NOTES:

Java Web Applications with Java Server Faces

Java development is not just on the desktop alone. Thousands of enterprise applications are written using Java Enterprise Edition (Java EE), which enables the development of sophisticated, robust, and secure applications. The most mainstream and mature framework for developing Java EE applications is Java Server Faces (JSF).

JDK 9 can be used along with some Java EE application servers, such as GlassFish , to enable the use of the Java 9 features. Although Java EE and JSF are far too big to cover in one blog, this will provide you with a glimpse into the world of web development with Java 9 and Java EE.

Here, we will cover the basics of the Java Server Faces framework, from developing a basic application to creating a sophisticated front end. Throughout the process, we will cover important information such as how to correctly scope your controller classes, and also how to generate a web application template.

In the end, you will be able to get started developing Java web applications or maintain existing JSF projects.

Since web application development contains a number of interconnected processes, it is recommended to utilize an integrated development environment such as NetBeans to more easily organize web projects.

Throughout this doc, we will demonstrate the solutions to the recipes utilizing NetBeans IDE 8.2. However, you can apply these same basic concepts to projects using any number of Java IDEs.

Note: An early access release of the GlassFish 5 application server along with JDK 9.

To configure the server to utilize JDK 9,

```
modify the GlassFish <<GlassFish-Home>>/config/asenv.conf file

and add the AS_JAVA property, pointing to an installation of JDK 9.

Next, modify the <<GlassFish-Home>>/ bin/admin file to make the last line as follows:

exec "$JAVA" --add-modules java.annotations.common -jar "$AS_INSTALL_LIB/client/appserver-cli.jar" "$@"
```

Creating and Configure a Web Project

Problem: You would like to create and configure a simple Java web application project that will utilize the JSF web framework.

Solution

There are a number of different project formats that can be used to create a web application . One of the most flexible is the Maven web application format. The Apache Maven build system makes it easy to organize a build and expand the functionality of an application as time goes on since it contains a robust dependency management system.

In this solution, utilize NetBeans IDE to generate a Maven Web Application project , and then configure the project for developing a JSF application.

- **First, open NetBeans IDE and select "File," "New Project," and then from the New Project window,** choose the "Maven" category, and the "Web Application" project, then click "Next."

- **Name the application "HelloJsf," and place it into a directory on your hard disk.** Change the "Package Name" to org.java9recipes, and keep all of the other defaults

- **Next, select a server to which the application will be deployed, and a Java EE version.** In this case, I will utilize the Payara 5 server (GlassFish will also suffice), and Java EE 7.

- **After the project is created, right-click the project and choose "Properties" to configure it for JSF** and to assign a Java Platform. In the property menu, select the "Frameworks" category, then choose "Add" and select JSF.

- **Next, click the "Components" tab within the same window and select "PrimeFaces".**
- **Click "OK" to save the project properties, and the project is now ready to be built utilizing JSF** as the framework, along with the PrimeFaces UI library.

How It Works

Development of web applications requires orchestration of a number of different files. While it is possible to develop a Java EE web application without the use of an IDE, using a development environment makes it almost a trivial task.

In this recipe, the NetBeans IDE is used to configure a Maven based web application . Maven is a build system similar to Apache Ant, and it is very useful for organization of application projects. Maven is not necessarily better than Ant, but it is easier to get started using.

Both Ant and Maven are build systems; however, Maven uses convention over configuration, whereby it assumes many default configurations so that one can use very easily. Ant, on the other hand, requires one to configure and write a build script before it can be used.

One of the key components of Maven is that it makes dependency management very easy. It has become one of the most popular project formats, and developing a Maven project in NetBeans creates a project that is portable.

During the project creation wizard, a number of fields must be filled in, although many of the defaults can be left in place. Most importantly, set up a proper package naming convention for the application, and choose the server and Java EE version.

Note The settings that are completed when utilizing the wizard can be changed after the project has been created by going into the project properties.

Once the initial wizard has completed, a basic Maven web project will have been generated. At

Once the initial Wizard has completed, a basic Maven web project will have been generated. At this point, the project can be configured to utilize web frameworks, different versions of the JDK, and so on, by changing the project properties.

Right-click a NetBeans project to enter the project properties screen, and utilize the category selection to view or change properties pertaining to the selected category.

In this case, selecting the “Frameworks” category will allow you the ability to add a web framework, such as JSF. When a framework is added to the project, all plumbing and configuration for the framework are completed.

Also at this point when choosing JSF, select the “Components” tab on the Frameworks properties and add any other JSF libraries that will be in use. In this case, add “PrimeFaces” since the application developed in this blog will utilize the PrimeFaces component library.

Once frameworks have been configured, be sure to select the “Sources” category within the properties dialog and select the “Source/Binary Format” pertaining to the JDK version that will be used for coding the application.

Next, select the “Build”->”Compile” category within the properties dialog and ensure that the “Java Platform” select aligns with the one that has been chosen on the “Source/Binary Format” category.

Once these selections have been made, the configuration is complete. Choose “OK” in the project properties. The project will be altered to include new views (index.xhtml and welcomePrimefaces.xhtml).

The web.xml deployment descriptor will also be altered for JSF configuration . The welcome file will now point to index.xhtml, and the FacesServlet, a key component of the JSF framework, will be configured.

The web.xml Configuration for a JSF Application usually looks very similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.

w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
<context-param>
<param-name>javax.faces.PROJECT_STAGE</param-name>
<param-value>Development</param-value>
</context-param>
<servlet>
<servlet-name>Faces Servlet</servlet-name> <servlet-class>javax.faces.webapp.FacesServlet</servlet-class> <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

```
</servlet-mapping>
<session-config>
<session-timeout>
</session-timeout>
</session-config>
<welcome-file-list>
<welcome-file>faces/index.xhtml</welcome-file> </welcome-file-list>
</web-app>
```

At this point, right-click the NetBeans project and choose “Run.” This will cause the application to be compiled and deployed to the application server that was selected in the project properties or at project creation time.

That does it for the creation and configuration of a JSF project within NetBeans. In the next recipe, I will delve into the world of JSF as the HelloJsf application is modified to add some functionality.

Developing a JSF Application

Problem

You have created a Maven web project configured with JSF, and you wish to add functionality to the application.

Solution

Build the application such that it will contain an HTML form with a number of fields to populate. The form, when submitted, will invoke a controller method. First, create a Java class that will be used as a container to hold the data that is submitted in the form.

Create a new Java class in a package named org.java9recipes.hellojsf.model, and name it User. In the class, create three private fields of type String for now: firstName, lastName, and email.

Next, generate accessor methods (getters and setters) for these fields by right-clicking in the file and choosing “Refactor->Encapsulate Fields” from the contextual menu.

This will open the “Encapsulate Fields” dialog, in which you should select all fields for creation of accessor methods and click “Refactor”.

Next, create the Contexts and Dependency Injection (CDI) managed bean. Right-click the project’s “Source Packages” node, and create a new package named org.java9recipes.hellojsf.jsf, which will be used to package all of the managed bean controller classes for the application.

Next, create a new Java class in the new package named HelloJsfController and make the class implement java.io.Serializable so that it can be passivation capable. Annotate the class with @ViewScoped to indicate that this bean will be managed in the view scope.

Also, annotate the class with @Named, which makes the controller class injectable and also allows one to reference the class from expression language within JSF views.

Next, create a private field of type User, name the field user and encapsulate fields to generate the accessor methods.

Within the getUser() method that is generated, perform a check to see if the user field is null, and if so, then instantiate a new User. At this point, the class should look as follows:

```
package org.java9recipes.hellojsf.jsf;
import javax.faces.view.ViewScoped;
import javax.inject.Named;
import org.java9recipes.hellojsf.model.User;
@Named
@ViewScoped
public class HelloJsfController implements java.io.Serializable {
private User user;
public User getUser() {
if(user == null){
user = new User();
}
return user;
}
public void setUser(User user) {
this.user = user;
}
```

```
}
```

Lastly, create a public method that has a void return type and names it createUser(). This method will be invoked when someone clicks the submit button on the form.

In the method, simply print a message to the screen to indicate that the user has been successfully created. To do this, obtain a handle on the current FacesContext instance, which pertains to the current session.

Once obtained, add a new FacesMessage to it by passing a null as the first parameter since the message will not be assigned to any single component, and pass the message as the second parameter. Finally, set the user object to null so that a new user object can be created.

Note FacesContext contains state information regarding a JSF request. The FacesContext is updated throughout the different phases of a JSF request processing lifecycle.

