scenario, you need to implement functionality so that a *session* can be created and maintained for the duration of a user's visit to the site. Servlet technology, which is the foundation of all Java-based web applications, provides for this with its `HttpSession` interface. You also need to define several classes, namely `ShoppingCart` and `ShoppingCartItem`, that allow the application to temporarily store user data while the session is being maintained.

This tutorial unit takes a different approach from others in the NetBeans E-commerce Tutorial. Instead of having you create project files and providing steps with code snippets for you to copy and paste into your own project, you open the completed project snapshot for this unit, and examine the code using the IDE's debugger and other tools. In the process, you'll learn how to apply an `HttpSession` object to your code so that each visit to the website results in a dedicated session. You also learn about *scoped variables*, and their usage in both Java classes and JSP pages. This unit also discusses `HttpSession`'s default mechanism for maintaining sessions (i.e., cookies) and shows what steps need to be taken in the event that cookies are deactivated in a user's browser. Finally, session time-outs are covered, and the unit demonstrates how to handle them by creating a simple filter that intercepts requests to check whether a session exists.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
GlassFish server	v3 or Open Source Edition 3.0.1
AffableBean project	snapshot 5

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.
- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could download the GlassFish server independently, but the version provided

with the NetBeans download has the added benefit of being automatically registered with the IDE.
- You can follow this tutorial unit without having completed previous units. To do so, see the setup instructions, which describe how to prepare the database and establish connectivity between the IDE, GlassFish, and MySQL.

Handling Session Data

Applications can manage user sessions with the `HttpSession` object. You can bind user-specific data to the `HttpSession` object, then access this data at a later stage. Both bind and access actions can be done from Java classes, as well as from session-scoped variables in EL expressions.

- Working with an HttpSession Object
- Working with Scoped Variables in Web Applications

Working with an HttpSession Object

The `AffableBean` application uses the `HttpSession` object to identify users over multiple requests. An `HttpSession` object is obtained using `getSession()` on a given request:

```
HttpSession session = request.getSession();
```

If a session object doesn't yet exist for the request, the method creates and returns a new session object.

You can use the session object as a vehicle for passing data between requests. You use the `setAttribute` method to bind objects to the session. Likewise, you use `getAttribute` to retrieve objects from the session. In the `AffableBean` application for example, the user's shopping cart is created and bound to the user session in the following manner:

```
ShoppingCart cart = new ShoppingCart();
session.setAttribute("cart", cart);
```

In order to retrieve the cart from the session, the `getAttribute` method is applied:

```
cart = (ShoppingCart) session.getAttribute("cart");
```

In JSP pages, you can access objects bound to the session using EL expressions. Continuing with the above example, if a `ShoppingCart` object named '`cart`' is bound to the session, you can access the object using the following EL expression:

```
${cart}
```

Accessing the `ShoppingCart` object on its own is of little value however. What you really want is a way to access values stored in the object. If you explore the new `ShoppingCart` class in the project snapshot, you'll note that it contains the following properties:

- `double total`
- `int numberOfItems`
- `List<String, ShoppingCartItem> items`

Provided that properties have matching getter methods, you can access values for singular properties using simple dot notation in an EL expression. If you examine the `cart.jsp` page, you'll see that this is exactly how the value for `numberOfItems` is accessed:

```
<p>Your shopping cart contains ${cart.numberOfItems} items.</p>
```

In order to extract data from properties that contain multiple values, such as the above `items` list, the `cart.jsp` page uses a `<c:forEach>` loop:

```
<c:forEach var="cartItem" items="${cart.items}" varStatus="iter">

  <c:set var="product" value="${cartItem.product}"/>

    <tr class="${((iter.index % 2) == 0) ? 'lightBlue' : 'white'}">
        <td>
            <img src="${initParam.productImagePath}${product.name}.png"
                alt="image of ${product.name}">
        </td>

        <td>${product.name}</td>

        <td>
            &euro; ${cartItem.total}
            <br>
            <span class="smallText">( &euro; ${product.price} / unit
)</span>
        </td>
        ...
    </tr>

</c:forEach>
```

`ShoppingCartItem`'s `product` property identifies the product type for a cart item. The above loop takes advantage of this by first setting a `product` variable to the expression `${cartItem.product}`. It then uses the variable to obtain information about that product (e.g., name, price).

Working with Scoped Variables in Web Applications

When working with JSP/Servlet technology, there are four scope objects available to you within the realm of the application. JSP technology implements *implicit objects* that allows you to access classes defined by the Servlet API.

Scope	Definition	Servlet Class	JSP Implicit Object
Application	Global memory for a web application	javax.servlet.ServletContext	applicationScope
Session	Data specific to a user session	javax.servlet.http.HttpSession	sessionScope

Request	Data specific to an individual server request	`javax.servlet.HttpServletRequest`	`requestScope`
Page	Data that is only valid in the context of a single page (JSPs only)	`[n/a]`	`pageScope`

If you open your project's `category.jsp` file in the editor, you'll see that EL expressions include various scoped variables, including `${categories}`, `${selectedCategory}` and `${categoryProducts}`. The `${categories}` variable is application-scoped, which is set in the `ControllerServlet`'s `init` method:

```
// store category list in servlet context
getServletContext().setAttribute("categories", categoryFacade.findAll());
```

The other two, `${selectedCategory}` and `${categoryProducts}`, are placed in the application's session scope from the `ControllerServlet`. For example:

```
// place selected category in session scope
session.setAttribute("selectedCategory", selectedCategory);
```

Note: If you are continuing from the previous tutorial units, you'll likely note that `${selectedCategory}` and `${categoryProducts}` were originally placed in the request scope. In previous units this was fine, but consider now what happens if a user clicks the 'add to cart' button in a category page. The server responds to an `addToCart` request by returning the currently viewed category page. It therefore needs to know the `selectedCategory` and the `categoryProducts` pertaining to the selected category. Rather than establishing this information for each request, you place it in the session scope from a `category` request so that it is maintained across multiple requests, and can be accessed when you need it. Also, examine the functionality provided by the cart page. (A functional description is provided below.) The 'continue shopping' button returns the user to the previously viewed category. Again, the `selectedCategory` and the `categoryProducts` variables are required.

When referencing scoped variables in an EL expression, you do not need to specify the variable's scope (provided that you do not have two variables of the same name in different scopes). The JSP engine checks all four scopes and returns the first variable match it finds. In `category.jsp` for example, the expression:

```
${categoryProducts}
```

is shorthand for:

```
${sessionScope.categoryProducts}
```

For more information, see the following resources:

- Designing Enterprise Applications with the J2EE Platform: State Scopes
- Sharing Information > Using Scoped Objects
- Unified Expression Language > Implicit Objects

Examining Session Data with the Java Debugger

Begin exploring how the application behaves during runtime. Use the IDE's debugger to step through code and examine how the `HttpSession` is created, and how other objects can be placed in the session scope to be retrieved at a later point.

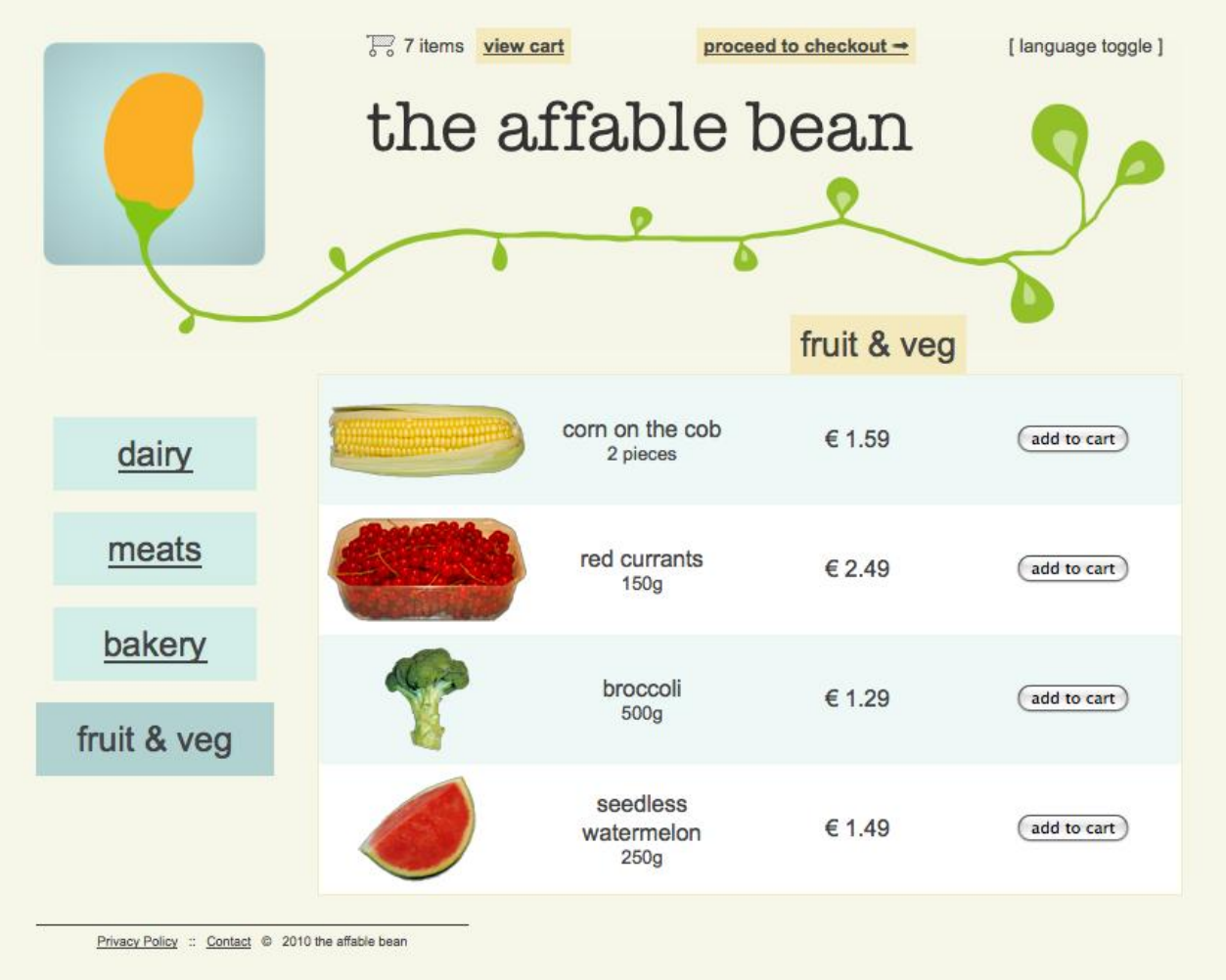
1. Open the project snapshot for this tutorial unit in the IDE. Click the Open Project () button and use the wizard to navigate to the location on your computer where you downloaded the project. If you are proceeding from the previous tutorial unit, note that this project snapshot includes a new `cart` package, containing `ShoppingCart` and `ShoppingCartItem` classes. Also, the following files have been modified:
   - `WEB-INF/web.xml`
   - `css/affablebean.css`
   - `WEB-INF/jspf/header.jspf`
   - `WEB-INF/jspf/footer.jspf`
   - `WEB-INF/view/cart.jsp`
   - `WEB-INF/view/category.jsp`
   - `WEB-INF/view/checkout.jsp`
   - `controller/ControllerServlet`
2. Run the project () to ensure that it is properly configured with your database and application server.

   If you receive an error when running the project, revisit the setup instructions, which describe how to prepare the database and establish connectivity between the IDE, GlassFish, and MySQL.

3. Test the application's functionality in your browser. If you are continuing directly from the previous tutorial unit, you'll note the following enhancements.

   category page

   - Clicking 'add to cart' for the first time enables the shopping cart and 'proceed to checkout' widgets to display in the header.
   - Clicking 'add to cart' results in an update to the number of cart items in the header's shopping cart widget.
   - Clicking 'view cart' results in the cart page displaying.
   - Clicking 'proceed to checkout' results in the checkout page displaying.



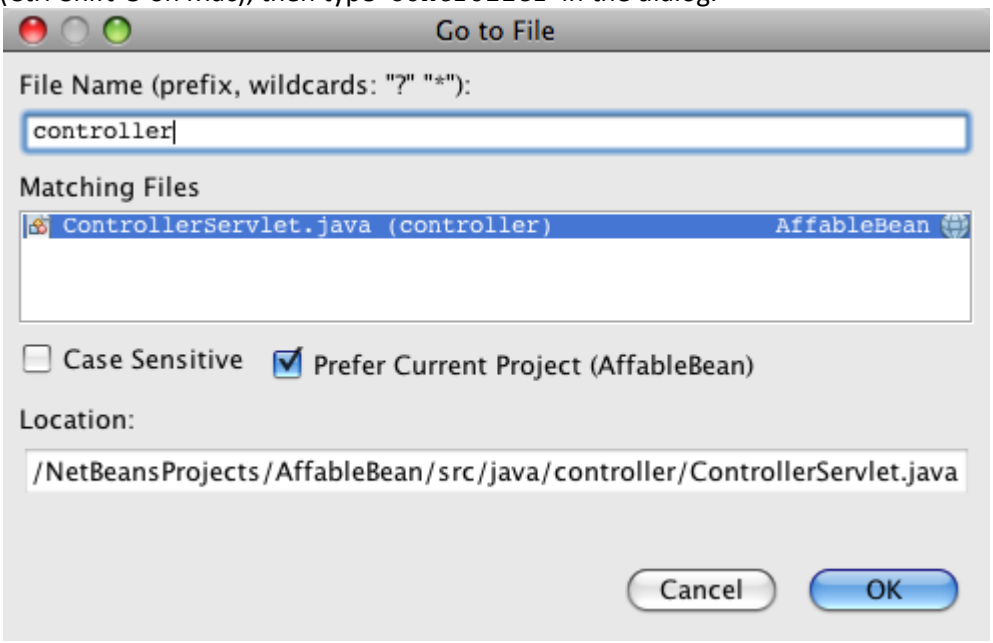
cart page

- Clicking 'clear cart' results in shopping cart being emptied of items.
- Clicking 'continue shopping' results in a return to the previously viewed category.
- Clicking 'proceed to checkout' results in the checkout page displaying.
- Entering a number (1 - 99) in an item's quantity field then clicking 'update' results in a recalculation of the total price for the item, and of the subtotal.
- Entering zero in an item's quantity field then clicking 'update' results in the item being removed from the displayed table.

checkout page

- Clicking 'view cart' results in the cart page displaying.
- Clicking 'submit' results in the confirmation page displaying (without user-specific data).

4. Use the Go to File dialog to open the `ControllerServlet` in the editor. Press Alt-Shift-O (Ctrl-Shift-O on Mac), then type '`Controller`' in the dialog.



5. Set a breakpoint in the `doPost` method on the line that creates an `HttpSession` object (line 150). To set a breakpoint, click in the left margin of the editor.

To toggle line numbers for the editor, right-click in the left margin and choose Show Line Numbers.

6. Run the debugger. Click the Debug Project () button in the IDE's main toolbar. The GlassFish server starts (or restarts, if it is already running) and opens a socket on its debug port number. The application welcome page opens in your browser.

   You can view and modify the debug port number from the Servers window (Tools > Servers). Select the Java tab for the server you are using. Specify the port number in the 'Address to use' field under Debug Settings.

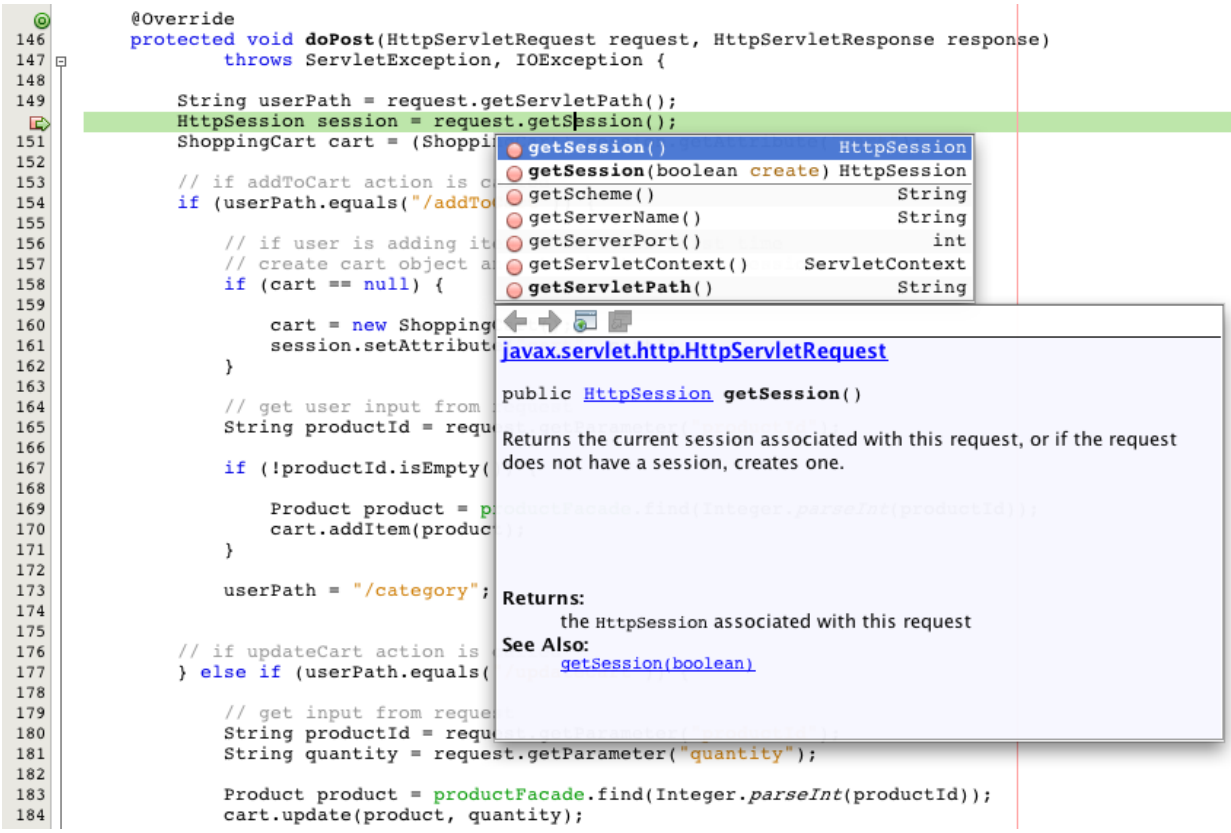
7. When the application's welcome page displays in the browser, click any category image to navigate to the category page. Recall that clicking the 'add to cart' button sends an `addToCart` request to the server:

   ```
   <form action="addToCart" method="post">
   ```

   As you may recall from Working with the Deployment Descriptor, the `ControllerServlet`'s `doPost` method handles requests for the `/addToCart` URL pattern. You can therefore expect that when a user clicks an 'add to cart' button, the `doPost` method is called.

