The NetBeans E-commerce Tutorial - Conclusion

Tutorial Contents

Congratulations! You have now finished developing the `AffableBean` application. By following this tutorial, you incrementally built a simple e-commerce application using Java-based technologies. In the process, you became familiar with the NetBeans IDE, and have learned how to use it for the development of Java EE and web projects. Referring back to the customer requirements, you can confirm that each requirement has been fully implemented, and through continuous feedback from the Affable Bean staff, you are confident that they'll be satisfied with the final product. At this stage however, you may ask, "What specifically needs to be delivered to the customer?" and "How can the application become deployed to the customer's production server so that it functions online?" This tutorial unit briefly discusses next steps in terms of handing off deliverables, and concludes with a discussion on how using a framework such as JavaServer Faces could improve the application and benefit your experience when developing future projects.

You can view a live demo of the `AffableBean` application: NetBeans E-commerce Tutorial Demo Application.

The completed `AffableBean` project is also available for download.

Delivering your Work

When delivering your work, you should prepare both a WAR (web archive) file, which is a compiled, ready-to-deploy version of your project, and a source distribution, which contains all the source files you created during the development phase.

1. **WAR File Distribution:** A WAR file is basically a compressed collection of classes, files and other artifacts that constitute the web application. You can create a WAR file for your project using the IDE. In the Projects window, right-click your project node and choose Clean and Build. When your project is built, a WAR file is generated and placed in a `dist` folder in your project. You can verify this by examining your project in the Files window (Ctrl-2; ⌘-2 on Mac). (Refer back to Setting up the Development Environment).
2. **Source Distribution:** A package containing all source and configuration files, typically in an archive file format (e.g., ZIP, TAR). You can use your NetBeans project as part of your source distribution. Before compressing your project, make sure to clean it (In the Projects window, right-click the project node and choose Clean) in order to delete `build` and `dist` folders, if they exist. You should also remove any of your environment-specific details included in the project. To do so, navigate to the project on your computer's file system, then expand the project's `nbproject` folder and delete the `private` folder contained therein. (When the project is opened again in the IDE, the `private` folder and its files are regenerated according to the current environment.)

> As part of your source distribution, you would need to also provide any scripts or artifacts that are necessary for setup, configuration, and population of the database. In this scenario, that would include the MySQL Workbench project from Unit 4, Designing the Data Model, the DDL script that creates the `affablebean` database schema, and possibly a separate script that populates the `category` and `product` tables with business data.

As was indicated in the tutorial Scenario, a "technically-oriented staff member is able to deploy the application to the production server once it is ready." Aside from necessary performance tuning (GlassFish tuning is discussed in Unit 12 Testing and Profiling) the person responsible for this would need to ensure that the database driver is accessible to the server (i.e., place the driver JAR file in the server's library folder). He or she would also need to know the JNDI name of the data source used by the application to interact with the database. This is found in the persistence unit (`persistence.xml` file) and, as you may recall, is: `jdbc/affablebean`. This is the only "link" between the application itself and the back-end database server.

> **Note:** Recall that the `sun-resources.xml` file, which you created in Unit 6, Connecting the Application to the Database contains entries that instruct the GlassFish server to create the JDBC resource and connection pool when the application is deployed. The `sun-resources.xml` file is a deployment descriptor specific to the GlassFish server only. Therefore, if the customer isn't using GlassFish as the production server, the file should be deleted before the application is deployed. If the `sun-resources.xml` file isn't removed from the WAR distribution however, it would simply be ignored by the server it is deployed to.

In terms of security, it would be necessary to set up a *user* and *group* on the production server, so that the server can authenticate persons wanting to log into the administration console. Also, SSL support for the production server would need to be enabled, and you would need to acquire a certificate signed by a trusted third-party Certificate Authority (CA), such as VeriSign or Thawte.