The method should look as follows once complete.

```
public void createUser(){
FacesContext context = FacesContext.getCurrentInstance(); context.addMessage(null, new FacesMessage("Successfully Added User: " +
user.getFirstName() + " " + user.getLastName()));
user = null;
}
Next, it is time to create the view. In this case, open the index.xhtml view file within NetBeans IDE, and add the HTML markup and JSF components that
will comprise the form.
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/ DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/jsf/html" xmlns:p="http://primefaces.org/ui">
<h:head>
<title>Facelet Title</title>
</h:head>
<h:body>
Hello from Facelets
<br />
<h:link outcome="welcomePrimefaces" value="Primefaces welcome page" />
<br/>
<h:form>
<p:messages id="messages"/>
<br/>
<p:outputLabel for="firstName" value="First: "/>
<p:inputText id="firstName" value="#{helloJsfController.user.firstName}"/> <br/>
<p:outputLabel for="lastName" value="Last: " />
<p:inputText id="lastName" value="#{helloJsfController.user.lastName}"/> <br/>
<p:outputLabel for="email" value="Email: " />
<p:inputText id="email" value="#{helloJsfController.user.email}"/> <br/><br/>
<p:commandButton id="submitUser" value="Submit" ajax="false" action="#{helloJsfController.createUser()}"/>
</h:form>
</h:body>
</html>
```

Once the view has been generated and the CDI controller has been created, the application can be built and ran by right-clicking the project and choosing “Run.”

How It Works

JSF was developed in 2004 by Sun Microsystems in an effort to help simplify web application development and make web applications easier to manage/support. It was an evolution of the JavaServer Pages (JSP) framework, adding a more organized development life cycle and the ability to more easily utilize modern web technologies.

JSF uses XML files in the XHTML format for view construction and Java classes for application logic, allowing it to adhere to the MVC architecture.

Every request in a JSF application is processed by the FacesServlet. The FacesServlet is responsible for building the component trees, processing events, determining navigation, and rendering responses.

JSF has now become a mature web framework and has many advantages over its previous renditions. There are also a large number of component and functional libraries that can be used to extend JSF applications.

The framework is very powerful, including easy integration with technologies such as Ajax and HTML5 making it effortless to develop dynamic content.

JSF works well with databases, using JDBC, Enterprise Java Bean (EJB), or RESTful technology to

work with the back end. JavaBeans, known as JSF managed beans, are used for application logic and support the dynamic content within each view.

They can adhere to different lifespans depending upon the scope that is used . Views can invoke methods within the beans to perform actions such as data manipulation and form processing.

Properties can also be declared within the beans and exposed within the views and evaluated utilizing a standard expression language, providing a convenient way to pass values to and from the server.

JSF allows developers to customize their applications with preexisting validation and conversion tags that can be used on components with the view. It is also easy to build custom validators, as well as custom components, that can be applied to components in a view.

In a nutshell, JSFs maturity makes it easy to develop just about any web application using the framework.

In this solution, a small application named HelloJsf is created. The application view contains a simple HTML form for submitting a few fields of data, a button for submitting the form to the back end, and a message component for displaying the response.

The controller class named UserController is ViewScoped, meaning that the scope of objects within the class will be retained for the life of the view. Once the user navigates to another view or closes the window, the objects are destroyed.

An object named User is used as a container for passing the user data around within the application, and the User is declared within the controller class and made available to the view via accessor methods.

```
private User user;
/**
@return the user
*/
public User getUser() {
if(user == null){
user = new User();
}
return user;
}
public void setUser(User user) {
this.user = user;
}
```

CDI controller classes contain the business logic for the views of a JSF application. In the solution, a class named HelloJsfController manages the processing and data for the HelloJsf application.

The controller is responsible for exposing fields and action methods to the JSF view such that data can be submitted directly into the fields and processed accordingly.

The controller also facilitates communication with the end user as messages can be created to clearly indicate if processing is successful or if issues have occurred, and the messages can be made available to the views.

The view for the application is an XHTML file, index.xhtml, and it includes an HTML form via the JSF <h:form> tags. At the top of the view, the required namespaces are imported so that PrimeFaces and JSF HTML components can be utilized. The form is composed of a number of HTML elements and JSF components.

The PrimeFaces components must be prefixed with “p” since the PrimeFaces namespace is assigned to that letter. Each of the JSF components contains a number of attributes that can be used to set values and configure the component’s behavior and functionality.

The message component <p: messages> is used to display messages that are made available via the FacesContext.

The p:outputLabel components render to an HTML labels, and the p:inputText components are rendered to HTML input elements of type text. The value attribute of the p:inputText components contains JSF expression language, referencing the HelloJsfController User object fields.

Finally, the p:commandButton component renders an HTML button (input element of type “submit”) to submit the form. The action attribute of the commandButton also utilizes JSF expression language to invoke the controller action method named createUser().

The ajax="false" attribute indicates that Ajax should not be used to process the form values asynchronously, but rather, the form should be submitted and refreshed.

This recipe packs a lot of information, but it demonstrates how easy it is to develop a simple JSF view with a managed controller class. In a real-life application, the data is likely stored in an RDBMS, such as Oracle or the like. The next recipe covers how to add a database and bind it to the application to store and retrieve user objects.

Developing a Model for Data

Problem: You would like to store data from a Java EE application within a relational database.

Solution

Bind the data within the application to Java objects so that the objects can be used to store and retrieve data from the database. In most cases, the Java Persistence API (JPA) is a suitable choice for working with data in the form of Java objects.

In the previous recipe, a JSF application was developed to submit User objects to a CDI controller. In this recipe, the data will be bound to an entity class and then stored/retrieved from a relational data store using JPA.

For the purposes of this recipe, the Apache Derby database will be utilized. First, create a database table to store the User objects. The following SQL can be used to generate the table, which includes a primary key field identified as ID.

```
CREATE TABLE HELLO_USER (
ID NUMERIC PRIMARY KEY,
FIRST_NAME VARCHAR(100),
LAST_NAME VARCHAR(50),
EMAIL VARCHAR(150));
```

Once the database table has been created, generate a corresponding entity class. For this solution, NetBeans IDE will be used to automatically create the class. To do so, right-click the “Source Packages” node of the HelloJsf project, and creating a package named org.java9recipes.hellojsf.entity.

Next, right-click the newly created package and select "New"->"Entity Classes from Database" from the contextual menu. Once the "New Entity Classes from Database" dialog appears, select or create a JDBC Data Source for your Apache Derby database.

Once selected, choose the USER table from the listing of available tables and add it to the "Selected Tables" list, then choose "Next." On the dialog screens that follow, accept all defaults and click through to "Finish" and create the entity class.

Once the entity class has been created, develop an EJB or JAX-RS RESTful web service class to work with the corresponding entity. In this solution, an EJB will be developed using NetBeans IDE by first creating another new package in the project named org.java9recipes.hellojsf.session.

This package will be used to hold the session beans or EJBs. Next, right-click the newly created package and select "Session Beans for Entity Classes" from the contextual menu. This will open the dialog which allows entity class(es) to be selected so that NetBeans IDE can automatically create the corresponding session beans.

Once selected, choose "Next," then finally select "Finish" to create the EJB . After doing so, NetBeans IDE will generate an abstract class entitled AbstractFacade, which will be extended by any entity class that is generated.

The NetBeans IDE will also generate the session bean, HelloJsfFacade. Once these classes have been generated, the model for the application is complete and the controller will be able to successfully work with the data.

How It Works

The model for an enterprise application is one of the most important components because data is at the heart of the enterprise. To generate a model for a Java EE application, one must have a data store, usually an RDBMS, and an object-relational mapping strategy must be coded to represent the database in a code format.

In this solution, the model is comprised of three classes: entity class, an abstract class containing standard object-relational mapping methods, and an EJB that extends the abstract class.

An entity class is essentially a Plain Old Java Object (POJO) that represents a database table as a Java object. The entity class has a field declared for each of the columns of the database table, and accessor methods are defined for each of the fields.

Annotations make entity classes work like magic, whereby a few easy annotations perform the task of binding the class, and subsequently the fields, to the database table and its columns. The @Entity annotation tells the compiler that this is an entity class.

Table Common Entity Class Annotations

Annotation Description

@Entity Marks a class as an entity class.

@Table Maps the entity class to a database table.

@Id Denotes the primary key field of the entity class.

@XmlRootElement Maps class to an XML Element.

@NamedQueries List of @NamedQuery elements which map names to predefined queries.

@Embeddable Denotes an embedded class.