8. Click 'add to cart' for any product in the category page. Switch back to the IDE and note that the debugger suspends on the breakpoint.

9. Place your cursor on the call to `getSession()` and press Ctrl-Space to invoke the Javadoc documentation.



According to the documentation, `getSession()` returns the `HttpSession` currently associated with the request, and if no session exists, the method creates a new session object.

Taking Advantage of the IDE's Javadoc Support

The IDE provides built-in Javadoc support for Java EE development. The IDE bundles with the Java EE 6 API Specification, which you can open in an external browser by choosing Help > Javadoc References > Java EE 6.

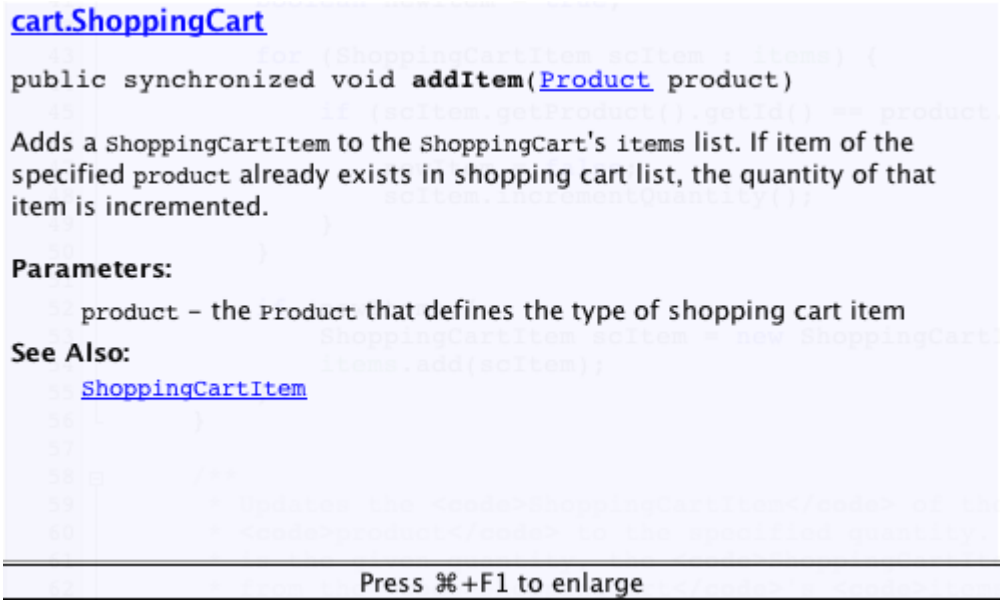
The IDE also includes various other features that enable easy access to API documentation:

- **Javadoc window:** Choose Window > Other > Javadoc. The Javadoc window opens in the bottom region of the IDE, and displays API documentation relevant to your cursor's location in the editor.
- **Javadoc Index Search:** Choose Help > Javadoc Index Search (Shift-F1; fn-Shift-F1 on Mac). Type in the name of the class you are looking for, then select a class from the listed results. The complete class description from the API Specification displays in the bottom pane of the window.
- **Documentation popup in the editor:** Javadoc documentation displays in a popup window when you press Ctrl-Space on a given element in the editor. You can click the External Browser ( ) button to have the documentation open in your browser. If you want to use Ctrl-Space for code completion only, you can deactivate the documentation popup by opening the Options window (Tools > Options; NetBeans > Preferences on Mac), then selecting Editor > Code Completion. Deselect the 'Auto Popup Documentation Window' option.

When you document your own work, consider adding Javadoc comments to your classes and methods. Open the `ShoppingCart` class and examine the Javadoc comments added to the class methods. Javadoc comments are marked by the `/** ... */` delimiters. For example, the `addItem` method has the following comment before its signature:

```
/**
 * Adds a <code>ShoppingCartItem</code> to the
<code>ShoppingCart</code>'s
 * <code>items</code> list. If item of the specified
<code>product</code>
 * already exists in shopping cart list, the quantity of that item is
 * incremented.
 *
 * @param product the <code>Product</code> that defines the type of
shopping cart item
 * @see ShoppingCartItem
 */
public synchronized void addItem(Product product) {
```

This enables you (and others working on the project) to view Javadoc documentation on the method. To demonstrate, open the Navigator (Ctrl-7; ⌘-7 on Mac) and hover your mouse over the `addItem` method.



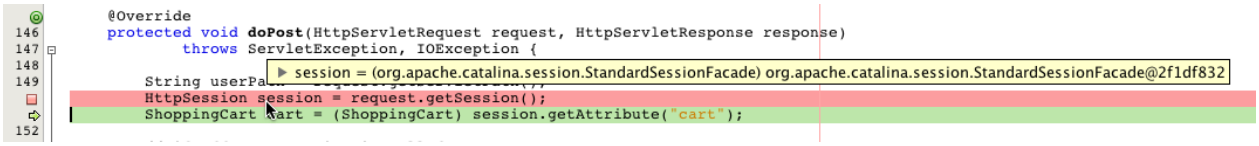
You can also use the IDE to generate a set of Javadoc HTML pages. In the Projects window, right-click your project node and choose Generate Javadoc. The IDE generates the Javadoc in the `dist/javadoc` folder of your project's directory and opens the index page in the browser.

For more information on Javadoc, see the following resources:

- Javadoc Tool Official Home Page
- How to Write Doc Comments for the Javadoc Tool

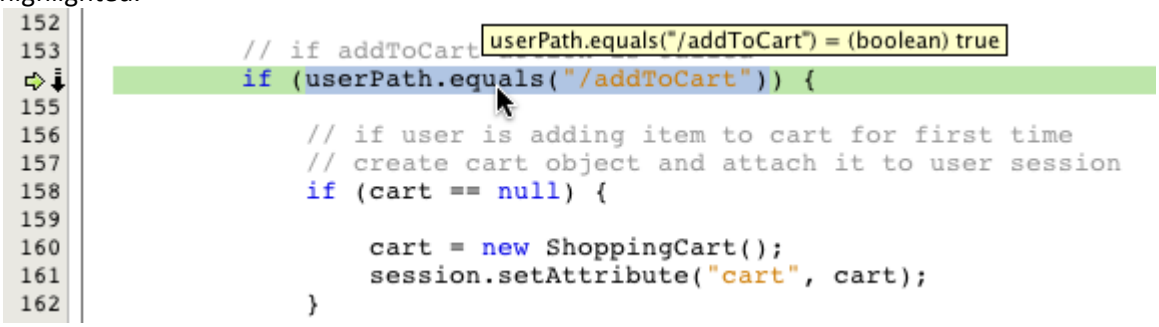
10. Hover your mouse over the `session` variable. Note that the debugger suspends on the line *it is about to execute.* The value returned by `getSession()` has not yet been saved into the `session` variable, and you see a popup stating that "`session` is not a known variable in the current context."

11. Click the Step Over ( ) button in the debugger toolbar located above the editor. The line is executed, and the debugger steps to the next line in the file.
12. Hover your mouse over the `session` variable again. Now you see the value currently set to the `session` variable.

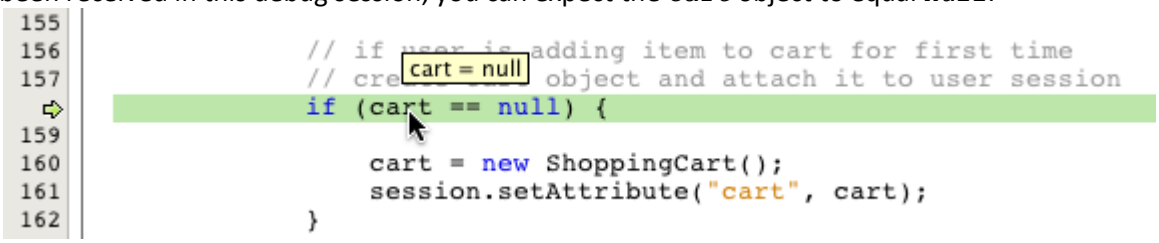


In NetBeans 6.9, you can click the grey pointer (▶) in the popup to expand a list of variable values contained in the highlighted element.

13. Click the Step Over ( ) button (F8; fn-F8 on Mac) to arrive at the `if` statement (line 154). Because you just clicked the 'add to cart' button in the browser, you know that the expression `userPath.equals("/addToCart")` should evaluate to `true`.
14. Highlight the `userPath.equals("/addToCart")` expression (by control-clicking with your mouse). This time you see a popup indicating the value of the expression you highlighted.



15. Press F8 (fn-F8 on Mac) to step to the next line (line 158). The application has been designed so that the `ShoppingCart` object for the user session is only created when the user adds an item to the cart for the first time. Since this is the first time the `addToCart` request has been received in this debug session, you can expect the `cart` object to equal `null`.



16. Press F8 (fn-F8 on Mac) to step to the next line (line 160). Then, on line 160, where the `ShoppingCart` object is created, click the Step Into () button. The debugger steps into the method being called. In this case, you are taken directly to the `ShoppingCart`'s constructor.

```
20
22      public ShoppingCart() {
23          items = new ArrayList<ShoppingCartItem>();
24          numberOfItems = 0;
25          total = 0;
        }
```

17. Press Ctrl-Tab to switch back to the `ControllerServlet`. Note that the IDE provides a Call Stack (◢) badge on line 160, indicating that the debugger is currently suspended somewhere on a method higher up in the call stack.

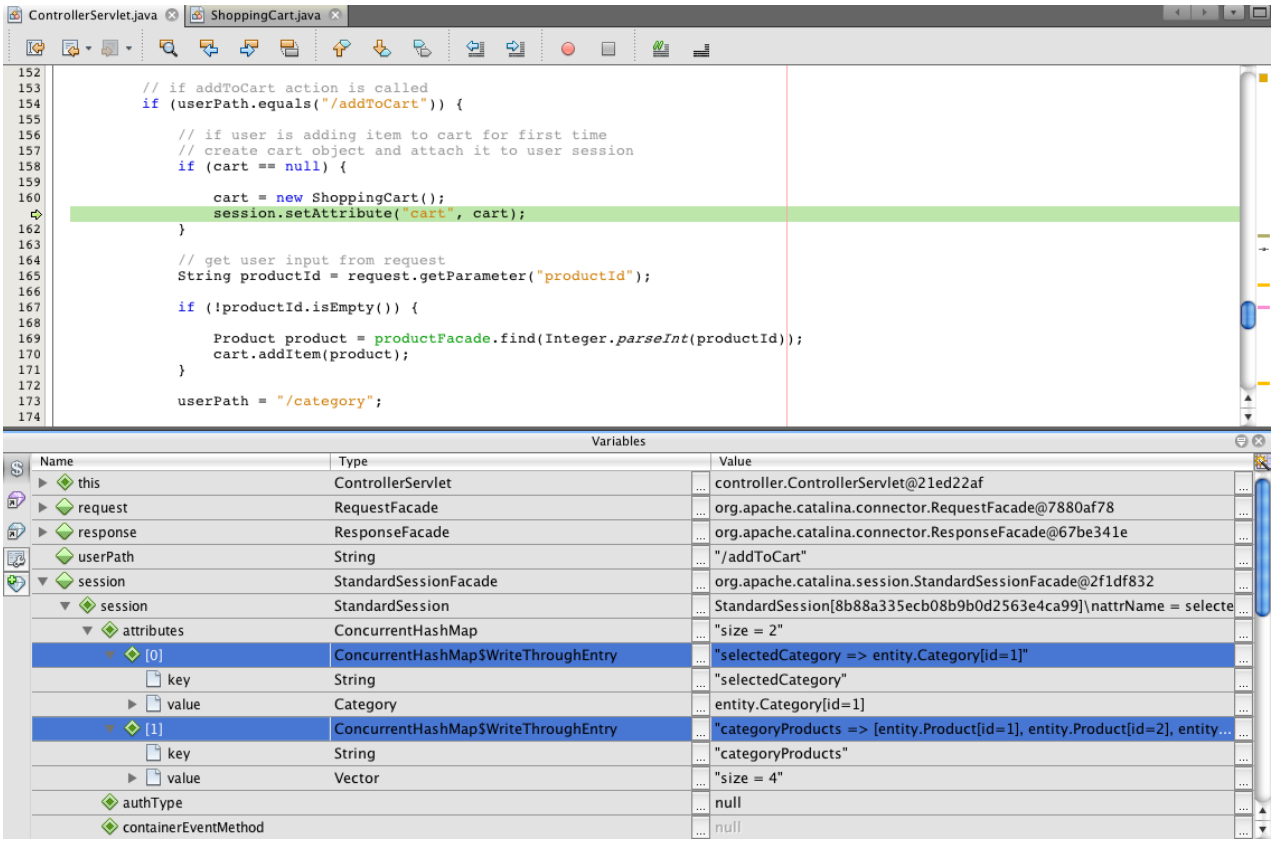
    Press Alt-Shift-3 (Ctrl-Shift-3 on Mac) to open the IDE's Call Stack window.

18. Press F8 (fn-F8 on Mac) to continue stepping through code. When the debugger completes the `ShoppingCart` constructor, you are taken back to the `ControllerServlet`.

    Line 161 of the `ControllerServlet` binds the newly-created `cart` object to the session.

    ```
    session.setAttribute("cart", cart);
    ```

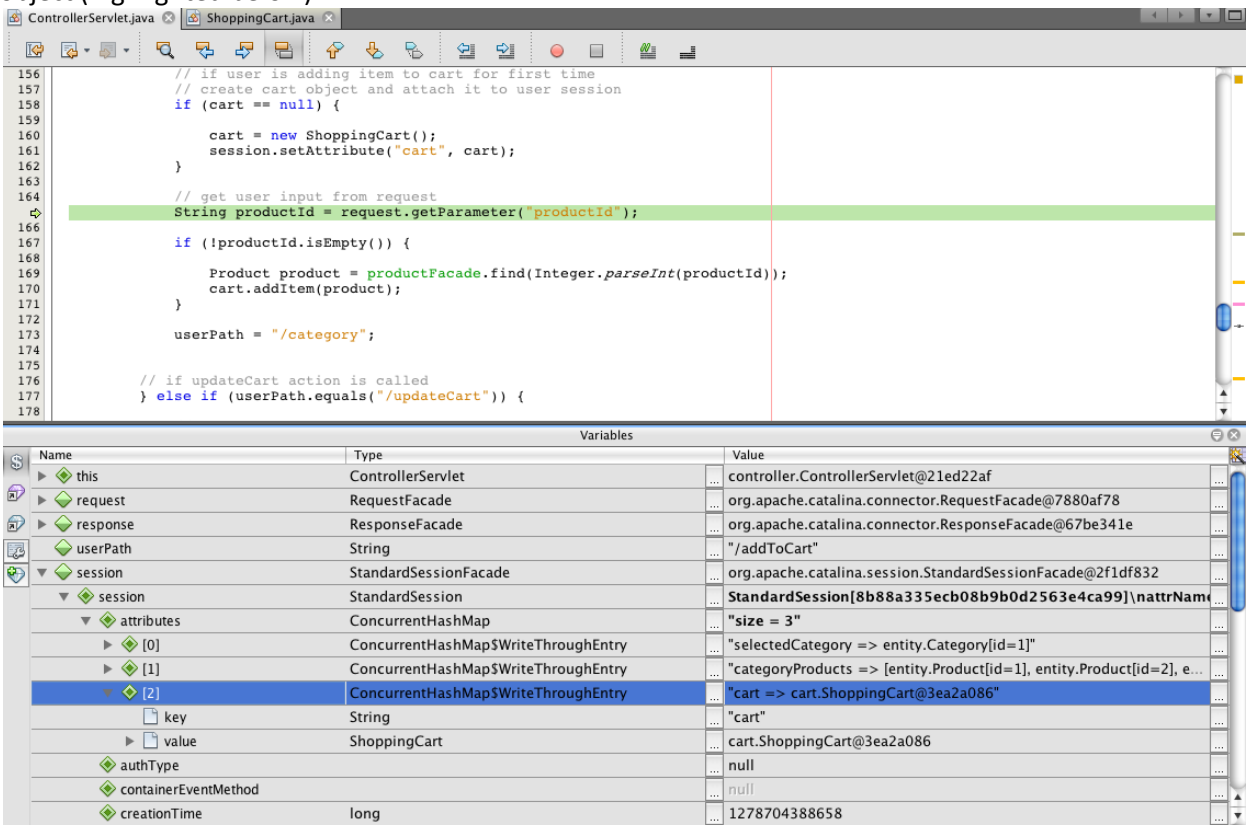
    To witness this, open the debugger's Variables window. Choose Window > Debugging > Variables, or press Alt-Shift-1 (Ctrl-Shift-1 on Mac).



If you expand the session > session > attributes node, you are able to view the objects that are bound to the session. In the above image, there are two items currently bound to the

session (highlighted). These are `selectedCategory` and `categoryProducts`, instantiated in the `ControllerServlet` at lines 83 and 89, respectively. Both of these items were bound earlier, when you clicked a category image, and the `ControllerServlet` processed the category page request.

19. Press F8 (fn-F8 on Mac) to execute line 161. The `cart` object is bound to the session, and the Variables window updates to reflect changes. In the Variables window, note that the session now contains three attributes, the third being the newly initialized `ShoppingCart` object (highlighted below).



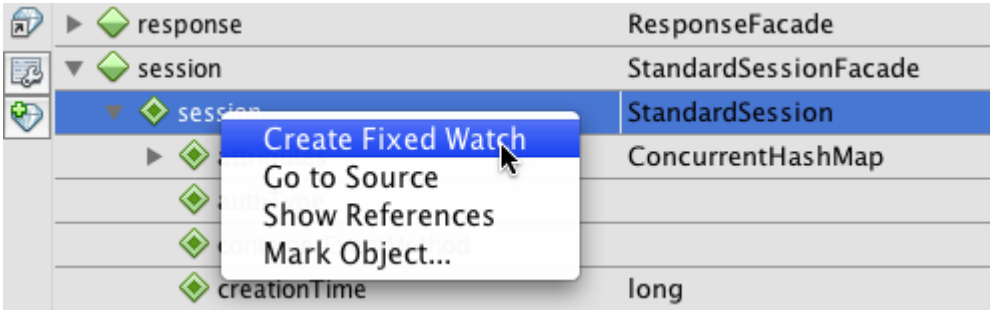
So far, we have not "proven" that the session, as listed in the Variables window, represents an `HttpSession`. As previously mentioned, `HttpSession` is actually an interface, so when we talk about an `HttpSession` object, or session object, we are in fact referring to any object that implements the `HttpSession` interface. In the Variables window, if you hover your cursor over '`session`', a popup displays indicating that the variable represents an `HttpSession` object. The `StandardSessionFacade` type, as displayed, is the internal class that GlassFish uses to implement the `HttpSession` interface. If you are familiar with Tomcat and are puzzled by the '`org.apache.catalina`' paths that appear in the Value column, this is because the GlassFish web/servlet container is in fact a derivative of the Apache Tomcat container.