Once the database is created and tables are populated with necessary data, the connection pool and JDBC resource are set up on the production server, and security measures have been taken, the application WAR file can be deployed to and launched on the production server. Using GlassFish, it is possible to deploy your applications via the Administration Console. (Select Applications in the left-hand Tree, then click the Deploy button to deploy a new application.)

> The GlassFish plugin support in NetBeans also enables you to connect to a remote instance of GlassFish. You can therefore work with a GlassFish production server from the IDE for monitoring, profiling, and debugging tasks. If you are interested in using GlassFish as a production server, refer to the See Also section below for a list of web hosting solutions.

Portability is among the key benefits of Java EE. As your application adheres to the technology specifications, it can theoretically be deployed to any server that supports the same specifications. Recall that the Introduction lists the specifications that you have used in this tutorial. All of these specifications are part of the Java EE 6 platform specification (JSR 316). Therefore, any server that is Java EE 6-compliant would be a candidate for running the `AffableBean` application.

Did you know?

The NetBeans IDE began as a student project (originally called Xelfi) at Charles University in Prague, Czech Republic in 1996. The goal was to write a Delphi-like Java IDE. Xelfi was the first Java IDE written in Java, with its first pre-releases in 1997.

NetBeans was later purchased by Sun Microsystems in 1999, and shortly thereafter became Sun's first sponsored open source project.

In June 2000, the initial netbeans.org website was launched. You can view an archived version of the site at: http://web.archive.org/web/20000815061212/https://netbeans.org/index.html

For more information, see A Brief History of NetBeans.

Using the JavaServer Faces Framework

Having developed a Java web application from scratch puts you in a great position to begin appreciating how a framework can benefit your work. This section briefly introduces the JavaServer Faces (JSF) framework, then examines the advantages of applying the framework to the `AffableBean` application.

What is the JavaServer Faces Framework?

The JavaServer Faces framework (JSR 314) is an integral part of the Java EE platform and aims to facilitate web development by providing the following:

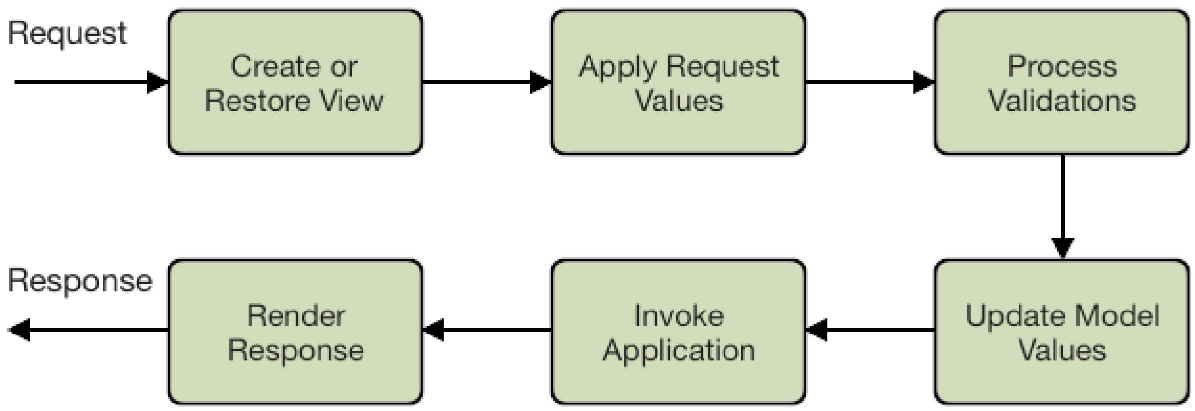
- **a user interface component model:** JSF includes a standard component API, which enables you to use and create custom UI components for your applications. A *UI component* is a widget that has a specific appearance and guarantees certain behavior. For example, this can be a simple text field that includes built-in data validation and conversion with accompanying error messages, or it can be a complex data table that interacts with a back-end data store and offers scrolling and column sorting for users. Being able to reuse UI components for your application's interface (or acquire custom components from third-party vendors) becomes increasingly important as your application grows in size and complexity.
- **an MVC development infrastructure:** The framework provides a `FacesServlet` which works behind the scenes to dispatch requests to their appropriate handlers (usually *backing beans* that you create). You author page views using Facelets, the default view handler technology for JSF 2.0. These features, when operating in tandem with JSF's *request processing lifecycle* (described below), encourage your work to adhere to the MVC paradigm.