The entity class is mapped to a named database table by annotating it with @Table and specifying the name of the database table as an attribute. NetBeans IDE also adds a couple of more annotations to the entity class for convenience, those being @XmlRootElement and @NamedQueries.

The @XmlRootElement annotation associates an XML root element with the class, thereby making the entity class available with XML-based APIs, such as JAX-RS and JAXB.

The @NamedQueries annotation provides a number of named queries for the entity (one for each field), making it easy to query the entity class by name, rather than writing JPQL each time it needs to be queried.

Entity classes also always contain a primary key, which is denoted via the @Id annotation, and each column of the database table are mapped to the class fields with @ Column.

Bean validation can also be added to the fields of an entity class, providing validation for any input or content that is added to the associated entity class field. Lastly, an entity class contains an equals() method to help compare objects against entities, and a toString() method. The final entity class for HelloUser should look as follows:

```
@Entity
@Table(name = "HELLO_USER")
@XmlRootElement
@NamedQueries({
@NamedQuery(name = "HelloUser.findAll", query = "SELECT h FROM HelloUser h"), @NamedQuery(name = "HelloUser.findById", query = "SELECT h
FROM HelloUser h WHERE h.id = :id"),
@NamedQuery(name = "HelloUser.findByFirstName", query = "SELECT h FROM HelloUser h WHERE h.firstName = :firstName"),
@NamedQuery(name = "HelloUser.findByLastName", query = "SELECT h FROM HelloUser h WHERE h.lastName = :lastName"),
@NamedQuery(name = "HelloUser.findByEmail", query = "SELECT h FROM HelloUser h WHERE h.email = :email")})
public class HelloUser implements Serializable {
private static final long serialVersionUID = 1L; @Id
@Basic(optional = false)
@NotNull
@Column(name = "ID")
private Integer id;
@Size(max = 100)
@Column(name = "FIRST_NAME")
private String firstName;
@Size(max = 50)
@Column(name = "LAST_NAME")
private String lastName;
@Pattern(regexp="[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@ (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?",
message="Invalid email")//if the field contains e-mail address consider using this annotation to enforce field validation
@Size(max = 150)
@Column(name = "EMAIL")
private String email;
public HelloUser() {
}
public HelloUser(Integer id) {
this.id = id;
}
public Integer getId() {
return id;
}
public void setId(Integer id) {
this.id = id;
}
```

```
public String getFirstName() {
return firstName;
}
public void setFirstName(String firstName) {
this.firstName = firstName;
}
public String getLastName() {
return lastName;
}
public void setLastName(String lastName) {
this.lastName = lastName;
}
public String getEmail() {
return email;
}
public void setEmail(String email) {
this.email = email;
}
@Override
public int hashCode() {
int hash = 0;
hash += (id != null ? id.hashCode() : 0);
return hash;
}
@Override
public boolean equals(Object object) {
TODO: Warning - this method won't work in the case the id fields are not set if (!(object instanceof HelloUser)) {
return false;
}
HelloUser other = (HelloUser) object;
if ((this.id == null && other.id != null) || (this.id != null && !this. id.equals(other.id))) {
return false;
}
return true;
}
@Override
public String toString() {
return "org.java9recipes.hellojsf.entity.HelloUser[ id=" + id + " ]";
}
}
```

Once an entity class has been generated, a session bean can be generated to facilitate work with the entity class. The session bean (a.k.a. EJB) declares a PersistenceContext, which provides communication with the underlying data store.

It then calls upon the PersistenceContext to perform any number of JPA tasks , such as creating, updating, or deleting records from a database via the entity class data. The NetBeans IDE generates the AbstractFacade abstract class, which is extended by all of the entity classes for the project.

This class essentially contains the methods that allow for basic manipulation of the entities : create(), findAll(), edit(), and remove(), enabling the developer to automatically gain access to such methods without recoding for each entity class. This leaves the developer with a fully functional session bean without any coding.

If additional queries or work against an entity needs to be created, the developer can modify the contents of the session bean, in this case, HelloJsfFacade, accordingly.

An EJB must be annotated with either @Stateful or @Stateless to designate whether the class will be a stateful or stateless session bean. A stateful session bean can be bound to a single user session,

allowing the state to be managed throughout that user's session.

Stateless is more often used such that the session bean will be shared across all of the user sessions in the application. The simple stateless session bean named HelloJsfFacade looks as follows.

```
@Stateless
public class HelloUserFacade extends AbstractFacade<HelloUser> {
@PersistenceContext(unitName = "org.java9recipes_HelloJsf_war_1.0-SNAPSHOTPU") private EntityManager em;
@Override
protected EntityManager getEntityManager() {
return em;
}
public HelloUserFacade() {
super(HelloUser.class);
}
}
```

The classes and code that has been discussed in this recipe constitute the model of an application.

In summary, the model binds the application to an underlying data store, thus making it possible to create, remove, update, and delete data using Java objects, rather than working directly with the database via SQL.

For more information on developing entity classes, please see the Java EE Tutorial online: https://docs. oracle.com/javaee/7/tutorial/.

Writing View Controllers

Problem

You have developed a JSF view which contains bound fields and a form, and you need to create the business logic to process the form and facilitate work with the session bean.

Solution

Create a managed bean controller class (CDI bean), which can be used to bind actions and fields to JSF views and facilitate work that needs to be performed within the EJB session bean. In this solution, the HelloJsfController class, seen in the following, is used as the CDI controller.

```
@Named
@ViewScoped
public class HelloJsfController implements java.io.Serializable {
private User user;
/**
@return the user
*/
public User getUser() {
if(user == null){
user = new User();
}
return user;
}
/**
@param user the user to set
*/
public void setUser(User user) {
this.user = user;
}
public void createUser(){
FacesContext context = FacesContext.getCurrentInstance();
context.addMessage(null, new FacesMessage("Successfully Added User: " + user.getFirstName() + " " + user.getLastName()));
user = null;
}
}
In the previous recipe, a data model was added to the HelloJsf application. Next, the controller class and view needs to be modified to make use of the da
ta model. To modify the controller, simply add a new private field of type HelloUser, and generate accessor methods for it. In the getHelloUser method, ch
eck first to see if the field is null, and if so, instantiate a new instance.
...
private HelloUser helloUser;
...
public HelloUser getHelloUser() {
if(helloUser == null){
helloUser = new HelloUser();
}
return helloUser;
}
public void setHelloUser(HelloUser helloUser) { this.helloUser = helloUser;
}
```

...
Next, inject the EJB into the controller class so that a new HelloUser can be persisted. To do so, inject a new private field of type HelloUserFacade as follows:

```
@EJB
private HelloUserFacade helloUserFacade;
```

Lastly, create a new action method named createAndPersistUser(), which will generally do the same as the createUser() method. However, this new method will persist a HelloUser object into the database by calling upon the EJB.

```
public void createAndPersistUser(){
FacesContext context = FacesContext.getCurrentInstance(); helloUserFacade.create(helloUser);
context.addMessage(null, new FacesMessage("Successfully Persisted User: " + user.getFirstName() + " " + user.getLastName()));
helloUser = null;
}
```

The data model has now been integrated into the controller logic. When a user clicks the button in the view, it should invoke the action method createAndPersistUser() within the controller. The fields contained within the form are processed via the controller as well since the User object was injected and exposed to the user interface.

How It Works

A JSF managed bean controller is used to facilitating work between the views and the session beans of a Java EE application. In the past, managed bean controllers used to adhere to a different set of rules, as JSF contained its own set of annotations for developing managed beans.

In recent releases of Java EE, JSF-specific managed beans have been phased out, and CDI beans have taken their place, allowing for a more cohesive and universal controller class.

In the solution, the class implements java.io Serializable since it may need to be persisted to disk in the event of the session ending abruptly.

The class is annotated with @Named to make it injectable and accessible via JSF expression language. The class is also annotated with a designated CDI scope, in this case, @ViewScoped, to indicate the CDI scope of the controller.

There are a number of different scopes, ViewScoped means that the controller state will be saved for the lifetime of the view.

The User object is declared within the controller as a private field, and it is made accessible as a property via the accessor methods. Lastly, the class contains a method named createUser(), which is public, and it creates a FacesMessage object and places it into the current FacesContext to display onscreen. The user object is then set to null.

The modified version of the controller class, which includes the data model, declares an instance field of type HelloUser.

Accessor methods for the HelloUser field are created, and within the getter method, a new instance is created if the field is null. The HelloUserFacade is injected into the controller so that it can be utilized to perform data model transactions (a.k.a.: database transactions).