A new `ShoppingCart` is added to the session, and the request continues to be processed. In order to complete implementation of the 'add to cart' functionality, the following actions are taken:

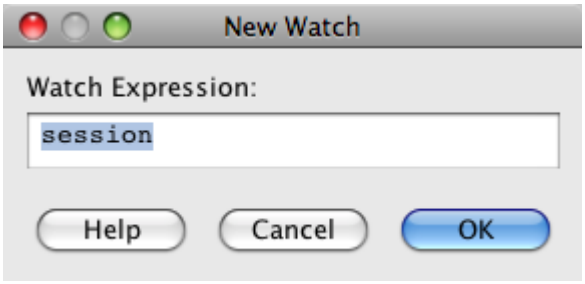
- the ID of the selected product is retrieved from the request (line 165)

- a `Product` object is created using the ID (line 169)
- a new `ShoppingCartItem` is created using the `product` (line 170)
- the `ShoppingCartItem` is added to `ShoppingCart`'s `items` list (line 170)

20. Press F8 (fn-F8 on Mac) to continue stepping through code while being mindful of the above-listed four actions. Pause when the debugger suspends on line 170.
21. Create a watch on the session. This will allow you to view values contained in the session when you step into the `addItem` method in the next step. Right-click the session in the Variables window and choose Create Fixed Watch.



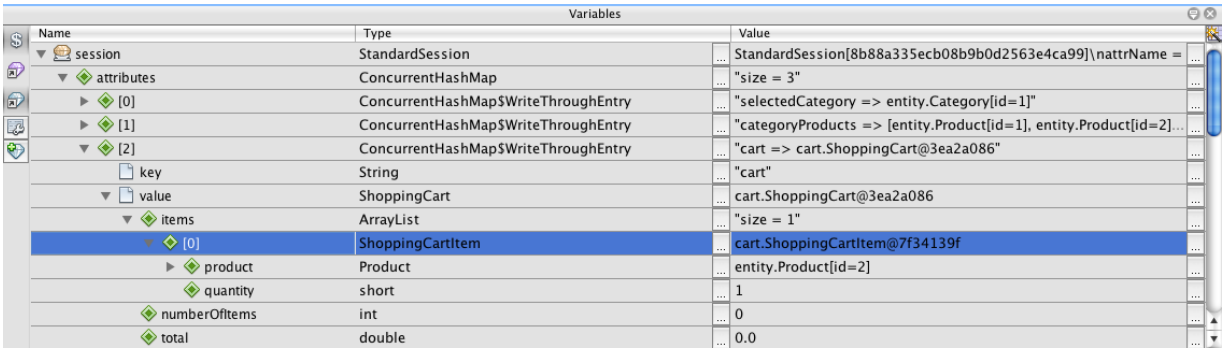
Alternatively, you can place your cursor on the `session` variable in the editor, then right-click and choose New Watch. The New Watch dialog enables you to specify variables or expressions to watch continuously when debugging an application. (In the case of expressions, highlight the expression first, then right-click and choose New Watch.)



A new watch is created on the `session` variable and all variables it contains. The watch is visible from the Watches window (Window > Debugging > Watches) or, if you toggle the Watches ( ) button in the left margin of the Variables window, it displays in the top row of the Variables window.

The debugger enables you to keep an eye on variables as it steps through code. This can be helpful, for example if you'd like to follow changes to specific variable values (and don't want to need to sift through the full list presented in the Variables window with each step), or if you temporarily step into a class that doesn't contain the variables you are interested in.

22. Click the Step Into ( ) button to step into `ShoppingCart`'s `addItem` method.
23. Step through the `addItem` method until you reach line 53. As the Javadoc states, `addItem` *"adds a `ShoppingCartItem` to the `ShoppingCart`'s `items` list. If item of the specified `product` already exists in shopping cart list, the quantity of that item is incremented."*
24. Examine the `session` variable which you created a watch on (step 21 above). The `items.add(scItem)` statement in line 51 added the new `ShoppingCartItem` to the `items` list in the `ShoppingCart`. This is evident by drilling into the third attribute (i.e., the `cart` variable) contained in the session.



At this stage, you can see how an `HttpSession` is created for the request, how a `ShoppingCart` object is created and attached to the session, and how a `ShoppingCartItem` is created based on the user's product choice, then added to the `ShoppingCart`'s list of `items`. The only remaining action is to forward the request to the `category.jsp` view.

25. Open the header JSP fragment (`header.jspf`) in the editor and place a breakpoint on line 86. This line contains the EL statement within the shopping cart widget that displays the number of cart items.

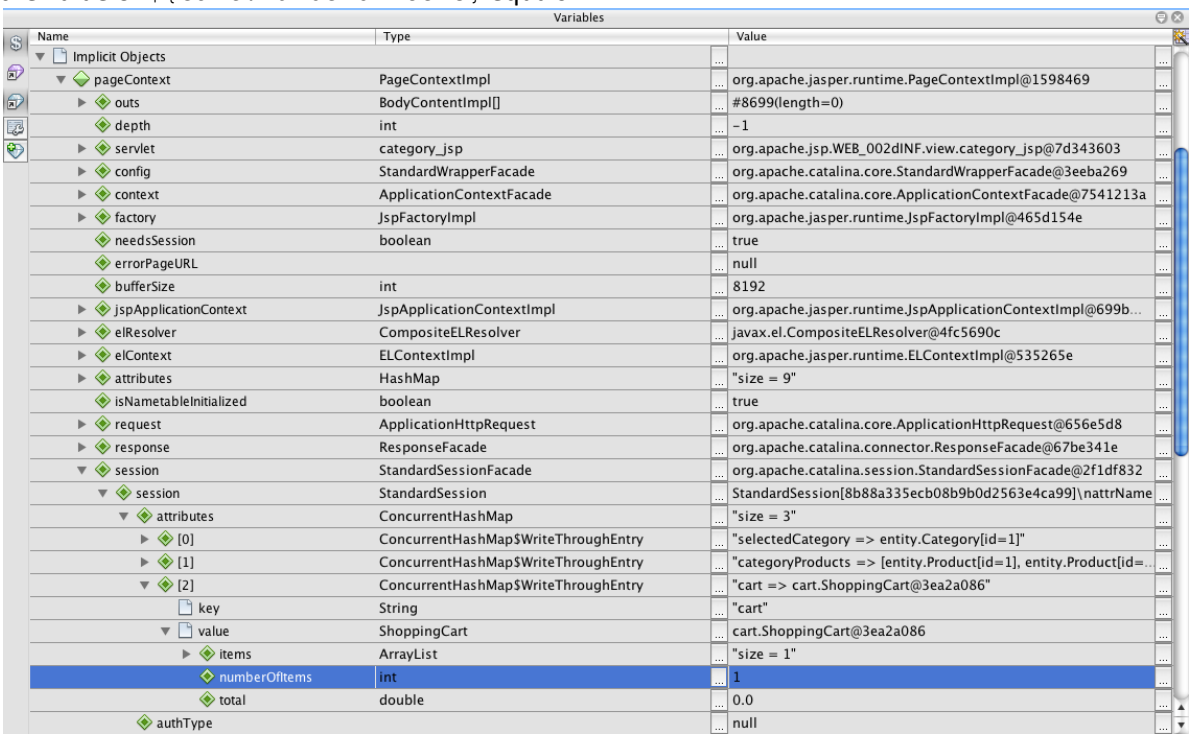
```
73                  <%-- shopping cart widget --%>
74                    <div class="headerWidget" id="viewCart">
75
76                      <img src="img/cart.gif" alt="shopping cart icon" id="cart">
77
78                      <%-- If 'numberOfItems' property doesn't exist, or if number o
79                           in cart is 0, output '0', otherwise output 'numberOfItems
80                      <span class="horizontalMargin">
81                        <c:choose>
82                          <c:when test="${cart.numberOfItems == null}">
83                            0
84                          </c:when>
85                          <c:otherwise>
                              ${cart.numberOfItems}
87                          </c:otherwise>
88                        </c:choose>
```

26. Click the Continue () button in the debugger toolbar. The debugger continues until execution completes, or until it reaches another breakpoint. In this case, the debugger suspends on line 86 in the header JSP fragment.

Note: In order to suspend the debugger in a JSP page, you need to set a breakpoint. For example, when the `ControllerServlet` forwards the request to the appropriate view, the debugger will not automatically suspend within the JSP page.

27. Open the Variables window (Alt-Shift-1; Ctrl-Shift-1 on Mac) if it is not already open. Unlike with Java classes, the debugger *does not* provide tooltips when you hover your mouse over variables or expressions in a JSP page. However, the Variables window does enable you to determine variable values as you step through code. So, where can you find the value for `${cart.numberOfItems}`?
28. In the Variables window, expand the Implicit Objects > pageContext > session > session > attributes node. This provides access to the session object, just as you saw earlier when working in the `ControllerServlet`. In fact, you may note that the session which you created a watch on in step 21 above points to the very same object. Here you can verify that

the value of `${cart.numberOfItems}` equals '1'.



Maximize the Variables window, or any window in the IDE, by right-clicking the window header, then choosing Maximize Window (Shift-Esc).

The debugger gives you access to the `pageContext` implicit object. `pageContext` represents the context of the JSP page, and offers direct access to various objects including the `HttpServletRequest`, `HttpSession`, and `ServletContext` objects. For more information, see the Java EE 5 Tutorial: Implicit Objects.

29. Click the Finish Session ( ) button. The runtime finishes executing, and the debug session terminates. The browser displays a fully-rendered category page, and you can see that the shopping cart widget in the page header contains one item.

Hopefully you now feel comfortable using the IDE's debugger not only to examine your project when it behaves unexpectedly, but also as a tool to become more familiar with code. Other useful buttons in the debugger toolbar include:

- ( ) **Step Out:** Steps you out of the current method call. Executes and removes the topmost method call in your call stack.
- ( ) **Run to Cursor:** Executes up to the line on which your cursor is placed.
- ( ) **Apply Code Changes:** After editing a file, you can press this button so that the file is recompiled and changes are taken into account in the debug session.
- ( ) **Step Over Expression:** Enables you to view the input parameters and resulting output values of each method call within an expression. You can inspect the output values for the previous method and the input parameters for the next method in the Local Variables window. When there are no further method calls, Step Over Expression behaves like the Step Over ( ) command.

Examining Session Tracking Options

There are three conventional ways of tracking sessions between client and server. By far the most common is with cookies. URL rewriting can be applied in the event that cookies are not supported or disabled. Hidden form fields can also be used as a means of "maintaining state" over multiple requests, but these are limited to usage within forms.

The `AffableBean` project includes an example of the hidden field method in both the category and cart pages. The 'add to cart' and 'update' buttons that display for product items contain a hidden field which relays the product ID to the server when the button is clicked. If you open the `cart.jsp` page in the editor, you'll see that the `<form>` tags contain a hidden field.

```
<form action="updateCart" method="post">
    <input type="hidden"
           name="productId"
           value="${product.id}">
    ...
</form>
```

In this manner, the product ID is sent as a request parameter which the server uses to identify the item within the user's cart whose quantity needs to be modified.

The Servlet API provides a high-level mechanism for managing sessions. Essentially, it creates and passes a cookie between the client and server with each request-response cycle. If the client browser doesn't accept cookies, the servlet engine automatically reverts to URL rewriting. The following two exercises demonstrate this functionality.

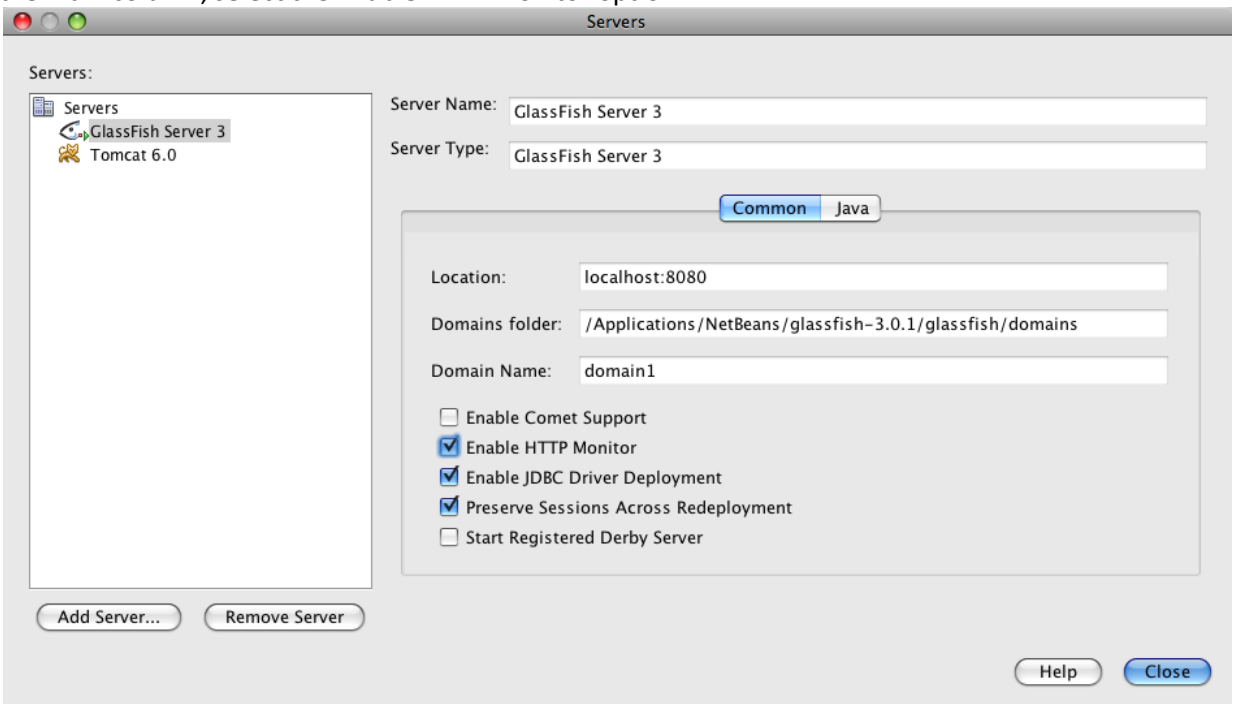
Examining Client-Server Communication with the HTTP Monitor

By default, the servlet engine uses cookies to maintain and identify sessions between requests. A random, alphanumeric number is generated for each session object, which serves as a unique identifier. This identifier is passed as a '`JSESSIONID`' cookie to the client. When the client makes a request, the servlet engine reads the value of the `JSESSIONID` cookie to determine the session which the request belongs to.

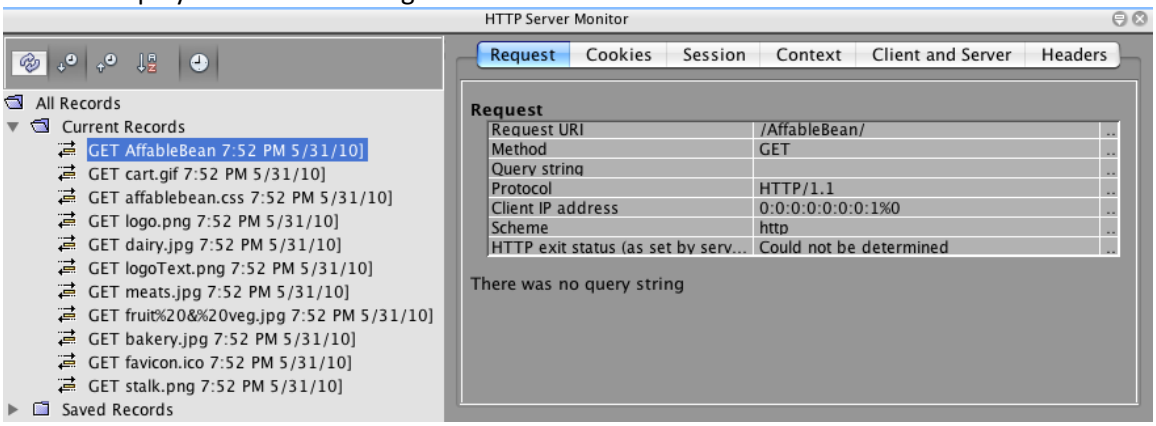
To demonstrate this, we'll use the debugger in tandem with the IDE's HTTP Monitor.

1. Begin by activating the HTTP Monitor for the server you are using. Choose Tools > Servers. In the left column of the Servers window, select the server you are using (GlassFish). Then, in

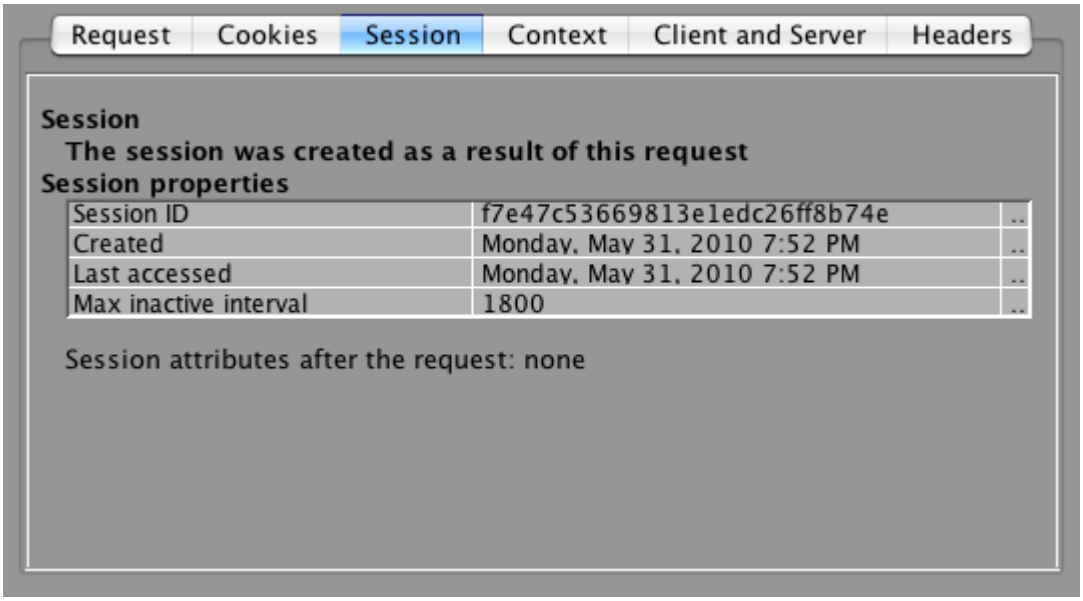
the main column, select the Enable HTTP Monitor option.