The JSF framework manages the request-response cycle by automating events that typically need to occur for each client request. These events are qualified into six distinct phases that are together known as the *JSF request processing lifecycle*. The book, JavaServer Faces 2.0: The Complete Reference by Ed Burns and Chris Schalk, describes the lifecycle phases as follows:

> *[T]he request processing lifecycle performs all of the necessary back-end processing for which one would otherwise have to write his or her own code. The lifecycle directs the processing of incoming request parameters, and it manages a server-side set of UI components and synchronizes them to what the user sees in a client browser. It also satisfies follow-up requests for images, style sheets, scripts, and other kinds of resources required to complete the rendering of the UI.*[1]

The six lifecycle phases, according to JavaServer Faces 2.0, are defined as follows:

1. **Create or Restore View:** Restores or creates a server-side component tree (View) in memory to represent the UI information from a client.
2. **Apply Request Values:** Updates the server-side components with fresh data from the client.
3. **Process Validations:** Performs validation and data type conversion on the new data.
4. **Update Model Values:** Updates any server-side Model objects with new data.
5. **Invoke Application:** Invokes any application logic needed to fulfill the request and navigate to a new page if needed.
6. **Render Response:** Saves state and renders a response to the requesting client.[2]



One important concept of the JSF framework is the server-side UI component tree, or Faces *View*. This component tree is built and maintained in server memory for each client request, and is primarily associated with the first and last phases of the request processing lifecycle depicted above. Consequently, the application is able to maintain state between requests in a way that doesn't involve any manual coding on the part of the developer. In other words, the request processing lifecycle handles synchronization between the server-side View and that which is presented to the client. This enables you, the Java web developer, to focus on code that is specific to your business problem.

How Can JSF Benefit Your Project?

To understand JSF's benefits, let's take a second look at the `AffableBean` project and consider how the framework could be applied.

Strong Templating Support

Rather than creating your application page views in JSP pages, you'd be using Facelets technology instead.[3] Facelets is a first-rate templating technology that enables you to maximize markup reuse and reduce redundancy in your page views. Also, because Facelets pages use the `.xhtml` file extension, you are able prepare views using standard XHTML syntax.

In the `AffableBean` project, we took measures to reduce redundancy by factoring out the header and footer markup for all page views into separate JSP fragment files, and then included them in views by using the `<include-prelude>` and `<include-coda>` elements in the deployment descriptor. Aside from the header, the layouts for each of the application's five page views were unique. However, many websites maintain the same layout across multiple pages. This is where templating comes in especially handy.

With Facelets templating, you have more control over which portions of markup get displayed for individual page views. For example, you could create a template layout that is common to all page views, and insert view-specific content into the template to render your views. In this manner, you could specify a title for each page view. (Notice that in the `AffableBean` application, the title remains the same for all page views.)

No Need to Handle Incoming Request Parameters

Upon reexamining the `AffableBean`'s `ControllerServlet`, you can see that each time we implemented code for the supported URL patterns, it was necessary to manually extract user parameters using the `request`'s `getParameter` method. When working in JSF, you often create *backing beans*, which are Java classes that are conceptually bound to a specific page view. Parameters are automatically extracted from a request (during phase 2 of the request processing lifecycle), and set as properties for the backing bean. JSF also takes care of casting the `String` values of your request parameters into the types that you have defined for your backing bean properties. For example, if you have a property defined as an `int`, and your incoming request parameter is a `String` whose value is "`33`", JSF automatically converts the value to an `int` before storing it in the backing bean.