The createAndPersistUser() method calls upon the HelloUserFacade create() method, passing a HelloUser instance to persist the object into the database. Similarly, if one wished to edit a HelloUser object, the HelloUserFacade edit() method can be invoked. Lastly, if one wishes to remove a user, the remove() method can be invoked.

A controller class may contain any number of action methods and field declarations , however, it is important to manage the size of a controller such that the controller is not responsible for performing too much work.

If a controller class contains too much functionality, for instance , if it is used to back more than one view, then it can become cumbersome and difficult to maintain. To learn more about CDI scopes for controller classes.

Developing Asynchronous Views

Problem

Rather than follow the old-style submit and response web application, you would like to generate a modern UI which will asynchronously submit data and post responses without refreshing the browser page or re-rendering the view to provide a better user experience.

Solution

Incorporate Asynchronous JavaScript and XML into your application to asynchronously send data to the server and render responses without refresh. There are a number of ways to create an AJAX-based view for a JSF application, and this recipe will demonstrate how to leverage the PrimeFaces AJAX API.

AJAX API.

In this solution, a new view will be created named helloAjax.xhtml, and it will generally be a copy of the original index.XHTML view which utilizes AJAX to submit the form.

The view will also asynchronously update the messages component, displaying the message that has been generated by the controller class.

A dataTable component is also added to helloAjax.xhtml, which is asynchronously updated to display the list of users that has been created and persisted to the database. The enhanced view looks as follows:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/ DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/jsf/html" xmlns:f="http://xmlns.jcp.org/jsf/core" xmlns:p="http://primefaces.org/u
i">
<h:head>
<title>Facelet Title</title>
</h:head>
<h:body>
Hello from Facelets
<br />
<h:link outcome="welcomePrimefaces" value="Primefaces welcome page" />
<br/>
<h:form>
<h:inputText id="firstNameType" value="#{helloJsfController.freeText}"> <f:ajax execute="@this" event="keyup" listener="#{helloJsfController. displayText
}" render="messages"/>
</h:inputText>
<p:messages id="messages"/>
<br/>
<p:panelGrid columns="2" style="width:100%">
<p:outputLabel for="firstName" value="First: "/>
<p:inputText id="firstName" value="#{helloJsfController.user.firstName}"/>
<p:outputLabel for="lastName" value="Last: " />
<p:inputText id="lastName" value="#{helloJsfController.user.lastName}"/>
<p:outputLabel for="email" value="Email: " />
<p:inputText id="email" value="#{helloJsfController.user.email}"/>
</p:panelGrid>
<br/>
<p:commandButton id="submitUser" value="Submit" action="#{helloJsfController.createUser()}" update="messages, helloUsers"/>
<br/>
<p:dataTable id="helloUsers" var="user" value="#{helloJsfController.

helloUserList}">
<p:column headerText="First Name">
<h:outputText value="#{user.firstName}"/>
</p:column>
<p:column headerText="Last Name">
<h:outputText value="#{user.lastName}"/>
</p:column>
<p:column headerText="Email">
<h:outputText value="#{user.email}"/>
</p:column>
</p:dataTable>
</h:form>
</h:body>
</html>
```

How It Works

It is very easy to apply the principles of AJAX to a JSF view. There are a few different ways to apply AJAX functionality, but the easiest is to utilize a sophisticated user interface framework, such as PrimeFaces, which includes the built-in AJAX functionality.

In fact, many of the PrimeFaces components perform AJAX submits by default, so they include an ajax attribute that can be set to false in order to operate in a synchronous manner.

In the solution to this recipe, a PrimeFaces commandButton is utilized to asynchronousl y send the form contents to the controller class. Once the action method is invoked, the data is persisted and a FacesMessage is generated, then the response is sent back to the view.

When the view receives the response, it asynchronously updates the components that are listed in the commandButton update attribute, those being the messages component, and the hello users dataTable component.

It is possible to asynchronously submit the contents of a JSF component without PrimeFaces by embedding the <f:ajax/> tag between the component's opening and closing tag.

The f: ajax tag makes use of an execute attribute to indicate which part of the view will be executed or submitted asynchronously, an event attribute to indicate which JavaScript event should invoke the asynchronous action, a listener attribute to bind an action method, amongst others. For example, the following inputText component has been made asynchronous via the use of the f:ajax tag:

```
<h:inputText id="firstNameType" value="#{helloJsfController.freeText}">

<f:ajax execute="@this" event="keyup" listener="#{helloJsfController.displayText}" render="messages"/>

</h:inputText>
```

In the example, when the keyup event occurs, the value that is typed within the inputText field is submitted to the helloJsfController.freeText property. The displayText() action method is also invoked, which places the contents of the freeText property into a FacesMessage, as seen in the following. Once the action is invoked and the request is sent back, the messages component is updated because the render attribute of f:ajax specifies its id.

```
public void displayText(AjaxBehaviorEvent evt){

FacesContext context = FacesContext.getCurrentInstance();

System.out.println("test: " + freeText);

context.addMessage(null, new FacesMessage(freeText));

}
```

There are a number of different techniques that can be used to asynchronously update JSF views. There are even more asynchronous components that are available amongst the many UI libraries that are available.

Although this solution demonstrates the use of PrimeFaces, as well as the f:ajax tag, small blogs could be written on the topic. JSF is a mature web framework, offering a plethora of tools to get the job done.

Choose which works best for the situation, and enjoy the ease of working with AJAX and limiting exposure to the underlying JavaScript.

Applying the Correct Scope

Problem

You are developing a JSF application, and you want to be sure that the controllers are configured to remain in scope for the correct amount of time, depending upon functionality and requirement.

Solution

Utilize CDI scopes to apply desired to scope to controller classes. For instance, if a controller class contains logic and data that is pertinent throughout the entire session, annotate the class with the javax. enterprise.context.SessionScoped.

However, if a controller class is only pertinent at the request level, annotate the class with javax.enterprise.context.RequestScoped. Apply each of the different scopes to the controller class(es) according to this logic.

How It Works

A controller class scope can change the way that application functions entirely. The amount of time in which a controller is in scope can make a big difference across the individual views of an application.

Fortunately, it is an easy task to apply different scopes to different controllers. However, programming methodology changes drastically depending upon the scope in which a controller class has been placed. CDI offers a number of scopes that can be utilized.

CDI Scopes

Scope Duration
@ApplicationScoped State is shared across all users' sessions within an application.
@Dependent Object receives the same lifecycle as a client bean. (Default scope)
@ConversationScoped Developer controls the start and end of the conversation, and the state is maintained throughout the entire conversation.
@RequestScoped State lasts for the duration of a single HTTP request.

@RequestScoped State lasts for the duration of a single HTTP request.

@SessionScoped State lasts for the duration of a user's session.

javax.faces.view.ViewScoped State lasts as long as NavigationHandler does not cause navigation to a different view.

As mentioned previously, one thing to keep in mind while applying scope is a controller's chosen scope will affect the rest of the application.

If a controller will be containing data that will be of use throughout a user's session, then @SessionScoped may be the best choice. Just keep in mind that all data within a @ SessionScoped bean will be retained throughout the session.

Therefore, if a List is declared and populated within the bean, the content of the bean must be refreshed or altered programmatically. Such is not the case if using some scope that causes a bean to be refreshed throughout the course of a user's session.

For instance, if the same bean is @RequestScoped, then the data in the List will be required and repopulated each time a request is made.

Note Scoping can also have a big impact on interaction with other managed beans. It is important to inject beans of the same scope

Generating and Applying a Template

Problem

You would like to apply the same visual template across all the views of an application.

Solution

Utilize a Facelets template and apply to each view. To create a template, you must first develop a new XHTML view file and then add the appropriate HTML/JSF/ XML markup to it.

Content from other views will displace the ui:insert elements in the template once the template has been applied to one or more JSF views. The following source is that of a template named template.xhtml this is the template that will be applied to all views within the HelloJsf application:

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://xmlns.jcp.org/jsf/facelets" xmlns:h="http://xmlns.jcp.org/jsf/html" xmlns:p="http://primefaces.org/ui">
<h:head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" /> <h:outputStylesheet library="css" name="default.css"/> <h:outputStylesheet library="css" name="cssLayout.css"/>
<title>Hello JSF</title>
</h:head>
<h:body>
<p:growl id="growl" life="3000" />
<p:layout fullPage="true">
<p:layoutUnit position="north" size="65" header="#{bundle.AppName}"> <h:form id="menuForm">
<p:menubar>
<p:menuitem value="Home" outcome="/index.xhtml" icon="ui-icon-home"/>
<p:menuitem value="Hello Main" outcome="/helloUser.xhtml" />
<p:menuitem value="PrimeFaces" outcome="/welcomePrimefaces.xhtml" /> <p:menuitem value="Hello Ajax" outcome="/helloAjax.xhtml" />
</p:menubar>
</h:form>
</p:layoutUnit>
<p:layoutUnit position="south" size="60">
<ui:insert name="footer"/>
</p:layoutUnit>
<p:layoutUnit position="center">
<ui:insert name="content"/>
</p:layoutUnit>
</p:layout>
</h:body>
</html>
```

The template defines the overall structure of the application views. However, it can use a CSS style sheet to declare the formatting for each of the elements within the template.