2. If your server is already running, you need to restart it. However, since we plan to use the debugger, and running the debugger restarts the server to communicate on a different port, just click the Debug Project ( ) button in the IDE's main toolbar. The server restarts, a debug session begins and the application's welcome page opens in your browser. The HTTP Monitor displays in the bottom region of the IDE.



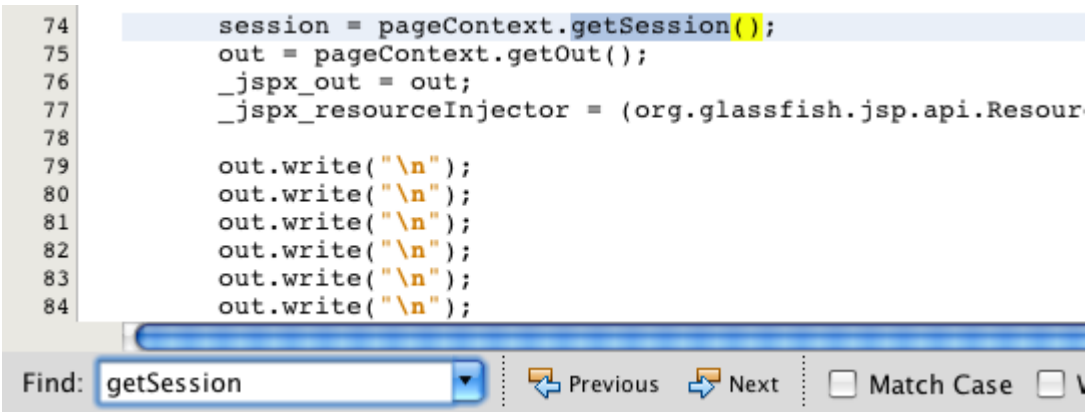
3. Click the AffableBean record in the left column (as shown in the above image). When you select records in the left column, the right (i.e., main) column refreshes to display corresponding data. In the above image, the Request tab displays the requested URI (`/AffableBean/`), the HTTP method (`GET`), and points out that there was no query string sent with the request.
4. Select the Session tab. Note that there is a statement, "The session was created as a result of this request." This is due to the fact that the server has sent a `Set-Cookie` header for the `JSESSIONID` cookie in its response. Also note that the new session ID is listed under 'Session properties'. As will later be shown, the session ID is the value of the `JSESSIONID` cookie.



You may wonder how a session object was created from a request for the site welcome page. After all, the `ControllerServlet` does not handle the initial request for `/AffableBean/`, and nowhere does this request encounter `getSession()`. Or does it? Recall that JSP pages are compiled into servlets upon deployment. Once you've deployed your project to the server, you can actually use the IDE to view the JSP's compiled servlet on your server.

5. In the Projects window, right-click the `index.jsp` file and choose View Servlet. An `index_jsp.java` file opens in the editor. This is the servlet that was automatically compiled from the `index.jsp` page.
6. Perform a search in the file for `getSession`. Press Ctrl-F (⌘-F on Mac), type '`getSession`' in the search bar, then press Enter.

   Ctrl-F (⌘-F on Mac) is a keyboard shortcut for Edit > Find.



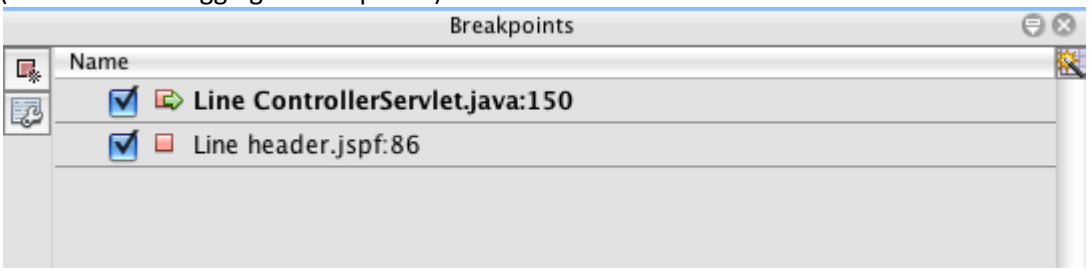
The `getSession` method is in fact called. The reason this occurs is because JSP pages include the `pageContext.session` implicit object by default. If you wanted to deactivate this behavior, you could add the following directive to the top of a JSP file:

```
<%@page session="false" %>
```

and the `getSession` method in the compiled servlet would be removed.

To find out the location of the compiled servlet on your server, you can hover your mouse over the servlet's name tab above the editor. A popup displays the path to the file on your computer.

7. In the browser, select a category then add an item to your cart. Switch back to the IDE. Note that the debugger suspends on the breakpoint in the `ControllerServlet` you set earlier (line 150). All breakpoints are remembered between sessions. To remove the breakpoint, you could click the breakpoint ( ) badge in the editor's left margin. However, since there are multiple breakpoints already set in the project, open the debugger's Breakpoints window (Window > Debugging > Breakpoints).

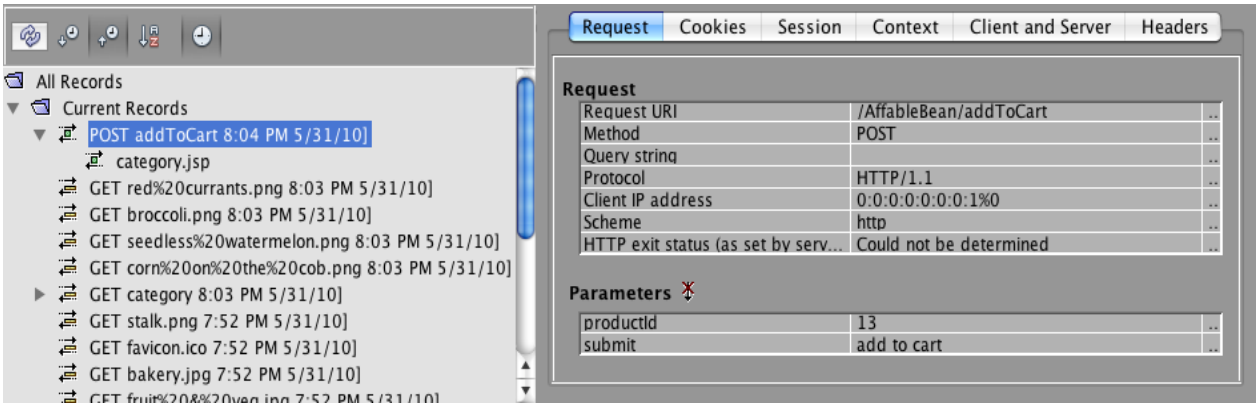


From the Breakpoints window, you can view and call actions on all breakpoints set in projects opened in the IDE.

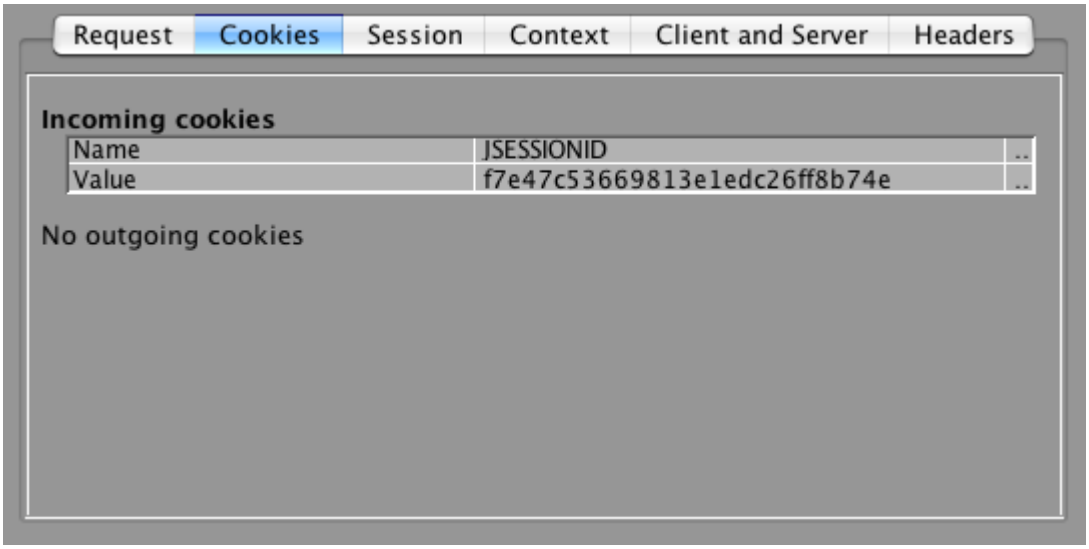
8. Right-click the breakpoint set in `header.jspf` and choose Delete. Then right-click the breakpoint set in the `ControllerServlet` and choose Disable. (You'll re-enable it later in this exercise.)
9. Click the Continue ( ) button. The request finishes executing, and the category page displays in the browser with one item added to the cart.
10. In the HTTP Monitor, search for the `addToCart` request in the left column, then select it to display details in the main column.

    Click the Ascending Sort ( ) button so that the most recent records are listed at the top.

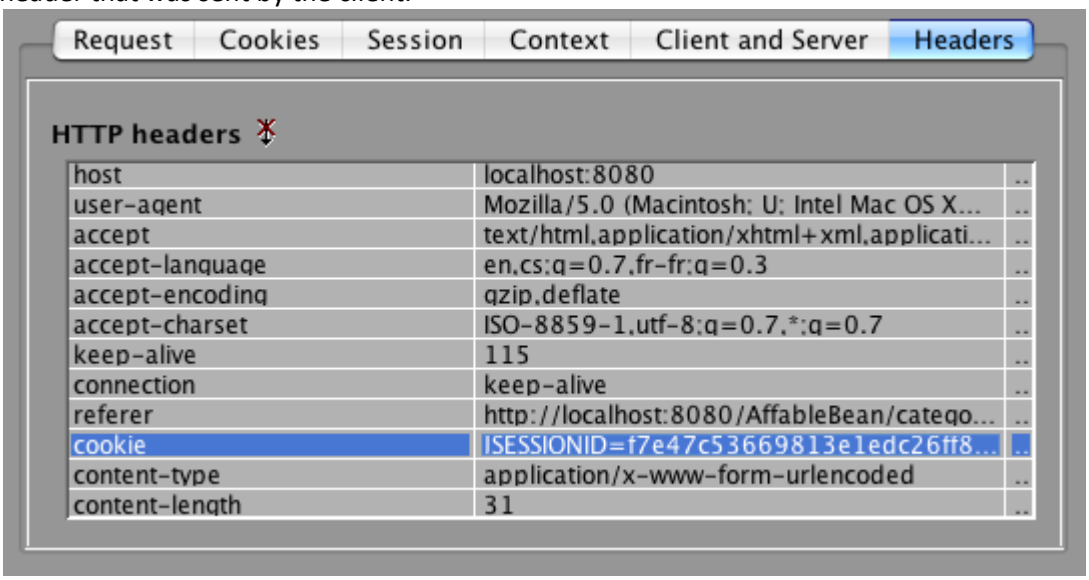
    Under the Request tab, note the requested URI (`/AffableBean/addToCart`), the HTTP method (`POST`), and the request parameters (`productId` and `submit`).



11. Select the Cookies tab. Here you see that a cookie named `JSESSIONID` exists, and was sent from the client to the server. Note that the value for the cookie is the same as the Session ID displayed under the Session tab.

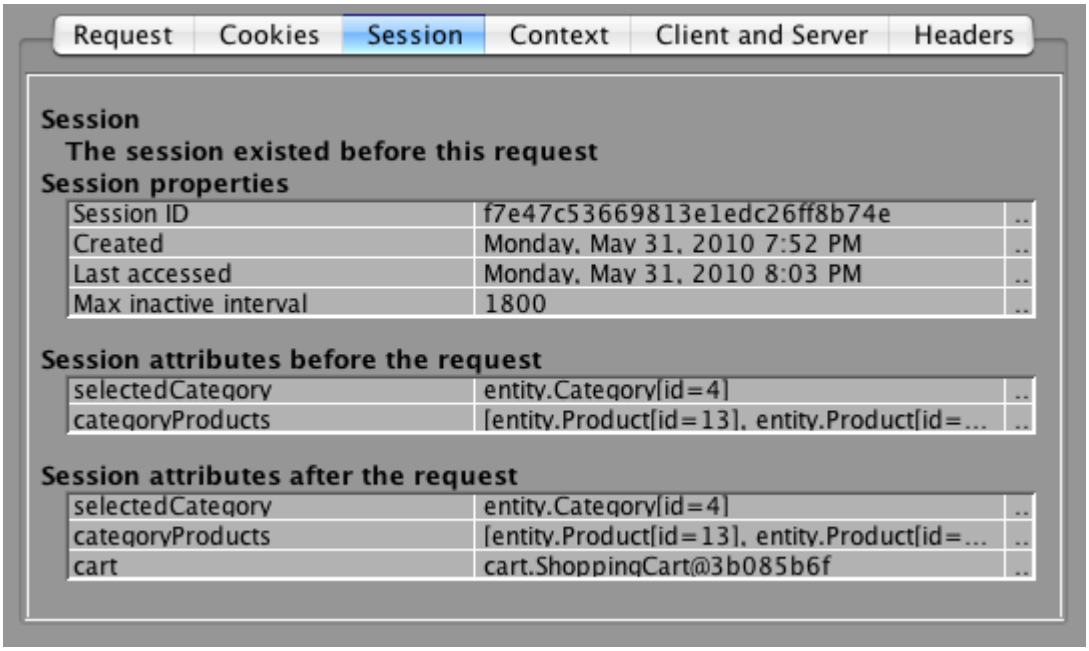


Likewise, if you click the Header tab, you see the cookie listed, since '`Cookie`' is a request header that was sent by the client.



See Wikipedia's List of HTTP headers for more information on request and response headers.

12. Select the Session tab. There is a statement which indicates, "The session existed before this request." Also note that the `cart` attribute is listed under 'Session attributes after the request'. This makes sense, since we know that the `cart` object is bound to the session when the `addToCart` request is processed for the first time.



In the next few steps, locate the session ID and `JSESSIONID` cookie in the Variables window.

13. Re-enable the breakpoint you set earlier in the `ControllerServlet`. Press Alt-Shift-5 (Ctrl-Shift-5 on Mac) to open the Breakpoints window, then click in the checkbox next to the breakpoint entry to re-enable it.
14. In the browser, click the 'add to cart' button for one of the listed products.
15. Switch to the IDE and note that the debugger is suspended on the breakpoint set in the `ControllerServlet`. Click the Step Over ( ) button so that the `session` variable is assigned to the session object.
16. Open the Variables window (Alt-Shift-1; Ctrl-Shift-1 on Mac) and expand session > session. You'll find the session ID listed as the value for the `id` variable.
17. To locate the `JSESSIONID` cookie, recall that you can normally access cookies from a servlet by calling the `getCookies` method on the `HttpServletRequest`. Therefore, drill into the request object: request > Inherited > request > request > Inherited > cookies. Here you see the `cookies` ArrayList. If you expand the list, you'll find the `JSESSIONID` cookie, the value of which is the session ID.
18. Click the Finish Session ( ) button to terminate the debug session.

Maintaining Sessions with URL Rewriting

As mentioned, the servlet engine detects whether cookies are supported for the client browser, and if not, it switches to URL rewriting as a means of maintaining sessions. This all happens transparently for the client. For you, the developer, the process isn't entirely transparent.

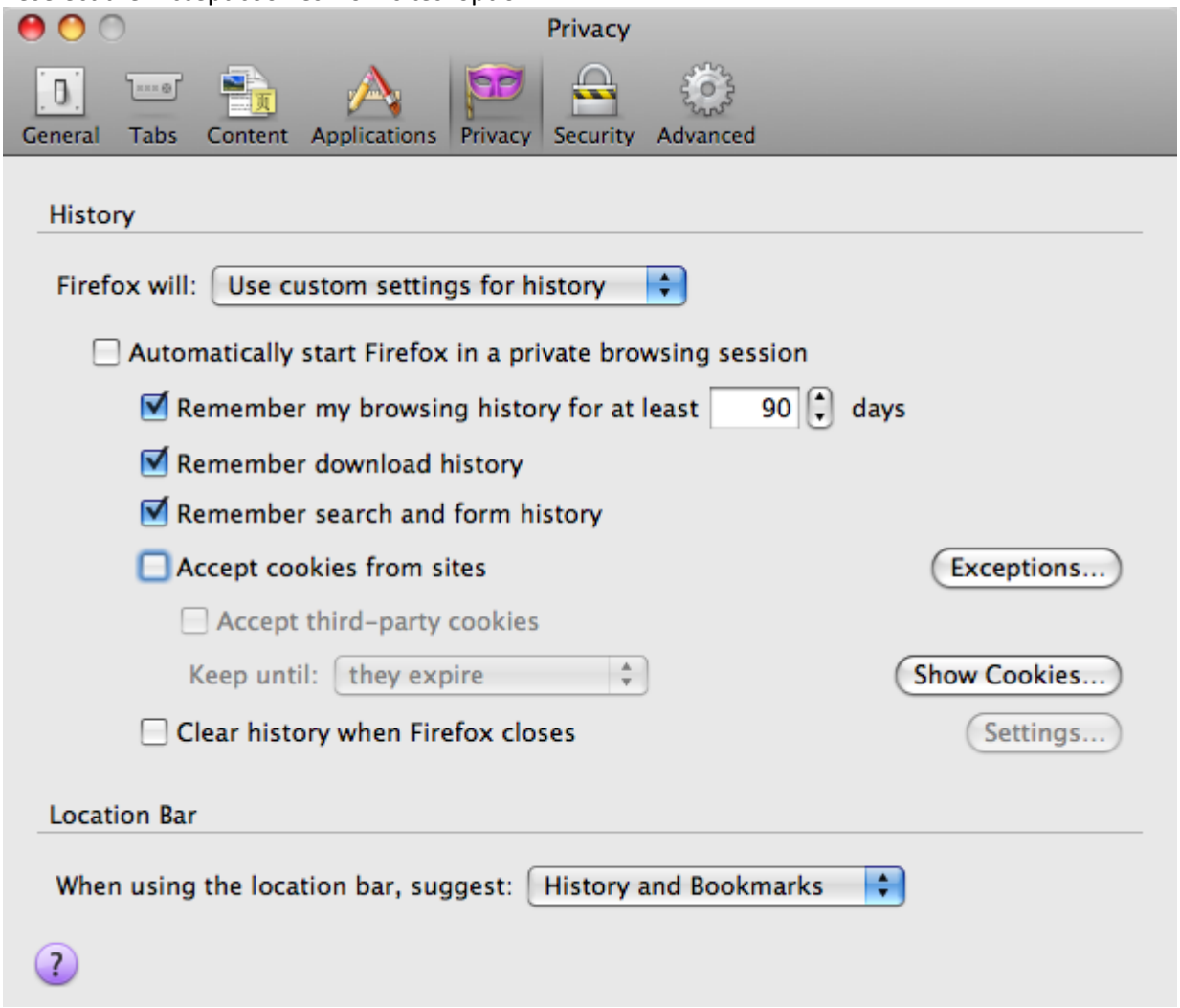
You need to ensure that the application is capable of rewriting URLs whenever cookies are disabled. You do this by calling the response's `encodeURL` method on all URLs returned by servlets in your application. Doing so enables the session ID to be appended to the URL in the event that the use of cookies is not an option; otherwise, it returns the URL unchanged.