No Need to Programmatically Configure Navigation

In order to set up navigation, we followed a certain pattern when implementing the `ControllerServlet`: For each incoming request, the `getServletPath` method is called to determine the requested URL pattern. After logic related to the URL pattern is performed, a `RequestDispatcher` is attained, and the request is forwarded to the appropriate page view. In numerous cases, the appropriate page view is specified by hard-coding the path using the `userPath` variable.

None of this is necessary when using JSF - navigation is handled by the framework. Your job would be to either associate page views with URL patterns and any logical outcomes using a Faces configuration file, or take advantage of JSF 2.0's *implicit navigation* feature, which automatically forwards a request to a view that has the same name as the requested URL pattern.

Built-in Validation Support

JavaServer Faces provides built-in server-side validation support. In the `AffableBean` project, we created a `Validator` class and manually coded logic to perform all validation. Using JSF, server-side validation would automatically occur at phase 3 of the request processing lifecycle.

It would be worthwhile to take advantage of this validation for the `AffableBean` checkout form, however some preliminary steps would be in order. Specifically, the HTML markup for form elements would need to be replaced with comparable tags from JSF's Standard HTML Library. This step converts the form elements into JSF UI components, which we can then specify validation actions on using JSF's Core Library. To give an idea, the side-by-side comparison below demonstrates an adaptation of the checkout form's "name" field.

HTML Markup

```
<label for="name">name:</label>
<input type="text"
       id="name"
       size="30"
       maxlength="45"
       value="${param.name}" />

<c:if test="${!empty
nameError}">
    Value is required.
</c:if>
```

JSF HTML Tag Library

```
<h:outputLabel value="name: " for="name">
    <h:inputText id="name"
                 size="30"
                 maxlength="45"
                 required="true"
                 value="#{checkoutBean.name}"
/>
</h:outputLabel>

<h:message for="name" />
```

The `<h:outputLabel>` tag renders as an HTML `<label>` tag, whereas `<h:inputText>` renders as an `<input>` tag whose `type` is set to "`text`". Note the `required` attribute, which is set to `true` (shown in **bold**). This is all that's needed to ensure that the field is not left blank by the user. The `<h:message>` tag identifies the location where any validation error messages for the field should display. JSF's default error message for a field that requires user input is, "Value is required."

Continuing with the example, if we wanted to check whether input for the field hasn't exceeded 45 characters, we could apply the `<f:validateLength>` tag.

```
<h:outputLabel value="name: " for="name">
    <h:inputText id="name"
                 size="30"
                 maxlength="45"
                 required="true"
                 value="#{checkoutBean.name}">

         <f:validateLength maximum="45" />
    </h:inputText>
</h:outputLabel>

<h:message for="name" />
```

Well-Defined Division of Labor

As stated in the Java EE 6 Tutorial, "One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behavior and presentation for web applications." If you are working on a large project that involves a team of developers, the framework functions as a blueprint which allows team members to focus on different areas of development simultaneously. For example, front-end developers can implement page views using tags from JSF's HTML Library, while programmers responsible for implementing component logic and behavior can "plug their work into" existing HTML library tags.

Ability to Render the View with Other Markup Languages

Suppose that the Affable Bean staff commission you at a later point to prepare a mobile version of their site, so users can access it using a hand-held device. JSF APIs are a flexible rendering technology that enable you to attach multiple renderers to the component tree (i.e., View) of a JSF-enabled application. In other words, it is possible to create custom components that, for example, render HTML when requested by a browser, or WML when requested by a PDA.