The style sheet should be contained within a resources directory in the application so that it will be accessible to the views. It is also possible to utilize JSF EL within a template.

If EL is utilized, typically a session or application scoped managed bean drives the content. A JSF client view of the template would contain <ui:composition/> tags surrounding the view content, and <ui:define/> tags surrounding the named segment of markup that belongs to the corresponding <ui:insert/> tags within the template. The following view would be an example of a client view of the template shown previously:

template shown previously.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/ DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://xmlns.jcp.org/jsf/html" xmlns:f="http://xmlns.jcp.org/jsf/core" xmlns:p="http://primefaces.org/ui" xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<ui:composition template="layout/template.xhtml">
<ui:define name="content">
Hello from Facelets
<br />
<h:link outcome="welcomePrimefaces" value="Primefaces welcome page" />
<br/>
<h:form>
<p:messages id="messages"/>
<br/>
<p:outputLabel for="firstName" value="First: "/>
<p:inputText id="firstName" value="#{helloJsfController.user.firstName}"/> <br/>
<p:outputLabel for="lastName" value="Last: " />
<p:inputText id="lastName" value="#{helloJsfController.user.lastName}"/> <br/>
<p:outputLabel for="email" value="Email: " />
<p:inputText id="email" value="#{helloJsfController.user.email}"/> <br/>
<p:commandButton id="submitUser" value="Submit" action="#{helloJsfController.createUser()}" update="messages, helloUsers"/>
<br/>
<p:dataTable id="helloUsers" var="user" value="#{helloJsfController.

helloUserList}">
<p:column headerText="First Name">
<h:outputText value="#{user.firstName}"/>
</p:column>
<p:column headerText="Last Name">
<h:outputText value="#{user.lastName}"/>
</p:column>
<p:column headerText="Email">
<h:outputText value="#{user.email}"/>
</p:column>
</p:dataTable>
</h:form>
</ui:define>
</ui:composition>
</html>
```

How It Works

To create a unified application experience, the views should be coherent in that they look similar and function in a uniform manner. The idea of developing web page templates has been around for a number of years, but unfortunately, many template implementations contain duplicate markup on every application page.

While duplicating the same layout for every separate web page works, it creates a maintenance nightmare. What happens when there is a need to update a single link within the page header?

Such a conundrum would cause a developer to visit and manually update every web page for an application if the template was duplicated on every page. The Facelets view definition language provides a robust solution for the development of view templates, and it is one of the major bonuses of working with the JSF technology.

Facelets provides the ability for a single template to be applied to one or more views within an application. This means a developer can create one view that constructs the header, footer, and other portions of the template, and then this view can be applied to any number of other views that are responsible for containing the main view content.

This technique mitigates issues such as changing a single link within the page header because now the template can be updated with the new link, and every other view within the application will automatically reflect the change.

To create a template using Facelets, create an XHTML view, declare the required namespaces, and then add HTML, JSF, and Facelets tags accordingly to design the layout you desire.

The template can be thought of as an “outer shell” for a web view , in that it can contain any number of other views within it. Likewise, any number of JSF views can have the same template applied, so the overall look and feel of the application will remain constant.

Facelets tags that are responsible for controlling the view layout. To utilize these Facelets tags, you'll need to declare the XML namespace for the Facelets tag library in the <html> element within the template. Note that the XML namespace for the standard JSF tag libraries is also specified here.

```
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://xmlns.jcp.org/jsf/facelets" xmlns:h="http://xmlns.jcp.org/jsf/html">
```

Facelets contains a number of special tags that can be used to help control page flow and layout. The table in the lists the Facelets tags that are useful for controlling page flow and layout. The only Facelets tag that is used within the template for this example is UI: insert.

The UI: insert tag contains a name attribute, which is set to the name of the corresponding UI: define an element that will be included in the view. Taking a look at the source for this example, you can see the following UI:

```
insert tag: <ui:insert name="content">Content</ui:insert>
```

Facelets Page Control and Template Tags

Tag Description

```
ui:component Defines a template component and specifies a file name for the component
ui:composition Defines a page composition and encapsulates all other JSF markup
ui:debug Creates a debug component, which captures debugging information, namely, the
state of the component tree and the scoped variables in the application, when the
component is rendered
ui:define Defines content that is inserted into a page by a template
ui:decorate Decorates pieces of a page
ui:fragment Defines a template fragment, much like ui:component, except that all content
outside of tag is not disregarded
ui:include Allows another XHTML page to be encapsulated and reused within a view
ui:insert Inserts content into a template
ui:param Passes parameters to an included file or template
```

ui:repeat Iterates over a collection of data
ui:remove Removes content from a page

If a view that uses the template, a.k.a. template client , it must list the template within the view <ui: composition> tag. Within the <ui: composition>, the view must specify a <ui: define> tag with the same name as the <ui: insert> name, then any content that is placed between the opening and closing <ui: define> tags will be inserted into the view in that location.

However, if the template client does not contain a <ui: define> tag with the same name as the <ui: insert> tag, then the content between the opening and closing <ui: insert> tags within the template will be displayed.

Nashorn and Scripting

In Java 6, the javax. script package was included for incorporating scripting languages with Java. It enabled developers to embed code written in scripting languages directly into Java applications.

This began a new generation of polyglot applications, as developers were able to construct Java solutions containing scripts written in languages such as JavaScript and Python.

The JavaScript engine that was used in Java 6 was called Rhino . It is an implementation of the JavaScript engine, developed entirely in Java. While it contains a full JavaScript implementation, it is an older engine and is no longer compliant with current JavaScript Standards.

Java 8 introduced a new JavaScript engine called Nashorn. It is based on the ECMAScript-262 Edition 5.1 language specification and supports the javax. script API introduced in Java 6.

Besides bringing a modern JavaScript engine to the Java platform, Nashorn also contains a few new features that make developing JavaScript and Java solutions easier and more robust.

The new command-line tool called JJS provides scripting abilities above and beyond those that were available with JRun Script. Nashorn also has full access to the JavaFX 8 API, allowing developers to construct JavaFX applications completely in JavaScript .

JDK 9 increases the usability of Nashorn even further by including a selected set of features from ECMAScript 6 specification at release. Over time, more features from ECMAScript 6 will likely be incorporated in updates of JDK 9 and subsequent releases of the JDK.

This blog touches on using the Nashorn engine to construct solutions that integrate the worlds of Java and JavaScript. It does not cover all of the features available with Nashorn, but you with provides enough to get up and running.

Loading and Executing JavaScript from Java

Problem

You want to load and execute JavaScript code from within your Java application.

Solution

Execute the JavaScript using the Nashorn engine, the next-generation JavaScript engine that is part of Java 8 and is used to execute JavaScript code. The Nashorn engine can be called upon to process in-line JavaScript or an external JavaScript file directly within Java code.

Execute an external JavaScript file or inline JavaScript code using the Java ScriptEngineManager. Once you've obtained a ScriptEngineManager(), you get an instance of the Nashorn engine to use for JavaScript code execution.

In the following example, a Nashorn ScriptEngine is used to invoke a JavaScript file that resides on the local file system.