For example, the browser sends a request for `AffableBean`'s third category (bakery): `category?3`. The server responds with session ID included in the URL:

```
/AffableBean/category;jsessionid=364b636d75d90a6e4d0085119990?3
```

As stated above, *all URLs returned by your application's servlets must be encoded*. Keep in mind that JSP pages are compiled into servlets. How can you encode URLs in JSP pages? JSTL's `<c:url>` tag serves this purpose. The following exercise demonstrates the problem and illustrates a solution.

1. Temporarily disable cookies in your browser. If you are using Firefox, you can choose Tools > Options (Firefox > Preferences on Mac). In the window that displays, select the Privacy tab, then under History, select 'Use custom settings for history' in the provided drop-down. Deselect the 'Accept cookies from sites' option.



2. Run the `AffableBean` project. When the welcome page displays, click into a category, then try adding an item to your cart. You'll see that the application's functionality is severely compromised in its present state.

As before, the server generates a session and binds objects to it. This is how the category page is able to display the selected category and products. However, the server has failed in its attempt to set a `JSESSIONID` cookie. Therefore, when the client makes a second request (when user clicks 'add to cart'), the server has no way of identifying the session which the request belongs to. It therefore cannot locate any of the attributes previously set in the session, such as `selectedCategory` and `categoryProducts`. This why the rendered response lacks the information specified by these attributes.

3. Open the project's `category.jsp` page in the editor. Locate the line that implements the 'add to cart' button (line 58). The `<form>` element's `action` attribute determines the request sent to the server.

   ```
   <form action="addToCart" method="post">
   ```

4. Modify the request so that it is passed through the `<c:url>` tag.

   ```
   <form action="<c:url value='addToCart'/>" method="post">
   ```

5. Press Ctrl-S (⌘-S on Mac) to save changes to the file. Recall that the IDE provides the Deploy on Save feature, which is enabled by default. This means that any saved changes are automatically deployed to your server.
6. In the browser, select a different category so that the application renders the newly modified category page.
7. Examine the source code for the page. In Firefox, you can press Ctrl-U (⌘-U on Mac). The 'add to cart' button for each product displays with the session ID appended to the URL.

```
<form action="addToCart;jsessionid=4188657e21d72f364e0782136dde"
method="post">
```

8. Click the 'add to cart' button for any item. You see that the server is now able to determine the session which the request belongs to, and renders the response appropriately.
9. Before proceeding, make sure to re-enable cookies for your browser.

Again, every link that a user is able to click on within the application, whose response requires some form of session-related data, needs to be properly encoded. Sometimes implementation is not as straight-forward as the example shown above. For example, the 'clear cart' widget used in `cart.jsp` currently sets a `clear` parameter to `true` when the link is clicked.

```
<%-- clear cart widget --%>
<c:if test="${!empty cart && cart.numberOfItems != 0}">
    <a href="viewCart?clear=true" class="bubble hMargin">clear cart</a>
</c:if>
```

The `<c:url>` tag can be applied to the URL in the following manner:

```
<%-- clear cart widget --%>
<c:if test="${!empty cart && cart.numberOfItems != 0}">

    <c:url var="url" value="viewCart">
        <c:param name="clear" value="true"/>
    </c:url>

    <a href="${url}" class="bubble hMargin">clear cart</a>
</c:if>
```

The `clear=true` parameter is set by adding a `<c:param` tag between the `<c:url>` tags. A variable named '`url`' is set using <c:url>'s `var` attribute, and `var` is then accessed in the HTML anchor tag using the `${url}` expression.

You can download and examine snapshot 6 to see how all links in the project have been encoded.

URL rewriting should only be used in the event that cookies are not an available tracking method. URL rewriting is generally considered a suboptimal solution because it exposes the session ID in logs, bookmarks, referer headers, and cached HTML, in addition to the browser's address bar. It also requires more server-side resources, as the server needs to perform additional steps for each incoming request in order to extract the session ID from the URL and pair it with an existing session.

Handling Session Time-Outs

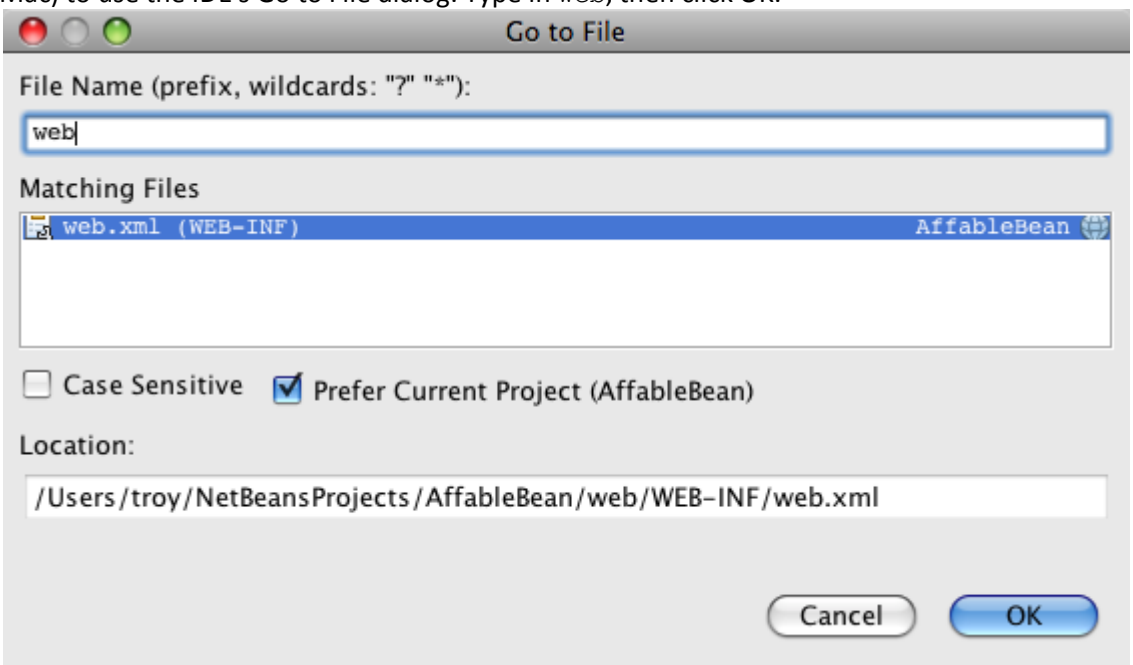
- Setting Session Time Intervals
- Programmatically Handling Session Time-Outs

Setting Session Time Intervals

You should consider the maximum time interval which your server maintains sessions for. If your website receives heavy traffic, a large number of sessions could expend your server's memory capacity. You might therefore shorten the interval in hopes of removing unused sessions. On the other hand, you certainly wouldn't want to cut sessions too short, as this could become a usability issue that might have a negative impact on the business behind the website. Taking the `AffableBean` application as an example, a user proceeds to checkout after filling her shopping cart with items. She then realizes she needs to enter her credit card details and goes off to find her purse. After returning to her computer with credit card in hand, she fills in the checkout form and clicks submit. During this time however, her session has expired on the server. The user sees that her shopping cart is empty and is redirected to the homepage. Will she really take the time to step through the process again?

The following steps demonstrate how to set the session time-out interval in the `AffableBean` project to 10 minutes. Of course, the actual duration ultimately depends on your server resources, the business objectives of the application, and the popularity of your website.

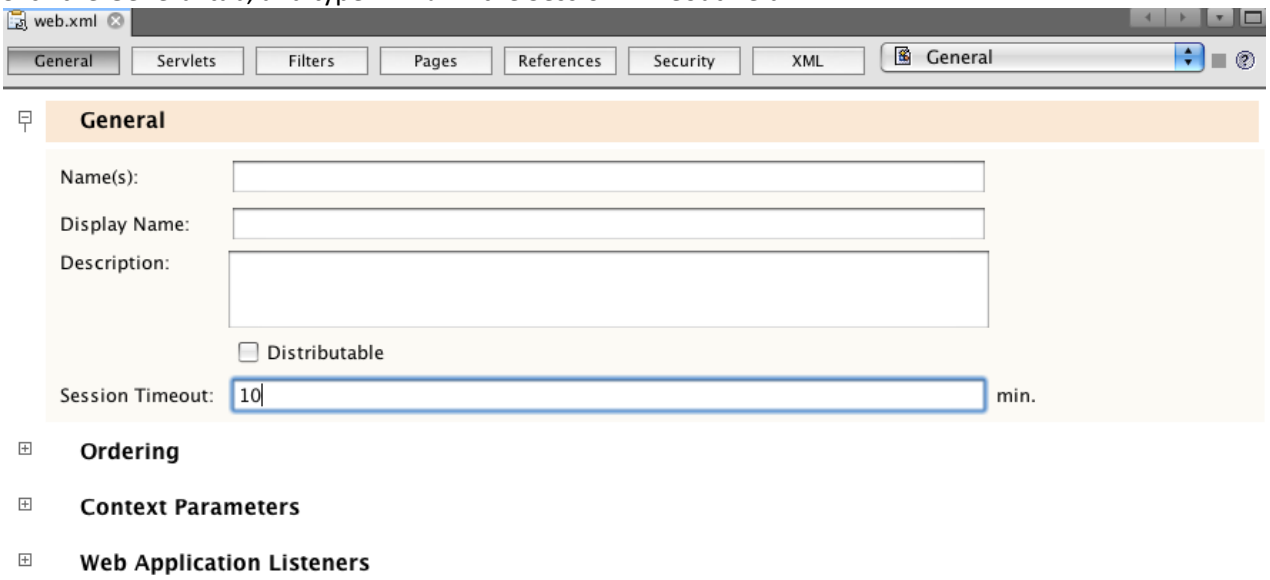
1. Open the application's deployment descriptor in the editor. Press Alt-Shift-O (Ctrl-Shift-O on Mac) to use the IDE's Go to File dialog. Type in `web`, then click OK.



   The editor displays the `web.xml` file in the XML view. The template that NetBeans provides for the `web.xml` file includes a default setting for 30 minutes.

```
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
```

6. Click the General tab, and type in '10' in the Session Timeout field.



7. Save the file (Ctrl-S; ⌘-S on Mac).

   If you switch back to the XML view, you'll see that the `<session-timeout>` element has been updated.

```
8. <session-config>
9.     <session-timeout>10</session-timeout>
   </session-config>
```

Note: Alternatively, you could remove the `<session-timeout>` element altogether, and edit the `session-properties` element in the GlassFish-specific deployment descriptor (`sun-web.xml`). This would set the global time-out for all applications in the server's web module. See the Sun GlassFish Enterprise Server v3 Application Development Guide: Creating and Managing Sessions for more details.

Programmatically Handling Session Time-Outs

If your application relies on sessions, you need to take measures to ensure that it can gracefully handle situations in which a request is received for a session that has timed out or cannot be identified. You can accomplish this in the `AffableBean` application by creating a simple filter that intercepts requests heading to the `ControllerServlet`. The filter checks if a session exists, and if not, it forwards the request to the site's welcome page.

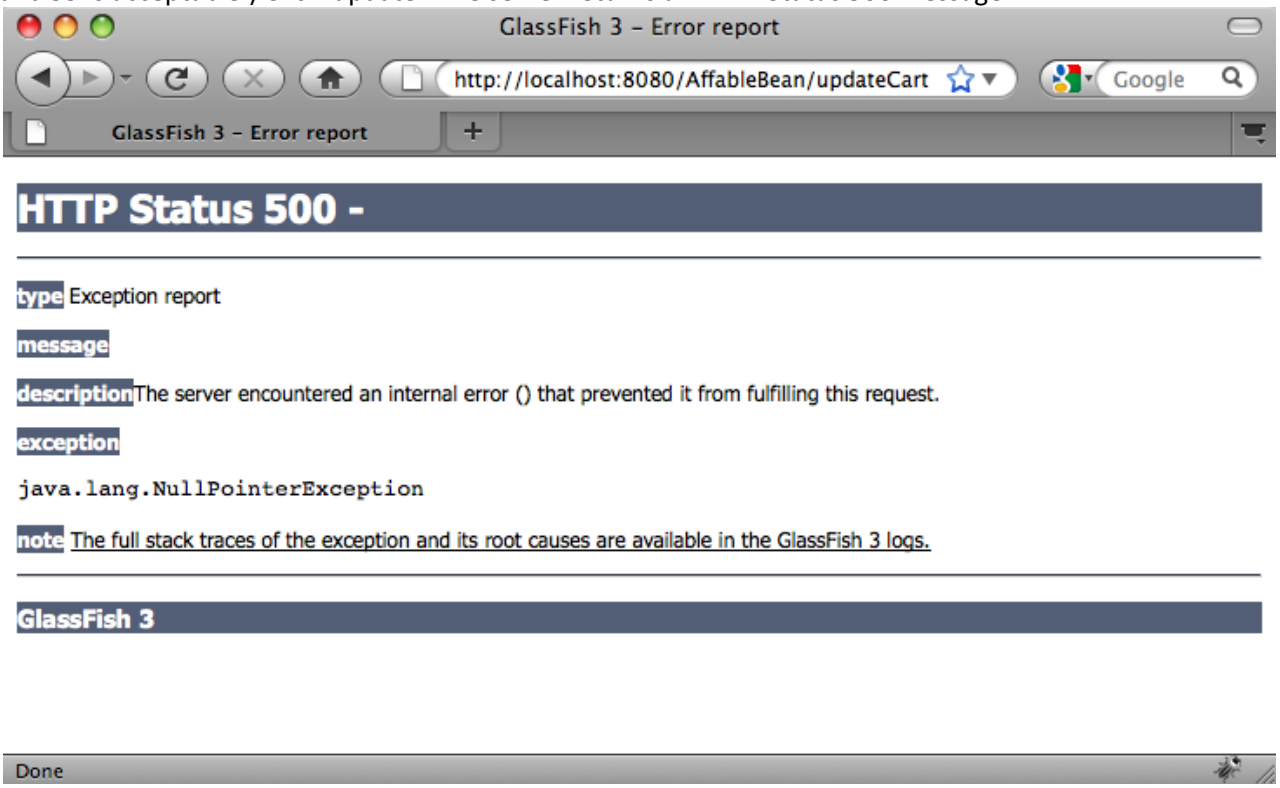
1. Start by examining the problem that arises when a session times out midway through a user's visit to the site. Temporarily reset the session time-out interval to one minute. Open the web deployment descriptor (`web.xml`) and enter '1' between the `<session-timeout>` tags.

```
2. <session-config>
3.     <session-timeout>1</session-timeout>
   </session-config>
```

4. Run the `AffableBean` project. In the browser, click into a category page, add several items to your cart, then click 'view cart'.

5. Wait at least one full minute.

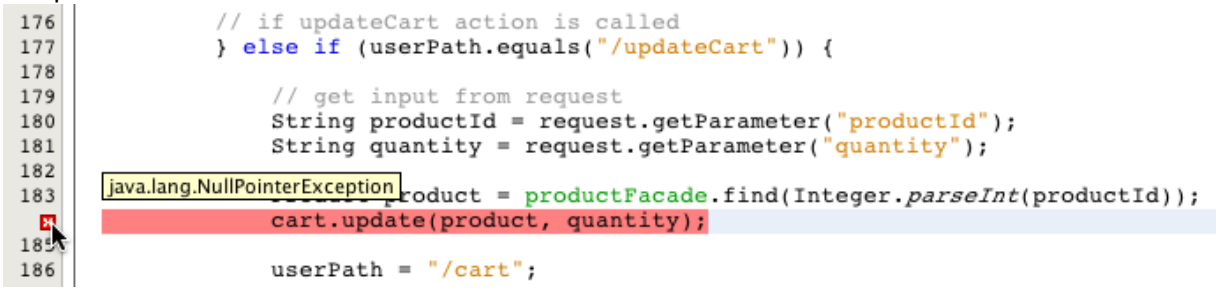
6. Update the quantity for one of the items displayed in the cart page. (Any number between 1 and 99 is acceptable.) Click 'update'. The server returns an HTTP Status 500 message.



7. Examine the GlassFish server log in the IDE. Open the Output window (Ctrl-4; ⌘-4 on Mac) and select the GlassFish Server tab. Scroll to the bottom of the log to examine the error's stack trace.

The server log indicates that a `NullPointerException` occurred at line 184 in the `ControllerServlet`. The Output window forms a link to the line where the exception occurred.

8. Click the link. You navigate directly to line 184 in the `ControllerServlet`. Hovering your mouse over the error badge in the editor's left margin provides a tooltip describing the exception.



Because the session had already expired before the request was received, the servlet engine was unable to associate the request with its corresponding session. It was therefore unable to locate the `cart` object (line 151). The exception finally occurred in line 184 when the engine attempted to call a method on a variable equating to `null`.

Now that we've identified the problem, let's fix it by implementing a filter.