Send Us Your Feedback

See Also

NetBeans Tutorials

Community-Contributed Extensions of E-commerce Tutorial

- Hierarchical Web Service Development with NetBeans IDE by Jayasurya Venug

JavaServer Faces

- JSF 2.0 Support in NetBeans IDE
- Introduction to JavaServer Faces 2.0
- Generating a JavaServer Faces 2.0 CRUD Application from a Database
- Scrum Toys - The JSF 2.0 Complete Sample Application

Contexts and Dependency Injection

- Getting Started with Contexts and Dependency Injection and JSF 2.0
- Working with Injection and Qualifiers in CDI
- Applying @Alternative Beans and Lifecycle Annotations
- Working with Events in CDI

JavaServer Faces

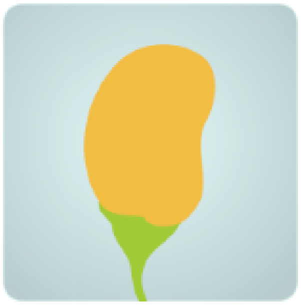
- **Product Page:** JavaServer Faces Technology
- **Specification Download:** JSR 314: JavaServer Faces 2.0
- **Reference Implementation:** GlassFish: Project Mojarra
- **Official Forum:** Web Tier APIs - JavaServer Faces
- The Java EE 6 Tutorial - Chapter 4: JavaServer Faces Technology

- The Java EE 6 Tutorial - Chapter 5: Introduction to Facelets
- JavaServer Faces 2.0: The Complete Reference [Book]
- Core JavaServer Faces [Book]
- JSF 2.0 Cookbook [Book]
- JSF 2.0 Refcard

GlassFish Web Hosting

- Joyent Cloud Hosting
- eApps Hosting
- Vision Web Hosting
- [DE]SYSTEMS
- JSPZone

About the NetBeans E-commerce Tutorial



The NetBeans E-commerce Tutorial and sample application were conceived of and written by Troy Giunipero. The application began as a project arising out of Sun's SEED program, and was developed from January 2009 to November 2010. The tutorial was prepared as part of ongoing efforts to provide documentation for the IDE's Java EE & Java Web Learning Trail.

Acknowledgments

Many people have helped with this project. I am especially grateful to following individuals for their help, support and contributions:

- Ed Burns, who was my SEED mentor, for his patience and guidance, and his willingness to share his technical expertise in our numerous discussions concerning Java web technologies.
- My managers, Patrick Keegan, for originally approving this project, and David Lindt, who showed continuous support.
- David Konecny and Andrei Badea for their invaluable help and advice, especially in regard Java Persistence, working with EclipseLink, and integrating EE 6 technologies.
- Don McKinney for providing the three beautiful diagrams used in Designing the Application.
- Eric Jendrock and the Java EE Tutorial team, for granting permission to adapt and reproduce diagrams from the Java EE 5 Tutorial. Diagrams were used in Securing the Application, and are based on Figure 28-6: Mapping Roles to Users and Groups and Figure 30-3: Form-Based Authentication.
- Jan Pirek, for coordinating and setting up necessary resources to make the live demo a reality.
- Ondrej Panek for providing a Czech translation of text used in the sample application.
- Also, special thanks to cobalt123 for graciously permitting usage of several photos, including Fresh Picks and Give Us Our Daily Bread #1.

Disclaimer

This tutorial and sample application are solely available for educative purposes. Although the sample application demonstrates a real-world scenario, there are several aspects that are decidedly not "real-world". For example, e-commerce sites do not typically store customer credit card details, but allow payment to be managed by a reputable third-party service, such as PayPal or WorldPay. Furthermore, although not discussed in the tutorial, customer trust is a hard-earned commodity. An e-commerce site's privacy policy, as well as the terms and conditions surrounding placed orders should be made easily available to customers and site visitors.

References

1. ^ Adapted from JavaServer Faces 2.0: The Complete Reference, Chapter 3: The JavaServer Faces Request Processing Lifecycle.
2. ^ Ibid.
3. ^ You can certainly use JavaServer Pages in a JSF application. Facelets is the default view handler technology for JSF version 2.0. For previous JSF versions, the default is JSP. In fact, when creating a new JSF 2.0 project in the IDE, you are able to specify the view technology you want to use (Facelets or JSP).