```
public static void loadExternalJs(){ ScriptEngineManager sem = new ScriptEngineManager(); ScriptEngine nashorn = sem.getEngineByName("nashorn");
try {
nashorn.eval("load('class='lazy' data-src/org/java9recipes/blog18/js/helloNashorn.js')"); } catch (ScriptException ex) {
Logger.getLogger(NashornInvoker.class.getName()).log(Level.SEVERE, null, ex);
}
}
The code that resides in the helloNashorn.js file is as follows:
print("Hello Nashorn!");
Next, let's take a look at some in-line JavaScript. In the following example, a Nashorn ScriptEngine is obtained, and then a JavaScript function is created f
or obtaining the gallons of water for an in-ground pool. The function is then executed to return a result.
public static void loadInlineJs(){
ScriptEngineManager sem = new ScriptEngineManager();
ScriptEngine nashorn = sem.getEngineByName("nashorn");
try {
nashorn.eval("function gallons(width, length, avgDepth){var volume = avgDepth * width * length;" +
"return volume * 7.48; }");
nashorn.eval("print('Gallons of water in pool: '+ gallons(16,32,5))");
} catch (ScriptException ex) {
```

```
} catch (ScriptException ex) {
Logger.getLogger(NashornInvoker.class.getName()).log(Level.SEVERE, null, ex);
}
}
}
Results:
run:
Hello Nashorn!
Gallons of water in pool: 19148.800000000003
```

How It Works

There are a couple of different ways to use the Nashorn engine to execute JavaScript within a Java application. For example, Nashorn can be invoked from the command-line interface (CLI) named JJS, or the ScriptEngineManager can be used.

In this recipe, the example covers two such techniques for executing JavaScript with Nashorn, and each of them requires the use of the ScriptEngineManager, which has been part of the JDK since Java 6.

To obtain a Nashorn engine from the ScriptEngineManager, first, create a new instance of the ScriptEngineManager. Once obtained, you can obtain a particular engine by passing the String value that represents the desired engine to the getEngineByName() method.

In this case, you pass the name nashorn to obtain the Nashorn engine for working with JavaScript. After obtaining the Nashorn engine, you are ready to invoke a JavaScript file or evaluate inline JavaScript code by calling on the engine's eval() method.

The first code example in this recipe demonstrates how to pass a JavaScript file to the engine for invocation. The helloNashorn.js, in this case, contains a single line of JavaScript that prints a message without returning any results.

Perhaps the most difficult part of executing a .js file is that you must ensure that the file is contained in the classpath, or that you are passing the full path to the file to the eval() method.

The second code example demonstrates how to write and evaluate inline JavaScript . First, a function identified as gallons is defined and it accepts three parameters and returns the number of gallons based on the width, length, and average depth of a pool. In a subsequent eval() call, the function is invoked, passing parameters and returning a result.

The important point to note in this example is that although the JavaScript spanned multiple eval() calls, the scope is maintained so that each eval() call within the engine can see objects created within previous calls.

Since Java 6, it has been possible to work with scripting languages from within Java code . The Nashorn engine is obtained in the same manner as others, by passing a String to indicate the engine by name.

The difference between this JavaScript engine and the previous rendition Rhino is that the new JavaScript engine is much faster and provides better compliance with the EMCA-normalized JavaScript specification.

Since JDK 8 Update 40, some features from the updated EMCA Script 6 specification have been ported into Nashorn.

Since there are a large number of new features in the updated specification, they will be added over time through various releases of the JDK. JDK 9 introduces support for a significant subset of the ECMAScript 6 features.

Executing JavaScript via the Command Line

Problem: You want to execute JavaScript via the command line for prototyping or execution purposes

Solution 1

Invoke the JJS tool, which comes as part of Java. To execute a JavaScript file, invoke the JJS tool from the command line, and then pass the fully qualified name (path included if not in CLASSPATH) of a JavaScript file to execute.

For example, to execute helloNashorn.js, use the following command:

JJS /class='lazy' data-src/org/java9recipes/blog/js/helloNashorn.js Hello Nashorn!

To pass arguments to a JavaScript file for processing, call the script in the same manner, but include trailing dashes --, followed by the argument(s) you want to pass. For example, the following code resides within a file named helloParameter.js:

```
#! /usr/bin/env
var parameter = $ARG[0];
print(parameter ? "Hello ${parameter}!": "Hello Nashorn!");
Use the following command to invoke this JavaScript file, passing the parameter Oracle:
jjs /class='lazy' data-src/org/java9recipes/blog18/js/helloParameter.js – Oracle
Here is the result:
Hello Oracle!
```

The JJS tool can also be utilized as an interactive interpreter by simply executing JJS without any options. The command interpreter allows you to work in a fully interactive JavaScript environment.

In the following lines of code, the JJS tool is invoked to open a command shell, and a function is

In the following lines of code, the JJS tool is invoked to open a command shell, and a function is declared and executed. Finally, the command shell is exited.

```
JJS

JJS> function gallon(width, length, avgDepth){return (avgDepth * width * length) * 7.48;} function gallon(width, length, avgDepth){return (avgDepth * width * length) * 7.48;} JJS> gallon(16,32,5)

19148.800000000003

JJS> exit()
```

Solution 2

Make use of the JSR 223 JRun Script tool to execute JavaScript. To execute a JavaScript file, invoke the JRun Script tool from the command line and pass the fully qualified name (path included if not in CLASSPATH) of a JavaScript file to execute. For example, to execute helloNashorn.js, use the following command:

```
jrunscript /class='lazy' data-src/org/java9recipes/blog18/js/helloNashorn.js Hello Nashorn!

Perhaps you want to pass JavaScript code inline, rather than executing a JavaScript file. In this case, you would invoke jrunscript with the –e flag and pas s the script in-line.

JRun Script -e "print('Hello Nashorn')"

Hello Nashorn
```

Note String interpolation is not available if you’re using the JRun Script utility. Therefore, you must use concatenation to achieve a similar effect.

Similarly to JJS, the JRun Script tool also accepts arguments to pass to a JavaScript file for processing.

To pass arguments using the JRun Script tool, simply append them to the command when invoking the script, with each argument separated by spaces. For instance, to call the file helloParameter.js and pass an argument, execute the following command:

```
JRun Script class='lazy' data-src/org/java9recipes/blog18/js/helloParameter.js Oracle
```

Also similar to JJS, the JRun Script tool can execute an interactive interpreter, allowing you to develop and prototype on the fly as seen in the following illustration.

How It Works

Since the release of Java SE 6, it has been possible to work with scripting languages from Java. In this recipe, two solutions were demonstrated for executing JavaScript via the command line or terminal.

In Solution 1, you looked at the JJS command-line tool, which was new in Java 8. This tool can be used to invoke one or more JavaScript files, or to start an interactive Nashorn interpreter.

In the example, you took a look at how to invoke a JavaScript file with and without passing arguments. You also took a look at how to invoke JJS as an interactive interpreter.

The tool contains several useful options. To see an entire list, refer to the documentation online at http://docs.oracle.com/javase/9/docs/technotes/tools/windows/ JJS.html.

The JJS tool is the desired tool for use with Nashorn because it contains many more options than the JRun Script tool, which was demonstrated in Solution 2.

The JRun Script tool was introduced in Java 6 and it allows you to execute scripts from the command line or invoke an interactive interpreter, similar to JJS. The difference is that JRun Script also allows you to use other scripting languages by passing the –l flag, along with the scripting engine name.

```
JRun Script –l js myTest.js
```

The JRun Script tool also contains options, but it is limited in comparison to those available with JJS. To see all of the options available for JRun Script, refer to the online documentation at Moved javase/9/docs/technotes/tools/windows/JRun Script.html.

Embedding Expressions in Strings

Problem: You want to refer to expressions or values within a String when invoking JavaScript via the JJS utility.

Solution

When using Nashorn as a shell scripting language via the JJS tool, it is possible to embed expressions or values in Strings by enclosing them within dollar signs $ and curly brackets {} in a double-quoted String of text.

The following JavaScript resides in a file named recipe18_3.js, and it can be executed by the JJS tool as a shell script. The String interpolation works in this example because the script has been made executable by adding the shebang as the first line.

```
#! /usr/bin/env
function gallons(width, length, avgDepth){var volume = avgDepth * width * length; return volume * 7.48; }
print("Gallons of water in pool: ${gallons(16,32,5)}");
Execute the JavaScript file via jjs as follows:
jjs class='lazy' data-src/org/java9recipes/blog18/js/recipe18_3.js Gallons of water in pool: 19148.800000000003
```

Note This example JavaScript file cannot be run from a ScriptEngineManager because it contains a

shebang (it is an executable script).

How It Works

When you're using Nashorn's shell scripting features, you can embed expressions or values in double-quoted Strings of text by enclosing them in dollar signs and curly braces ${...}.

This concept is known a String interpolation in the Unix world, and Nashorn borrows the concept to make it easy to develop shell scripts for evaluating and displaying information.

String interpolation makes it possible to alter the contents of a String, replacing variables and expressions with values. Using this feature, it is easy to embed the contents of a variable in-line within a String without performing manual concatenation.

In the example for this recipe, a script that is stored within a .js file contains an embedded expression, and it calls on a JavaScript function to return the calculated number of liquid gallons.

This is likely the most useful technique for real-world scenarios, but it is also possible to make use of embedded expressions when using the JJS tool as an interactive interpreter.