9. Click the New File () button in the IDE's toolbar. (Alternatively, press Ctrl-N; ⌘-N on Mac.)
10. Select the **Web** category, then select **Filter** and click Next.
11. Name the filter `SessionTimeoutFilter`. Type `filter` into the Packages field so that the filter class is placed in a new package when created.
12. Click Next. Accept default settings and click Finish. A template for the `SessionTimeoutFilter` is generated and opens in the editor.

Note: Currently, in NetBeans 6.9, it isn't possible to use the wizard to set a mapping to a servlet that isn't registered in the web deployment descriptor. (`ControllerServlet` was registered using the `@WebServlet` annotation.) We'll therefore modify the generated code in the next step.

13. Modify the `@WebFilter` annotation signature so that it appears as follows.

```
14.  @WebFilter(servletNames = {"Controller"})
     public class SessionTimeoutFilter implements Filter {
```

This sets the filter to intercept any requests that are handled by the `ControllerServlet`. (Alternatively, you could have kept the `urlPatterns` attribute, and listed all patterns that the `ControllerServlet` handles.)

Note that '`Controller`' is the name of the `ControllerServlet`, as specified in the servlet's `@WebServlet` annotation signature. Also note that you've removed the `filterName` attribute, since the name of the filter class is used by default.

The IDE's filter template provides a lot of interesting code which is worth inspecting in its own right. However, most of it is not needed for our purposes here. Any filter class must implement the `Filter` interface, which defines three methods:

- **`init`:** performs any actions after the filter is initialized but before it is put into service
- **`destroy`:** removes the filter from service. This method can also be used to perform any cleanup operations.
- **`doFilter`:** used to perform operations for each request the filter intercepts

Use the Javadoc Index Search to pull up documentation on the `Filter` interface. Press Shift-F1 (fn-Shift-F1 on Mac), then type '`Filter`' into the search field and hit Enter. Select the 'Interface in javax.servlet' entry. The Javadoc documentation displays in the lower pane of the index search tool.

15. Replace the body of the `SessionTimeoutFilter` with the following contents.

```
16.  @WebFilter(servletNames = {"Controller"})
17.  public class SessionTimeoutFilter implements Filter {
18.
```

```
19.      public void doFilter(ServletRequest request, ServletResponse
   response, FilterChain chain)
20.              throws IOException, ServletException {
21.
22.          HttpServletRequest req = (HttpServletRequest) request;
23.
24.          HttpSession session = req.getSession(false);
25.
26.          // if session doesn't exist, forward user to welcome page
27.          if (session == null) {
28.              try {
29.
   req.getRequestDispatcher("/index.jsp").forward(request, response);
30.              } catch (Exception ex) {
31.                  ex.printStackTrace();
32.              }
33.              return;
34.          }
35.
36.          chain.doFilter(request, response);
37.      }
38.
39.      public void init(FilterConfig filterConfig) throws
   ServletException {}
40.
41.      public void destroy() {}
42.
   }
```

43. Press Ctrl-Shift-I (⌘-Shift-I on Mac) to fix import statements. (Imports need to be added for `HttpServletRequest` and `HttpSession`.) Also, use the editor hints to add the `@Override` annotation to the `init`, `destroy`, and `doFilter` methods.

    In the coming steps, you run the debugger on the project and step through the `doFilter` method to see how it determines whether the request is bound to an existing session.
44. Open the Breakpoints window (Alt-Shift-5; Ctrl-Shift-5 on Mac) and ensure that you do not have any existing breakpoints set. To delete a breakpoint, right-click the breakpoint and choose Delete. (If you completed the above exercise, Examining Client-Server Communication with the HTTP Monitor, you may have an outstanding breakpoint set in the `ControllerServlet`.)
45. Run the debugger. Click the Debug Project () button in the IDE's main toolbar.
46. When the welcome page displays in the browser, select a category, then add several items to your shopping cart.
47. Set a breakpoint on the line in `SessionTimeoutFilter`'s `doFilter` method that tries to access the session (line 32).

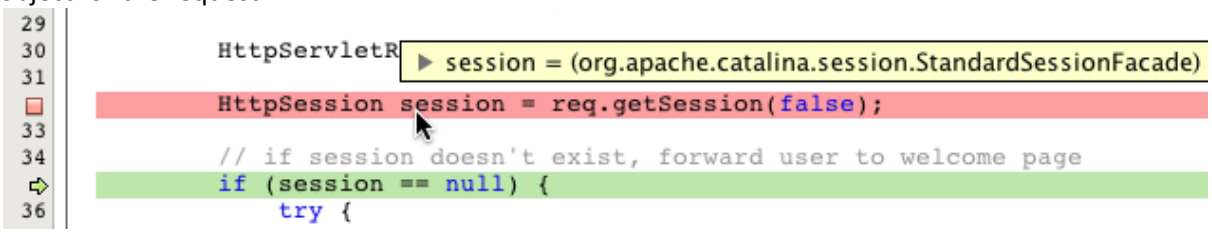
```
27      @Override
28      public void doFilter(ServletRequest request, ServletResponse response
29              throws IOException, ServletException {
30
31          HttpServletRequest req = (HttpServletRequest) request;
32
33          HttpSession session = req.getSession(false);
```

48. In the browser, click the 'view cart' button. Switch to the IDE and note that the debugger has suspended on the breakpoint.

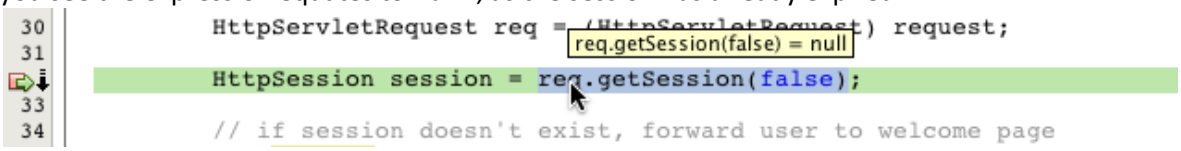
    Recall that `getSession()` creates a new session object if the current one doesn't exist.

Here, we use `getSession(false)`, which refrains from creating a new object if none is found. In other words, the method returns `null` if the session doesn't exist.

49. Click the Step Over () button, then hover your mouse over the `session` variable. Provided that a minute hasn't passed since the previous request was sent, you'll see that the variable has been assigned to a `StandardSessionFacade`. This represents the session object for the request.



50. Continue stepping through the method until the request is processed. Since `session` doesn't equal `null`, you skip the `if` statement and `chain.doFilter` then forwards the request to the `ControllerServlet` (line 44).
51. In the browser, make sure a full minute has passed, then update a quantity for one of the product items in your cart. This is the same procedure we went through earlier in the exercise when the status 500 message was returned. Now that the filter intercepts requests heading to the `ControllerServlet`, let's see what happens when a session time-out occurs.
52. After clicking 'update', switch to the IDE and note that the debugger is again suspended on the breakpoint set in the filter.
53. Highlight the `req.getSession(false)` expression, then hover your mouse over it. Here you see the expression equates to `null`, as the session has already expired.



54. Continue stepping through the code. Now that the `session` variable equals `null`, the `if` statement on line 35 is processed, and the request is forwarded to `/index.jsp`. When the debugger finishes executing, you'll see that the browser displays the site's welcome page.
55. Click the Finish Session () button to terminate the debug session.
56. Open the project's `web.xml` file and change the session time-out interval back to 10 minutes.

```
57.  <session-config>
58.      <session-timeout>10</session-timeout>
     </session-config>
```

59. Save (Ctrl-S; ⌘-S on Mac) the file.

One final topic concerning session management should be mentioned. You can explicitly terminate a session by calling the `invalidate` method on the session object. If the session is no longer needed, it should be removed in order to conserve the memory available to your server. After you complete the next unit, Integrating Transactional Business Logic, you will see how the `ControllerServlet`, upon successfully processing a customer order, destroys the user's `cart` object and terminates the session using the `invalidate` method.

```
// if order processed successfully send user to confirmation page
if (orderId != 0) {

    // dissociate shopping cart from session
    cart = null;
```

```
        // end session
        session.invalidate();

        ...
    }
```

This is demonstrated in project snapshot 8 (and later snapshots).

The NetBeans E-commerce Tutorial - Integrating Transactional Business Logic

Tutorial Contents

The purpose of this tutorial unit is to demonstrate how you can use the object-relational mapping (ORM) capabilities provided by EJB and JPA technologies to gather data from a web request and write to a back-end database. Of particular interest is EJB's support for *container-managed* transactions (refer to the GlassFish v3 Java EE Container diagram). By applying several non-intrusive annotations, you can transform your EJB class into a transaction manager, thereby ensuring the integrity of the data contained in the database. In other words, the transaction manager handles multiple write actions to the database as a single unit of work. It ensures that the work-unit is performed either in its entirety or, if failure occurs at some point during the process, any changes made are rolled back to the database's pre-transaction state.

Within the context of the `AffableBean` application, this tutorial unit focuses on processing a customer order when data from the checkout form is received. You create an `OrderManager` EJB to process the checkout form data along with the session `cart` object. The `OrderManager` performs a transaction that involves multiple write actions to the `affablebean` database. If any of the actions fails, the transaction is rolled back.

You can view a live demo of the application that you build in this tutorial: NetBeans E-commerce Tutorial Demo Application.

Software or Resource	Version Required
NetBeans IDE	Java bundle, 6.8 or 6.9
Java Development Kit (JDK)	version 6
GlassFish server	v3 or Open Source Edition 3.0.1
AffableBean project	snapshot 7

Notes:

- The NetBeans IDE requires the Java Development Kit (JDK) to run properly. If you do not have any of the resources listed above, the JDK should be the first item that you download and install.
- The NetBeans IDE Java Bundle includes Java Web and EE technologies, which are required for the application you build in this tutorial.
- The NetBeans IDE Java Bundle also includes the GlassFish server, which you require for this tutorial. You could download the GlassFish server independently, but the version provided with the NetBeans download has the added benefit of being automatically registered with the IDE.
- You can follow this tutorial unit without having completed previous units. To do so, see the setup instructions, which describe how to prepare the database and establish connectivity between the IDE, GlassFish, and MySQL.

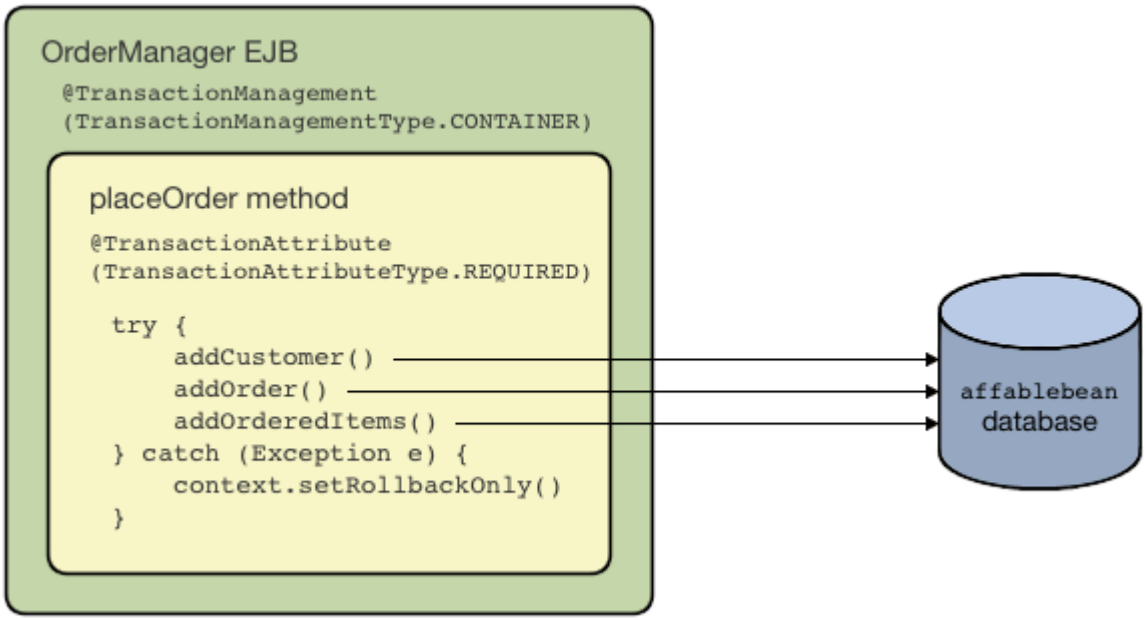
Overview of the Transaction

In order to process the data from the checkout form as well as the items contained in the customer's shopping cart, you create an `OrderManager` EJB. The `OrderManager` uses the provided data and performs the following write actions to the database:

- A new `Customer` record is added.
- A new `CustomerOrder` record is added.
- New `OrderedProduct` records are added, according to the items contained in the `ShoppingCart`.

We'll implement this by creating a `placeOrder` method which performs the three write actions by sequentially calling private helper methods, `addCustomer`, `addOrder`, and `addOrderedItems`. We'll also implement the three helper methods in the class. To leverage EJB's container-managed transaction service, we only require two annotations. These are:

- `@TransactionManagement(TransactionManagementType.`*`CONTAINER`*`)` : Used to specify that any transactions occurring in the class are container-managed.
- `@TransactionAttribute(TransactionAttributeType.`*`REQUIRED`*`)` : Used on the method that invokes the transaction to specify that a new transaction should be created (if one does not already exist).



Because we are implementing the transaction within a larger context, we'll approach this exercise by dividing it into several easily-digestible tasks.

Examining the Project Snapshot

Begin by examining the project snapshot associated with this tutorial unit.

1. Open the project snapshot for this tutorial unit in the IDE. Click the Open Project ( ) button and use the wizard to navigate to the location on your computer where you downloaded the project. If you are proceeding from the previous tutorial unit, note that this project snapshot is identical to the state of the project after completing the previous unit, but with the following exceptions:

- The `confirmation.jsp` page is fully implemented.
- The `affablebean.css` stylesheet includes rules specific to the `confirmation.jsp` page implementation.

2. Run the project ( ) to ensure that it is properly configured with your database and application server.

   If you receive an error when running the project, revisit the setup instructions, which describe how to prepare the database and establish connectivity between the IDE, GlassFish, and MySQL.

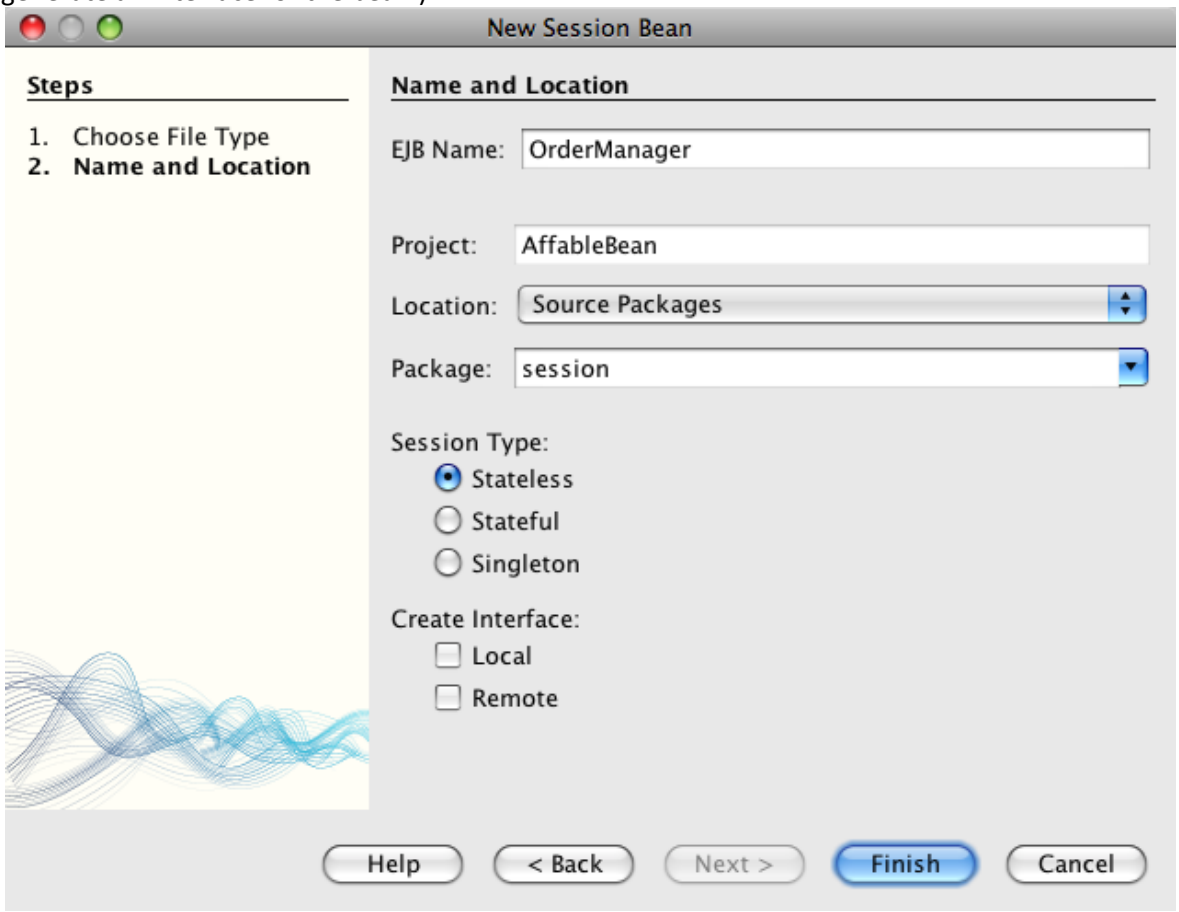
3. Test the application's functionality in your browser. In particular, step through the entire business process flow. When you click the submit an order from the checkout page, the confirmation page currently displays as follows:

No data related to the order is displayed on the confirmation page. In fact, in its current state the application doesn't do anything with the data from the checkout form. By the end of this tutorial unit, the application will gather customer data and use it to process an order. In its final state, the application will display a summary of the processed order on the confirmation page, remove the user's `ShoppingCart` and terminate the user session. (Snapshot 8 completes the request-response cycle when a checkout form is submitted.)

Creating the OrderManager EJB

1. Click the New File ( ) button in the IDE's toolbar. (Alternatively, press Ctrl-N; ⌘-N on Mac.) In the New File wizard, select the Java EE category, then select Session Bean.
2. Click Next. Name the EJB '`OrderManager`', place the EJB in the `session` package, and accept other default settings. (Create a stateless session bean, and do not have the wizard

generate an interface for the bean.)

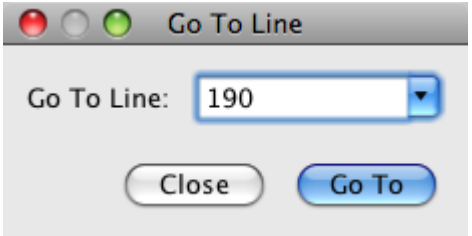


3. Click Finish. The new `OrderManager` class is generated and opens in the editor.

Handling Request Parameters

1. Open the project's `ControllerServlet`. (Either select it from the Projects window, or press Alt-Shift-O (Ctrl-Shift-O on Mac) and use the Go to File dialog.)
2. Locate the area in the `doPost` method where the `/purchase` request will be implemented (line 190).

   Press Ctrl-G to use the Go To Line dialog.



3. Implement code that extracts the parameters from a submitted checkout form. Locate the `TODO: Implement purchase action` comment, delete it, and add the following:

```
4. // if purchase action is called
```

```
} else if (userPath.equals("/purchase")) {

    if (cart != null) {

        // extract user data from request
          String name = request.getParameter("name");
          String email = request.getParameter("email");
          String phone = request.getParameter("phone");
          String address = request.getParameter("address");
          String cityRegion = request.getParameter("cityRegion");
          String ccNumber = request.getParameter("creditcard");
    }

    userPath = "/confirmation";
}
```

Implementing placeOrder and Helper Methods

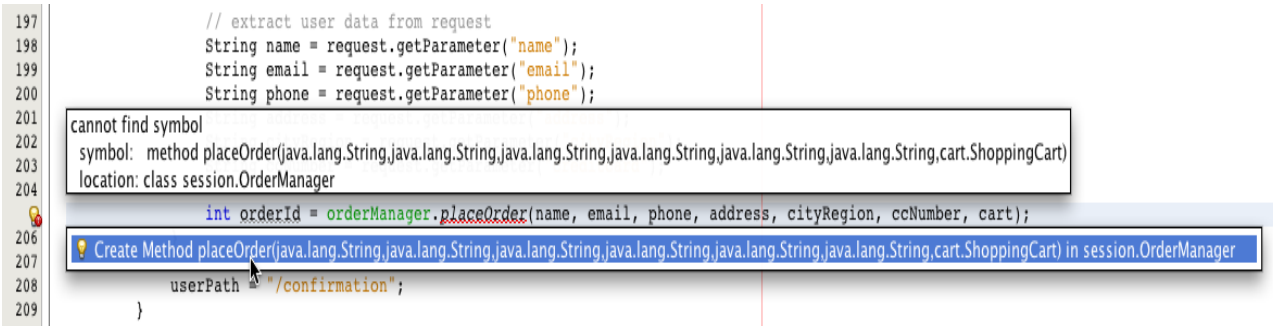
1. In the `ControllerServlet`, add a reference to the `OrderManager` EJB. Scroll to the top of the class and add a reference beneath the session facade EJBs that are already listed.

```
public class ControllerServlet extends HttpServlet {

    private String userPath;
    private String surcharge;
    private ShoppingCart cart;

    @EJB
    private CategoryFacade categoryFacade;
      @EJB
      private ProductFacade productFacade;
      @EJB
    private OrderManager orderManager;
```

13. Press Ctrl-Shift-I (⌘:-Shift-I on Mac) to allow the editor to add an import statement for `session.OrderManager`.
14. Use the extracted parameters, as well as the session `cart` object, as arguments for the `OrderManager.placeOrder` method. Add the following code:

```
 // if purchase action is called
 } else if (userPath.equals("/purchase")) {

     if (cart != null) {

         // extract user data from request
         String name = request.getParameter("name");
         String email = request.getParameter("email");
         String phone = request.getParameter("phone");
         String address = request.getParameter("address");
         String cityRegion = request.getParameter("cityRegion");
         String ccNumber = request.getParameter("creditcard");

         int orderId = orderManager.placeOrder(name, email, phone,
address, cityRegion, ccNumber, cart);
     }

     userPath = "/confirmation";
}
```

Note that we haven't created the `placeOrder` method yet. This is why the editor flags an error. You can use the tip that displays in the left margin, which allows you to generate the method signature in the appropriate class.



32. Click the tip. The IDE generates the `placeOrder` method in the `OrderManager` class.

```
33.  @Stateless
34.  public class OrderManager {
35.
36.      public int placeOrder(String name, String email, String phone,
     String address, String cityRegion, String ccNumber, ShoppingCart
     cart) {
37.          throw new UnsupportedOperationException("Not yet
     implemented");
38.      }
39.
40.      ...
     }
```

The import statement for `cart.ShoppingCart` is also automatically inserted at the top of the file.

41. In the new `placeOrder` method, use the method arguments to make calls to the (yet nonexistent) helper methods. Enter the following:

```
42.  public int placeOrder(String name, String email, String phone,
     String address, String cityRegion, String ccNumber, ShoppingCart
     cart) {
43.
44.      Customer customer = addCustomer(name, email, phone, address,
     cityRegion, ccNumber);
45.      CustomerOrder order = addOrder(customer, cart);
46.      addOrderedItems(order, cart);
     }
```

Note that we need to follow a particular order due to database constraints. For example, a `Customer` record needs to be created before the `CustomerOrder` record, since the `CustomerOrder` requires a reference to a `Customer`. Likewise, the `OrderedItem` records require a reference to an existing `CustomerOrder`.

47. Press Ctrl-Shift-I (⌘:-Shift-I on Mac) to fix imports. Import statements for `entity.Customer` and `entity.CustomerOrder` are automatically added to the top of the file.
48. Use the editor hints to have the IDE generate method signatures for `addCustomer`, `addOrder`, and `addOrderedItems`. After utilizing the three hints, the `OrderManager` class looks as follows.

```
@Stateless
public class OrderManager {

    public int placeOrder(String name, String email, String phone,
String address, String cityRegion, String ccNumber, ShoppingCart
cart) {

        Customer customer = addCustomer(name, email, phone,
address, cityRegion, ccNumber);
        CustomerOrder order = addOrder(customer, cart);
        addOrderedItems(order, cart);
    }

    private Customer addCustomer(String name, String email, String
phone, String address, String cityRegion, String ccNumber) {
        throw new UnsupportedOperationException("Not yet
implemented");
    }

    private CustomerOrder addOrder(Customer customer, ShoppingCart
cart) {
        throw new UnsupportedOperationException("Not yet
implemented");
    }

    private void addOrderedItems(CustomerOrder order, ShoppingCart
cart) {
        throw new UnsupportedOperationException("Not yet
implemented");
    }

}
```

Note that an error is still flagged in the editor, due to the fact that the method is currently lacking a return statement. The `placeOrder` signature indicates that the method returns an `int`. As will later be demonstrated, the method returns the order ID if it has been successfully processed, otherwise `0` is returned.

71. Enter the following return statement.

```
 public int placeOrder(String name, String email, String phone,
String address, String cityRegion, String ccNumber, ShoppingCart
cart) {

    Customer customer = addCustomer(name, email, phone, address,
cityRegion, ccNumber);
    CustomerOrder order = addOrder(customer, cart);
    addOrderedItems(order, cart);
    return order.getId();
}
```

At this stage, all errors in the `OrderManager` class are resolved.

78. Begin implementing the three helper methods. For now, simply add code that applies each method's input parameters to create new entity objects.

addCustomer

Create a new `Customer` object and return the object.

```
private Customer addCustomer(String name, String email, String phone,
String address, String cityRegion, String ccNumber) {

    Customer customer = new Customer();
    customer.setName(name);
    customer.setEmail(email);
    customer.setPhone(phone);
    customer.setAddress(address);
    customer.setCityRegion(cityRegion);
    customer.setCcNumber(ccNumber);

    return customer;
}
```

addOrder

Create a new `CustomerOrder` object and return the object. Use the `java.util.Random` class to generate a random confirmation number.

```
private CustomerOrder addOrder(Customer customer, ShoppingCart cart)
{

    // set up customer order
    CustomerOrder order = new CustomerOrder();
    order.setCustomer(customer);
    order.setAmount(BigDecimal.valueOf(cart.getTotal()));

    // create confirmation number
    Random random = new Random();
    int i = random.nextInt(999999999);
    order.setConfirmationNumber(i);

    return order;
}
```

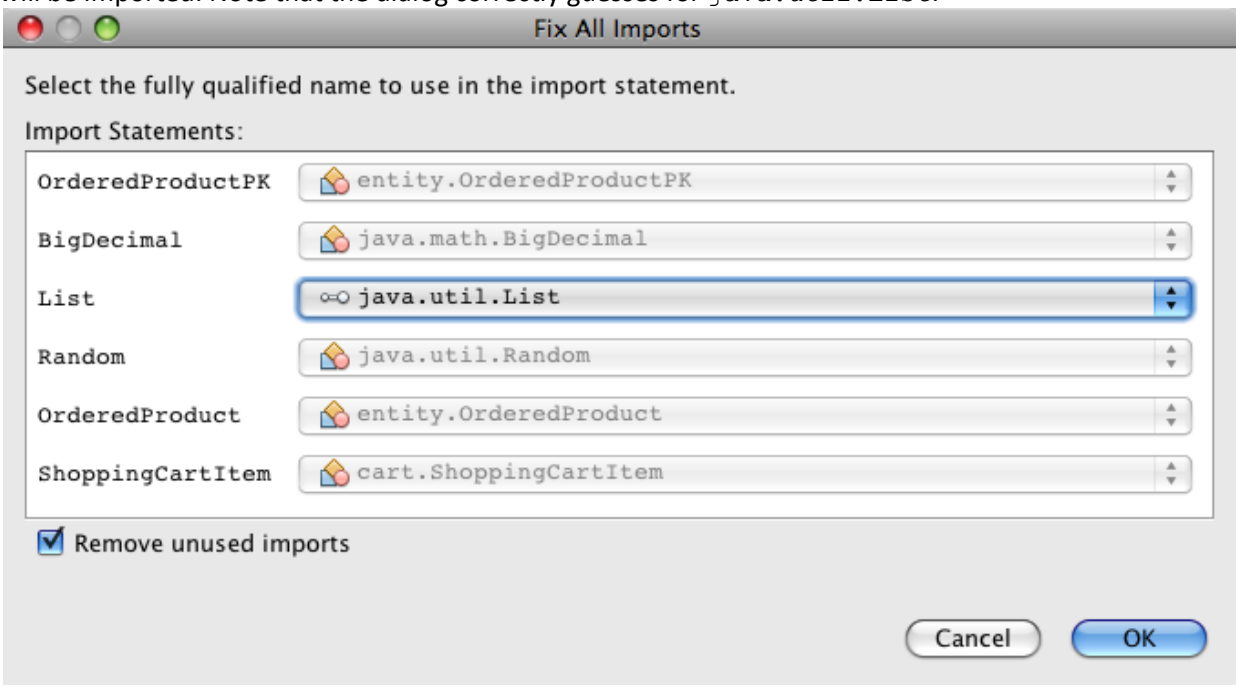
addOrderedItems

Iterate through the `ShoppingCart` and create `OrderedProduct`s. In order to create an `OrderedProduct`, you can use the `OrderedProductPK` entity class. The instantiated `OrderedProductPK` can be passed to the `OrderedProduct` constructor, as demonstrated below.

```
private void addOrderedItems(CustomerOrder order, ShoppingCart cart)
{

    List<ShoppingCartItem> items = cart.getItems();

    // iterate through shopping cart and create OrderedProducts
    for (ShoppingCartItem scItem : items) {
```

```
            int productId = scItem.getProduct().getId();

            // set up primary key object
            OrderedProductPK orderedProductPK = new OrderedProductPK();
            orderedProductPK.setCustomerOrderId(order.getId());
            orderedProductPK.setProductId(productId);

            // create ordered item using PK object
            OrderedProduct orderedItem = new
OrderedProduct(orderedProductPK);

            // set quantity
            orderedItem.setQuantity(scItem.getQuantity());
        }
}
```

79. Press Ctrl-Shift-I (⌘-Shift-I on Mac) to fix imports. A dialog opens to display all classes that will be imported. Note that the dialog correctly guesses for `java.util.List`.



80. Click OK. All necessary import statements are added, and the class becomes free of any compiler errors.

Utilizing JPA's EntityManager

As was mentioned in Adding Entity Classes and Session Beans, the `EntityManager` API is included in JPA, and is responsible for performing persistence operations on the database. In the `AffableBean` project, all of the EJBs employ the `EntityManager`. To demonstrate, open any of the session facade beans in the editor and note that the class uses the `@PersistenceContext` annotation to express a dependency on a container-managed `EntityManager` and its associated persistence context (`AffableBeanPU`, as specified in the `persistence.xml` file). For example, the `ProductFacade` bean looks as follows:

```
@Stateless
public class ProductFacade extends AbstractFacade<Product> {
    @PersistenceContext(unitName = "AffableBeanPU")
    private EntityManager em;

    protected EntityManager getEntityManager() {
        return em;
    }

    ...

    // manually created
    public List<Product> findForCategory(Category category) {
        return em.createQuery("SELECT p FROM Product p WHERE p.category =
:category").
                setParameter("category", category).getResultList();
    }

}
```

To be able to write to the database, the `OrderManager` EJB must take similar measures. With an `EntityManager` instance, we can then modify the helper methods (`addCustomer`, `addOrder`, `addOrderedItems`) so that the entity objects they create are written to the database.

1. In `OrderManager`, apply the `@PersistenceContext` annotation to express a dependency on a container-managed `EntityManager` and the `AffableBeanPU` persistence context. Also declare an `EntityManager` instance.

```
@Stateless
public class OrderManager {

    @PersistenceContext(unitName = "AffableBeanPU")
    private EntityManager em;

    ...
}
```

9. Press Ctrl-Shift-I (⌘:-Shift-I on Mac) to fix imports. Import statements for `javax.persistence.EntityManager` and `javax.persistence.PersistenceContext` are added to the top of the class.
10. Use the `EntityManager` to mark entity objects to be written to the database. This is accomplished using the `persist` method in the `EntityManager` API. Make the following modifications to the helper methods.

addCustomer

```
private Customer addCustomer(String name, String email, String phone,
String address, String cityRegion, String ccNumber) {

    Customer customer = new Customer();
    customer.setName(name);
    customer.setEmail(email);
    customer.setPhone(phone);
    customer.setAddress(address);
    customer.setCityRegion(cityRegion);
    customer.setCcNumber(ccNumber);
```

```
    em.persist(customer);
    return customer;
}
```

addOrder

```
private CustomerOrder addOrder(Customer customer, ShoppingCart cart)
{

    // set up customer order
    CustomerOrder order = new CustomerOrder();
    order.setCustomer(customer);
    order.setAmount(BigDecimal.valueOf(cart.getTotal()));

    // create confirmation number
    Random random = new Random();
    int i = random.nextInt(999999999);
    order.setConfirmationNumber(i);

    em.persist(order);
    return order;
}
```

addOrderedItems

```
private void addOrderedItems(CustomerOrder order, ShoppingCart cart)
{

    List<ShoppingCartItem> items = cart.getItems();

    // iterate through shopping cart and create OrderedProducts
    for (ShoppingCartItem scItem : items) {

        int productId = scItem.getProduct().getId();

        // set up primary key object
        OrderedProductPK orderedProductPK = new OrderedProductPK();
        orderedProductPK.setCustomerOrderId(order.getId());
        orderedProductPK.setProductId(productId);

        // create ordered item using PK object
        OrderedProduct orderedItem = new
OrderedProduct(orderedProductPK);

        // set quantity

orderedItem.setQuantity(String.valueOf(scItem.getQuantity()));

        em.persist(orderedItem);
    }
}
```

The `EntityManager`'s `persist` method does not immediately write the targeted object to the database. To describe this more accurately, the `persist` method places the object in

the *persistence context*. This means that the `EntityManager` takes on the responsibility of ensuring that the entity object is synchronized with the database. Think of the persistence context as an intermediate state used by the `EntityManager` to pass entities between the object realm and the relational realm (hence the term 'object-relational mapping').

What is the scope of the persistence context? If you open the IDE's Javadoc Index Search (Shift-F1; Shift-fn-F1 on Mac) and examine the Javadoc documentation for the `@PersistenceContext` annotation, you'll note that the `type` element is used to "specif[y] whether a transaction-scoped persistence context or an extended persistence context is to be used." A *transaction-scoped* persistence context is created at the start of a transaction, and terminated when the transaction ends. An *extended* persistence context applies to stateful session beans only, and spans multiple transactions. The Javadoc documentation also informs us that `javax.persistence.PersistenceContextType.TRANSACTION` is the default value for the `type` element. Therefore, although we didn't specify that the `EntityManager` place objects in a transaction-scoped persistence context, this is in fact how a container-managed `EntityManager` behaves by default.

Synchronizing the Persistence Context with the Database

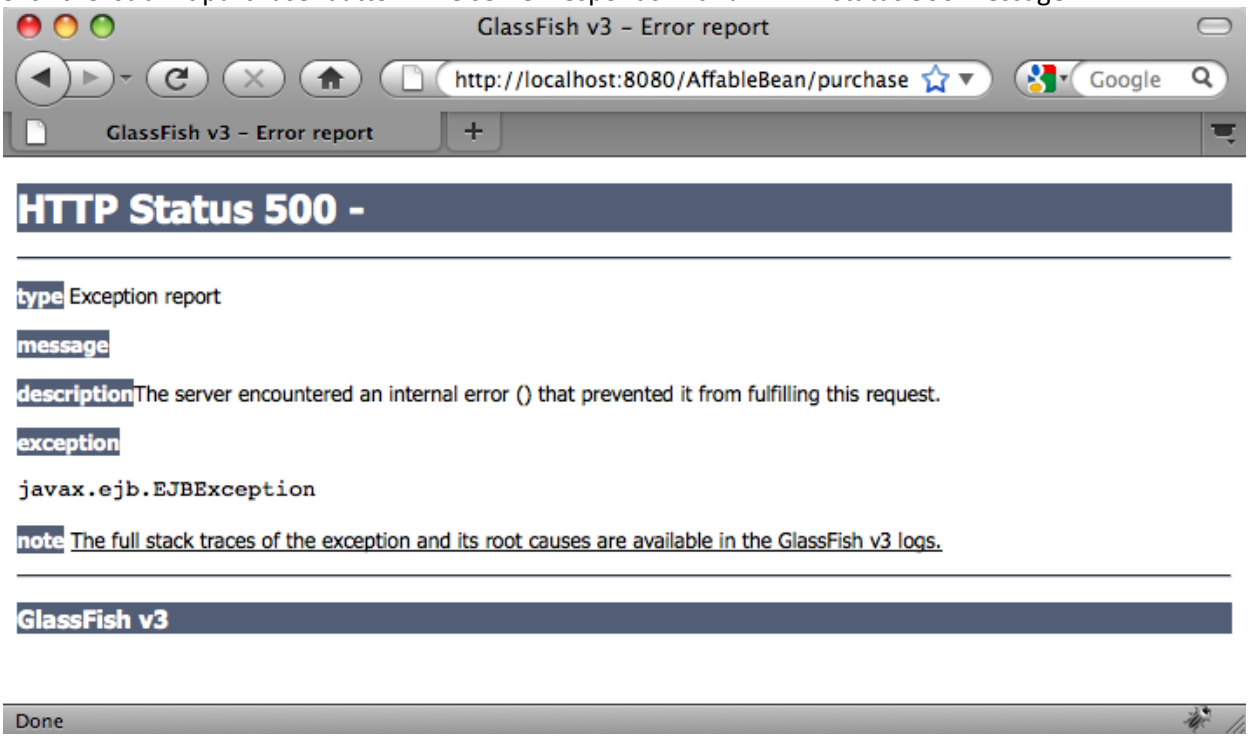
At this stage you might assume that, transaction or no transaction, the `OrderManager` is now able to successfully write entity objects to the database. Run the project and see how customer orders are currently being processed.