```
jjs -scripting
JJS> "The current date is ${Date()}"
The current date is Wed Apr 30 2014 23:44:41 GMT-0500 (CDT)
```

Note If you’re not using the scripting features of JJS, String interpolation will not be available . Also, double quotes must be placed around the String of text, as Strings in single quotes are not interpolated.

In the example, the shebang (#! usr/bin/env) is used to make the script executable, thereby invoking the scripting features of JJS.

Passing Java Parameters

Problem: You want to pass Java parameters to JavaScript for use.

Solution

Utilize a javax.script.SimpleBindings instance to provide a String-based name for any Java field, and then pass the SimpleBindings instance to the JavaScript engine invocation. In the following example, a Java String parameter is passed to the Nashorn engine, and then it's printed via JavaScript.

```
String myJavaString = "This is a Java parameter!"; SimpleBindings simpleBindings = new SimpleBindings(); simpleBindings.put("myString", myJavaString ); ScriptEngineManager sem = new ScriptEngineManager(); ScriptEngine nashorn = sem.getEngineByName("nashorn"); nashorn.eval("print (myString)", simpleBindings);

Here is the result:

This is a Java parameter!
```

More than one Java type value can be passed in a SimpleBindings instance. In the following example, three float values are passed in a single SimpleBindings instance, and then they're passed to a JavaScript function.

```
float width = 16;
float length = 32;
float depth = 5;
SimpleBindings simpleBindings2 = new SimpleBindings(); simpleBindings2.put("globalWidth", width); simpleBindings2.put("globalLength", length); simple Bindings2.put("globalDepth", depth);
nashorn.eval("function gallons(width, length, avgDepth){var volume = avgDepth * width * length; "+
" return volume * 7.48; } " +
"print(gallons(globalWidth, globalLength, globalDepth));", simpleBindings2);
Result:
19148.800000000003
```

How It Works

To pass Java field values to JavaScript, use the javax.script.SimpleBindings construct, which is basically a HashMap that can be used for binding and passing values to the ScriptEngineManager.

When values are passed to the Nashorn engine in this manner, they can be accessed as global variables within the JavaScript engine.

Passing Return Values from JavaScript to Java

Problem: You want to invoke a JavaScript function and return the result to the Java class that invoked it.

Solution

Create a ScriptEngine for use with Nashorn and then pass the JavaScript function to it for evaluation. Next, create an Invocable from the engine and then call its invokeFunction() method, passing the String-based name of the JavaScript function, along with an array of the arguments to be used.

In the following example, a JavaScript function named gallons is passed to the ScriptEngine for evaluation, and it is later invoked using this technique. It then returns a double value.

```
ScriptEngineManager manager = new ScriptEngineManager(); ScriptEngine engine = manager.getEngineByName("nashorn");
// JavaScript code in a String
String gallonsFunction = "function gallons(width, length, avgDepth){var volume = avgDepth * width * length; "
+ " return volume * 7.48; } ";
try {
evaluate script engine.eval(gallonsFunction); double width = 16.0;
double length = 32.0; double depth = 5.0;
Invocable inv = (Invocable) engine;
double returnValue = (double) inv.invokeFunction("gallons",
new Double[]{width,length,depth});
System.out.println("The returned value:" + returnValue);
} catch (ScriptException | NoSuchMethodException ex) { Logger.getLogger(Recipe18_5.class.getName()).log(Level.SEVERE, null, ex);
}
Here's the result:
run:
The returned value:19148.800000000003
In the following example, a JavaScript file is invoked and returns a String value. The name of the
```

JavaScript file is recipe18_5.js and its contents are as follows:

```
function returnName( name){
return "Hello " + name;
}
```

Next, use the ScriptEngine to create an Invocable and call on the JavaScript function within the external JavaScript file.

```
engine.eval("load('/path-to/class='lazy' data-src/org/java9recipes/blog18/recipe18_05/js/recipe18_5.js')"); Invocable inv2 = (Invocable) engine;

String returnValue2 = (String) inv2.invokeFunction("returnName", new String[]{"Nashorn"}); System.out.println("The returned value:" + returnValue2);
```

How It Works

One of the most useful features of embedded scripting is the ability to integrate the code invoked via a script engine along with a Java application. In order to effectively integrate script engine code and Java code, the two must be able to pass values to each other. This recipe covers the concept of returning values from JavaScript back to Java.

To do so, set up a ScriptEngine and then coerce it into a javax.script.Invocable object . The Invocable object can then be used to execute script functions and methods, returning values from those invocations.

An Invocable object enables you to execute a named JavaScript function or method and return values to the caller. Invocable can also return an interface that will provide a way to invoke the member functions of the scripting object. To provide this functionality, the Invocable object contains several methods.

Before an Invocable can be generated, the JavaScript file or function must be evaluated by the ScriptEngine.

The example demonstrates calling on the eval() method to evaluate an in-line JavaScript function (a String named gallonsFunction), and it shows how to evaluate an external JavaScript file. Once the eval() method has been called, the ScriptEngine can be coerced into an Invocable object, as follows:

```
Invocable inv = (Invocable) engine;
```

Invocable can then be called upon to execute functions or methods within the evaluated script code.

In this recipe's examples, the invokeFunction method is used to call on the functions contained in the script. The first argument to invokeFunction is the String-based name of the function being called upon, and the second argument is a list of Objects that are being passed as arguments.

The Invocable returns an Object from the JavaScript function call, which can be coerced into the appropriate Java type.

Sharing values between Java and ScriptEngine instances is very useful. In a real-life scenario, it may be very useful to call on an external JavaScript file, and have the ability to pass values back and forth between the Java code and the script. The underlying JavaScript file can be modified, if needed, without recompiling the application.

This situation can be very useful when your application contains some business logic that needs to change from time to time. Imagine that you have a rules processor that can be used to evaluate Strings, and the rules are constantly evolving. In this case, the rule engine can be written as an external JavaScript file, enabling dynamic changes to that file.

Using Java Classes and Libraries

Problem: You want to call upon Java classes and libraries within your Nashorn solution.

Solution

Create JavaScript objects based on Java classes or libraries using the Java.type() function. Pass the fully qualified String-based name of the Java class that you want to utilize to this function and assign it to a variable. The following code represents a Java object named Employee, which will be utilized via a JavaScript file in this application.

```
package org.java9recipes.blog18.recipe18_06;
import java.util.Date;
public class Employee {
private int age;
private String first;
private String last;
private String position;
private Date hireDate;
public Employee(){
}
public Employee(String first,
String last,
Date hireDate){
this.first = first;
this.last = last;
this.hireDate = hireDate;
}
/**
@return the first
*/
public String getFirst() {
return first;
}
/**
@param first the first to set
*/
public void setFirst(String first) {
this.first = first;
}
/**
```

```
/**
@return the last
*/
public String getLast() {
return last;
}
/**
@param last the last to set
*/
public void setLast(String last) {
this.last = last;
}
...
}
```

Next, let's take a look the JavaScript file that makes use of the Employee class. This JavaScript code creates a couple of Employee instances and then prints them back out. It also uses the java.util.Date class to demonstrate using standard Java classes.

```
var oldDate = Java.type("java.util.Date");
var array = Java.type("java.util.ArrayList");
var emp = Java.type("org.java9recipes.blog18.recipe18_06.Employee");
var empArray = new array();
var emp1 = new emp("Josh", "steve", new oldDate());
var emp2 = new emp("Joe", "Blow", new oldDate());
empArray.add(emp1);
empArray.add(emp2);
empArray.forEach(function(value, index, ar){
print("Employee: " + value);
print("Hire Date: " + value.hireDate);
});
Lastly, you execute the JavaScript file using a ScriptEngineManager:
ScriptEngineManager sem = new ScriptEngineManager(); ScriptEngine nashorn = sem.getEngineByName("nashorn"); try {
nashorn.eval("load('/path-to/employeeFactory.js');"); } catch (ScriptException ex) {
Logger.getLogger(NashornInvoker.class.getName()).log(Level.SEVERE, null, ex);
}
Here are the results:
Employee: Josh steve
Hire Date: Thu April 24 23:03:53 CDT 2016
Employee: Joe Blow
Hire Date: Fri April 25 12:00:00 CDT 2016
```

How It Works

It is very natural to use Java classes and libraries from within a Nashorn solution. The example in this recipe demonstrates how to use a Java class that has been generated specifically for use with a custom application, as well as how to use Java classes and libraries that are part of Java SE.

In order to make such classes available to JavaScript, you must call the Java.type function from within the JavaScript and pass the String-based fully qualified name of the Java class to be used. The Java.type function returns a JavaScript reference to the Java type.

In the following excerpt from the example, the java.util.Date, java.util.ArrayList and Employee classes are made available to JavaScript using this technique.