1. Press F6 (fn-F6 on Mac) to run the project.
2. Step through the business process flow. When you arrive at the checkout page, be sure to enter data that you know will not cause SQL errors to occur when the write actions are performed. (Validation is discussed in a later tutorial unit.) For example, enter the following into the checkout form:

- **name:** `Hugo Reyes`
- **email:** `hurley@mrcluck.com hurley @ mrcluck.com`
- **phone:** `606252924`
- **address:** `Karlova 33`
- **prague:** `1`
- **credit card number:** `1111222233334444`

In the coming steps, you are going to examine the server log in the IDE's Output window. Before submitting the checkout form, open the Output window and clear the server log. You can accomplish this by right-clicking in the server log and choosing Clear (Ctrl-L; ⌘-L on Mac).

3. Click the 'submit purchase' button. The server responds with an HTTP status 500 message.



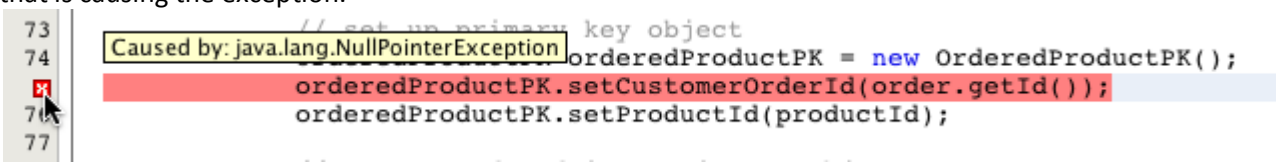
4. Switch to the IDE and examine the server log. The server log is located in the Output window (Ctrl-4; ⌘-4 on Mac) under the GlassFish server tab. You come across the following text.

```
5. WARNING: A system exception occurred during an invocation on EJB
   OrderManager method
6. public int
   session.OrderManager.placeOrder(java.lang.String,java.lang.String,jav
   a.lang.String,java.lang.String,java.lang.String,java.lang.String,cart
   .ShoppingCart)
7. javax.ejb.EJBException
8. ...
9. Caused by: java.lang.NullPointerException
10.         at
   session.OrderManager.addOrderedItems(OrderManager.java:75)
        at session.OrderManager.placeOrder(OrderManager.java:33)
```

Maximize the Output window by pressing Shift-Esc.

The underlines displayed in the server log form links allowing you to navigate directly to the lines in your source files where errors are occurring.

11. Click the link to `session.OrderManager.addOrderedItems`. The editor displays the line that is causing the exception.



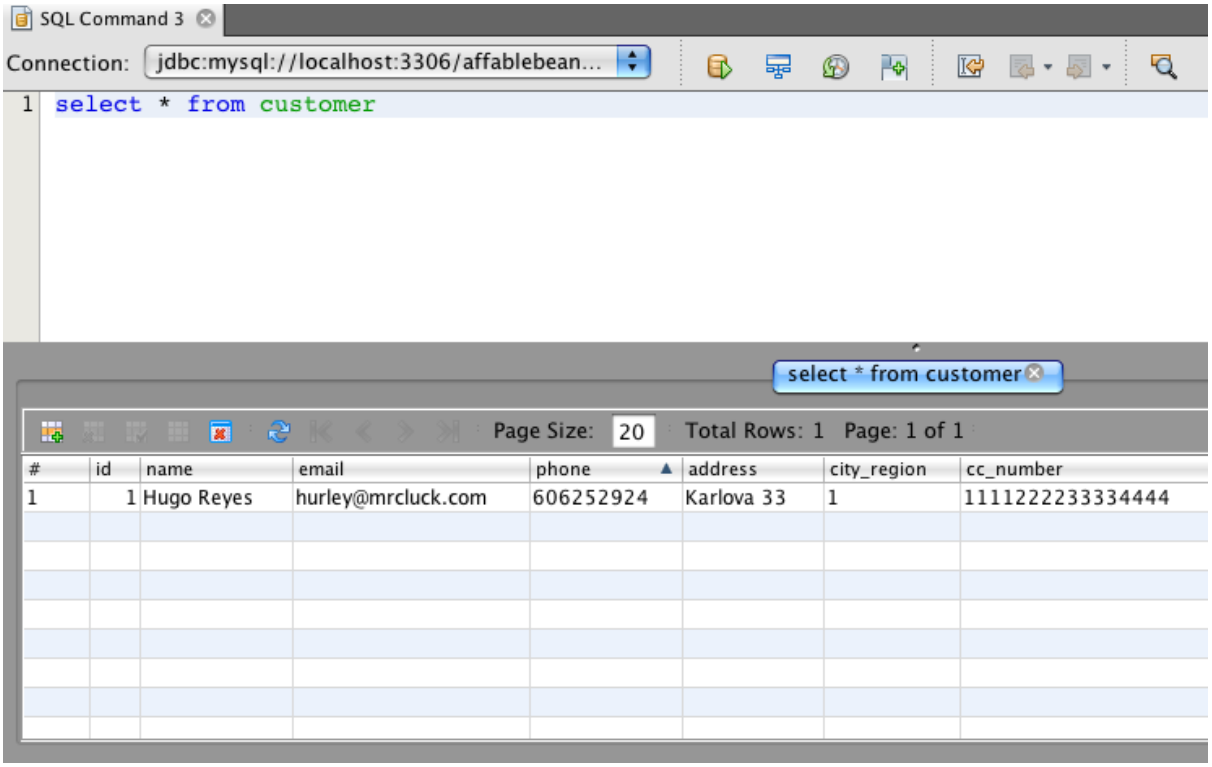
To understand why `order.getId` method returns `null`, consider what the code is actually trying to accomplish. The `getId` method attempts to get the ID of an order which is currently in the process of being created. Since the ID is an auto-incrementing primary key, the database automatically generates the value only when the record is added. One way to

overcome this is to manually synchronize the persistence context with the database. This can be accomplished using the `EntityManager`'s `flush` method.

12. In the `addOrderedItems` method, add a call to flush the persistence context to the database.

```
13.  private void addOrderedItems(CustomerOrder order, ShoppingCart
   cart) {
14.
15.      em.flush();
16.
17.      List<ShoppingCartItem> items = cart.getItems();
18.
19.      // iterate through shopping cart and create OrderedProducts
20.      for (ShoppingCartItem scItem : items) {
21.
22.          int productId = scItem.getProduct().getId();
23.
24.          // set up primary key object
25.          OrderedProductPK orderedProductPK = new OrderedProductPK();
26.          orderedProductPK.setCustomerOrderId(order.getId());
27.          orderedProductPK.setProductId(productId);
28.
29.          // create ordered item using PK object
30.          OrderedProduct orderedItem = new
   OrderedProduct(orderedProductPK);
31.
32.          // set quantity
33.
   orderedItem.setQuantity(String.valueOf(scItem.getQuantity()));
34.
35.          em.persist(orderedItem);
36.      }
   }
```

37. Rerun the project and step through the business process flow. This time, when you submit the checkout form the confirmation page displays.
38. To confirm that the details have been recorded in the database, open the IDE's Services window (Ctrl-5; ⌘-5 on Mac). Locate the `affablebean` connection node. If the node appears broken (), right-click the node and choose Connect.
39. Drill into the connection and locate the `affablebean` database's `customer` table. Right-click the table and choose View Data. A graphical display of the `customer` table appears in the editor. The customer details that you added in the checkout form display as a record in the table.



In this manner, you can also examine the `customer_order` and `ordered_product` tables to determine whether data has been recorded.

Setting up the Transaction Programmatically

A transaction's primary function is to ensure that all operations are performed successfully, and if not, then none of the individual operations are performed.[1] The following steps demonstrate how to ensure that the write operations performed in the `placeOrder` method are treated as a single transaction.

1. Refer to the transaction diagram above. Add the two transaction-related annotations to the `OrderManager` EJB.

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class OrderManager {

    @PersistenceContext(unitName = "AffableBeanPU")
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
     public int placeOrder(String name, String email, String phone,
String address, String cityRegion, String ccNumber, ShoppingCart
cart) {

        try {
           ...
```

The `@TransactionManagement` annotation is used to specify that any transactions occurring in the `OrderManager` EJB are container-managed. The `@TransactionAttribute` annotation placed on the `placeOrder` method specifies that any operations occurring in the method must be treated as part of a transaction.

According to the EJB Specification, container-managed transactions are enabled by default for session beans. Furthermore, if you examine the Javadoc for both of the above annotations, you will rightly point out that `CONTAINER` is the default `TransactionManagementType`, and `REQUIRED` is the default `TransactionAttributeType`. In other words, neither of the two annotations is required for your code to run properly. However, it is often helpful to explicitly include default settings in your sources to improve readability.

13. Currently, the `placeOrder` method returns the ID of the processed order. In the event that the transaction fails and the order isn't processed, have the method return '`0`'. Use a `try-catch` expression.

```
14.  @TransactionAttribute(TransactionAttributeType.REQUIRED)
15.  public int placeOrder(String name, String email, String phone,
   String address, String cityRegion, String ccNumber, ShoppingCart
   cart) {
16.
17.      try {
18.          Customer customer = addCustomer(name, email, phone,
   address, cityRegion, ccNumber);
19.          CustomerOrder order = addOrder(customer, cart);
20.          addOrderedItems(order, cart);
21.          return order.getId();
22.      } catch (Exception e) {
23.          return 0;
       }
```

NetBeans Support for Code Templates

When you work in the editor, take advantage of the IDE's support for code templates. Becoming proficient in using code templates ultimately enables you to work more efficiently and reliably.

For example, in the above step you can apply the `trycatch` template by typing in '`trycatch`' then pressing Tab. The template is added to your file.

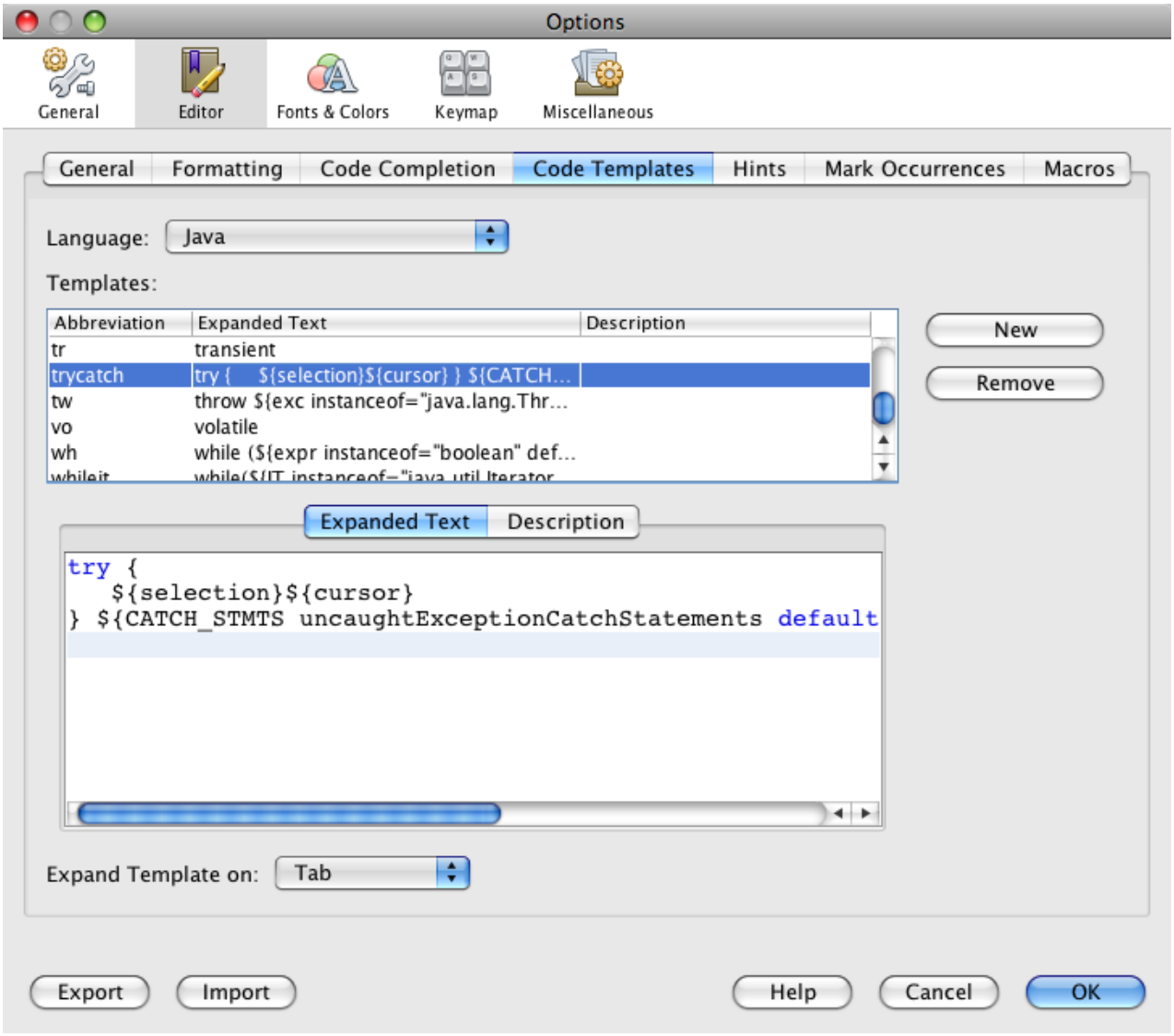
```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public int placeOrder(String name, String email, String phone, String
address, String cityRegion, String ccNumber, ShoppingCart cart) {

    try {

    } catch (Exception e) {
    }
    Customer customer = addCustomer(name, email, phone, address,
cityRegion, ccNumber);
    CustomerOrder order = addOrder(customer, cart);
    addOrderedItems(order, cart);
```

```
    return order.getId();
```

You can then move the four existing lines into the `try` clause by highlighting the lines, then holding Alt-Shift (Ctrl-Shift on Mac) and pressing the up arrow key. When you are finished, press F while holding Alt-Shift (Ctrl-Shift on Mac) to format the code.

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public int placeOrder(String name, String email, String phone, String
address, String cityRegion, String ccNumber, ShoppingCart cart) {

    try {
        Customer customer = addCustomer(name, email, phone, address,
cityRegion, ccNumber);
        CustomerOrder order = addOrder(customer, cart);
        addOrderedItems(order, cart);
        return order.getId();
    } catch (Exception e) {
    }
```

It is also possible to view and edit existing code templates, and add new templates in the IDE. Choose Tools > Options (NetBeans > Preferences on Mac) to open the Options window. Select Editor > Code Templates.



If you'd like to see more templates, consult the Keyboard Shortcuts Card. The Keyboard Shortcuts Card provides a list of commonly-used code templates and keyboard shortcuts. Choose Help > Keyboard Shortcuts Card from the main menu.

24. Add the following code. Explanation follows.

```
25.  @PersistenceContext(unitName = "AffableBeanPU")
26.  private EntityManager em;
27.  @Resource
28.  private SessionContext context;
29.
30.  @TransactionAttribute(TransactionAttributeType.REQUIRED)
31.  public int placeOrder(String name, String email, String phone,
   String address, String cityRegion, String ccNumber, ShoppingCart
   cart) {
32.
33.      try {
34.          Customer customer = addCustomer(name, email, phone,
   address, cityRegion, ccNumber);
```

```
35.             CustomerOrder order = addOrder(customer, cart);
36.             addOrderedItems(order, cart);
37.             return order.getId();
38.         } catch (Exception e) {
39.             context.setRollbackOnly();
40.             return 0;
41.         }
    }
```

Unfortunately, placing the three methods in the `try` clause means that if one of them fails during runtime, the engine immediately jumps to the `catch` clause, thus skipping any rollback operations that would normally follow.

You can test this by commenting out the `em.flush()` line you previously added. This way, you know that the first two methods (`addCustomer` and `addOrder`) process successfully, but the third method (`addOrderedItems`) fails. Run the project and submit the checkout form in the browser. Since the transaction doesn't roll back, the customer and order records are written to the database, but any ordered items are not. This leads to a situation where the database is corrupt.

To overcome this, you explicitly set the transaction for rollback in the `catch` clause. The above `@Resource` annotation is applied to grab an instance of the EJB's current `SessionContext`. The transaction is marked for rollback using the `setRollbackOnly` method.

42. Run the project and step through the business process flow. When you submit an order, return to the IDE and examine the server log. You'll see output similar to the following:

Press Shift-Esc on the Output window to maximize it.

As shown in the above image, the green text indicates output from EclipseLink. Recall how in Adding Entity Classes and Session Beans you set EclipseLink's logging level to `FINEST` in the persistence unit. Being able to examine this output is key to understanding how the

persistence provider interacts with the database and is a great help when you need to debug your project.

You've now successfully integrated the transaction into the `AffableBean` project. You can download snapshot 8 to examine code that completes the request-response cycle when a checkout form is submitted. The snapshot implements a `getOrderDetails` method in the `OrderManager`, which gathers all details pertaining to the placed order. If the transaction succeeds, the `ControllerServlet` places order details in the request scope, destroys the user's `cart` object, terminates the session, and forwards the request to the confirmation view. If the transaction fails, the `ControllerServlet` flags an error and forwards the response to the checkout view, enabling the user to attempt a resubmit.