```
var oldDate = Java.type("java.util.Date");

var array = Java.type("java.util.ArrayList");

var emp = Java.type("org.java9recipes.blog18.recipe18_06.Employee");
```

Once the types have been made available to JavaScript, they can be invoked in a similar

Once the types have been made available to JavaScript, they can be invoked in a similar manner to their Java counterparts. For instance, new oldDate() is used to instantiate a new instance of java.util.Date in the example.

An important difference is that you don't use getters and setters to call upon Java properties. Rather, you omit the "get" or "set" portion of the method and begin with a lowercase letter for the field name, thereby calling upon the fields directly.

This makes property access from within JavaScript quite easy and much more productive and readable. An example of such access can be seen from within the forEach loop in the script. To access the employee hireDate property, simply call employee.hireDate rather than employee.getHireDate().

The ability to access Java seamlessly from within JavaScript makes it possible to create seamless Java and JavaScript integrations.

Accessing Java Arrays and Collections in Nashorn

Problem

You need to gain access to a Java array or collection from within your Nashorn solution.

Solution

Use the Java.type function to coerce Java arrays to JavaScript. Once coerced, you instantiate the arrays by calling new and then accessing the members by specifying their index by number. In the following example, a Java int array type is created within JavaScript, and then it is instantiated and used for storage.

```
jjs> var intArray = Java.type("int[]");
jjs> var intArr = new intArray(5);
jjs> intArr[0] = 0;
0
jjs> intArr[1] = 1;
1
jjs> intArr[0]
0
jjs> intArr.length
5
```

Working with collections is quite similar. To access a Java Collection type, you call upon the Java. type function, passing the String-based name of the type you want to create. Once the type reference has been obtained, it can be instantiated and accessed from JavaScript.

```
jjs> var ArrayList = Java.type("java.util.ArrayList")
jjs> var array = new ArrayList();
jjs> array.add('hi');
true
jjs> array.add('bye');
true
jjs> array
[hi, bye]
jjs> var map = Java.type("java.util.HashMap")
jjs> var jsMap = new map();
jjs> jsMap.put(0, "first");
null
```

```
jjs> jsMap.put(1, "second");
null
jjs> jsMap.get(1);
second
```

How It Works

To make use of Java arrays and collections from within JavaScript, you invoke the Java.type() function and pass the name of the Java type that you want to access, assigning it to a JavaScript variable.

The JavaScript variable can then be instantiated and utilized in the same manner as the Java type would be used from within Java code. The examples in this recipe demonstrate how to access Java arrays, ArrayLists, and HashMaps from within JavaScript.

When working with a Java array type from JavaScript, the type of array must be passed to the Java.type() function, including an empty set of brackets. Once the type has been obtained and assigned to a JavaScript variable, it can be instantiated by including the static size of the array within brackets, just as an array would be instantiated in the Java language.

Similarly, the array can be accessed by specifying indices to assign and retrieve values from the array.

To go backward and pass a JavaScript array to Java, use the java.to.() function, passing the JavaScript array to its Java-type counterpart. In the following code, a JavaScript String array is coerced into a Java type.

```
jjs> var strArr = ["one","two","three"]
jjs> var javaStrArr = Java.type("java.lang.String[]");
jjs> var javaArray = Java.to(strArr, javaStrArr);
jjs> javaArray[1];
two
jjs> javaArray.class
class [Ljava.lang.String;
```

Collections are very similar to arrays, in that the Java.type() function must be used to obtain the Java type and assign it to a JavaScript variable. The variable is then instantiated and the Collection type is then accessed in the same manner as it would be in the Java language.

Implementing Java Interfaces

Problem: You want to make use of a Java interface from your Nashorn solution.

Solution

Create a new instance of the interface, passing a JavaScript object consisting of properties. The JavaScript object properties will implement the methods defined in the interface. In the following example, an interface used for declaring employee position types is implemented within

a JavaScript file.

The example demonstrates custom method implementation, as well as use of a default method. The following code is the interface, PositionType, which will be implemented in JavaScript.

```
import java.math.BigDecimal;
public interface PositionType {
public double hourlyWage(BigDecimal hours, BigDecimal wage);
/**
Hourly salary calculation
@param wage
@return
*/
public default BigDecimal yearlySalary(BigDecimal wage){
return (wage.multiply(new BigDecimal(40))).multiply(new BigDecimal(52));
}
}
Next, let's take a look at the code within the JavaScript file that implements the PositionType interface.
var somePosition = new org.java9recipes.blog18.recipe18_08.PositionType({ hourlyWage: function(hours, wage){
return hours * wage;
}
});
print(somePosition instanceof Java.type("org.java9recipes.blog18.recipe18_08. PositionType"));
var bigDecimal = Java.type("java.math.BigDecimal");
print(somePosition.hourlyWage(new bigDecimal(40), new bigDecimal(12.75)));
```

How It Works

Using a Java interface in JavaScript can be beneficial for creating objects that adhere to the implementation criteria. However, user interfaces in JavaScript is a bit different than using them in a Java solution.

For example, interfaces cannot be instantiated in Java. This is not the case when using them in JavaScript; you must actually instantiate an object of the interface type in order to use it.

The example demonstrates the implementation of an interface, PositionType, which is used for defining a number of methods within an employee position.

The methods are used for calculating an employee's hourly and yearly wage. To make use of the PositionType interface from JavaScript, the new keyword is used to instantiate an instance of that interface, assigning it to a JavaScript variable.

When instantiating the interface, a JavaScript object is passed to the constructor. The object contains implementations for each of the nondefault methods within the interface by identifying the name of the method, followed by the implementation.

In the example, there is only one method implemented on instantiation, and it is identified as hourlyWage(). If there had been more than one method implemented, the implementations would be separated by commas.

Although using Java interfaces is a bit different in JavaScript, they certainly provide a benefit.

In reality, they are performing the same task within JavaScript as they are within Java.

In Java, in order to implement an interface, you must create an object that implements it. You do the same thing within JavaScript, except that in order to create the implementing object, you must instantiate an instance of the interface.

Extending Java Classes

Problem

You want to extend a concrete Java class in your Nashorn JavaScript solution.

Solution

First obtain a reference to the Java class that is to be extended by calling the Java.type() function within your JavaScript file. Then create the subclass by calling on the Java.extend() function and passing the reference to the class that will be extended, along with a JavaScript object containing the implementations that will be altered.

The following code is that of the Employee class, which will later be extended from within a JavaScript file.

```
package org.java9recipes.blog18.recipe18_09;
import java.math.BigDecimal;
import java.util.Date;
public class Employee {
private int age;
private String first;
private String last;
private String position;
private Date hireDate;
...
public BigDecimal grossPay(BigDecimal hours, BigDecimal rate){ return hours.multiply(rate);
}
}
```

Here's the JavaScript code used to extend the class and use it:

```
var Employee = Java.type("org.java9recipes.blog18.recipe18_09.Employee"); var bigDecimal = Java.type("java.math.BigDecimal"); var Developer = Java.
extend(Employee, {
grossPay: function(hours, rate){
var bonus = 500;
return hours.multiply(rate).add(new bigDecimal(bonus));
}
});
var javaDev = new Developer();
javaDev.first = "Joe";
javaDev.last = "Dynamic";
print(javaDev + ""s gross pay for the week is: " + javaDev.grossPay(new bigDecimal(60), new bigDecimal(80)));
```

Here's the result:

```
Joe Dynamic's gross pay for the week is: 5300
```

To extend a standard Java class from within JavaScript, you call on the Java.extend() function, passing the Java class that you'd like to extend, along with a JavaScript object containing any fields or functions that will be altered in the subclass.

For the example in this recipe, a Java class entitled Employee is extended. However, the same technique can be used to extend any other Java interface, such as Runnable, Iterator, and so on.

In this example, to obtain the Employee class in JavaScript, the Java.type() function is called upon, passing the fully qualified class name. The object that is received from the call is stored in a JavaScript variable named Employee.

Next, the class is extended by calling on the Java.extend() function and passing the Employee class, along with a JavaScript object.

In the example, the JavaScript object that is sent to the Java.extend() function includes a different implementation of the Employee class grossPay() method. The object that is returned from the Java.extend() function is then instantiated and accessed via JavaScript.

Extending Java classes within JavaScript can be a very useful feature when you're working with a Nashorn solution. The ability to share objects from Java makes it possible to access existing Java solutions and build on them.