JSF: Managed Beans

Originals of Slides and Source Code for Examples:
http://www.coreservlets.com/JSF-Tutorial/

This somewhat old tutorial covers JSF 1, and is left online for those maintaining existing projects. All new projects should use JSF 2, which is both simpler and more powerful. See http://www.coreservlets.com/JSF-Tutorial/jsf2/.

Topics in This Section

- **Using beans to represent request parameters**
  - Data that came from the form submission
- **Using beans to store results data**
  - Data that came from the business logic
- **Referring to beans in input forms**
- **Outputting bean properties**
  - Standard JSF approach
  - JSP 2.0 expression language

Background: Beans

What Are Beans?

- **Java classes that follow certain conventions**
  - Must have a zero-argument (empty) constructor
    - You can satisfy this requirement either by explicitly defining such a constructor or by omitting all constructors
  - Should have no public instance variables (fields)
    - I hope you already follow this practice and use accessor methods instead of allowing direct access to fields
  - Persistent values should be accessed through methods called get*Xxx* and set*Xxx*
    - If class has method getTitle that returns a String, class is said to have a String *property* named title
    - Boolean properties use is*Xxx* instead of get*Xxx*
  - Unlike in Struts, JSF beans need extend no special class
    - In JSF world, these are sometimes called "backing beans"
      - Beans that represent the form (form parameters, action controller methods, event handling methods, placeholders for results data).

Why You Should Use Accessors, Not Public Fields

- **To be a bean, you cannot have public fields**
- **So, you should replace**

```
public double speed;
```

- **with**

```
private double speed;

public double getSpeed() {
  return(speed);
}
public void setSpeed(double newSpeed) {
  speed = newSpeed;
}
```

- **You should do this in *all* your Java code anyhow. Why?**

Why You Should Use Accessors, Not Public Fields

- You can put constraints on values

```
public void setSpeed(double newSpeed) {
  if (newSpeed < 0) {
     sendErrorMessage(...);
     newSpeed = Math.abs(newSpeed);
   }
   speed = newSpeed;
}
```

  – If users of your class accessed the fields directly, then they would each be responsible for checking constraints.

Why You Should Use Accessors, Not Public Fields

- You can change your internal representation without changing interface

```
// Now using metric units (kph, not mph)

public void setSpeed(double newSpeed) {
   speedInKPH = convert(newSpeed);
}

public void setSpeedInKPH(double newSpeed) {
   speedInKPH = newSpeed;
}
```

Why You Should Use Accessors, Not Public Fields

- **You can perform arbitrary side effects**

```
public double setSpeed(double newSpeed) {
  speed = newSpeed;
  updateSpeedometerDisplay();
}
```

  – If users of your class accessed the fields directly, then they would each be responsible for executing side effects. Too much work and runs huge risk of having display be inconsistent from actual values.

Beans Should Be Serializable
(If they will ever be session-scoped)

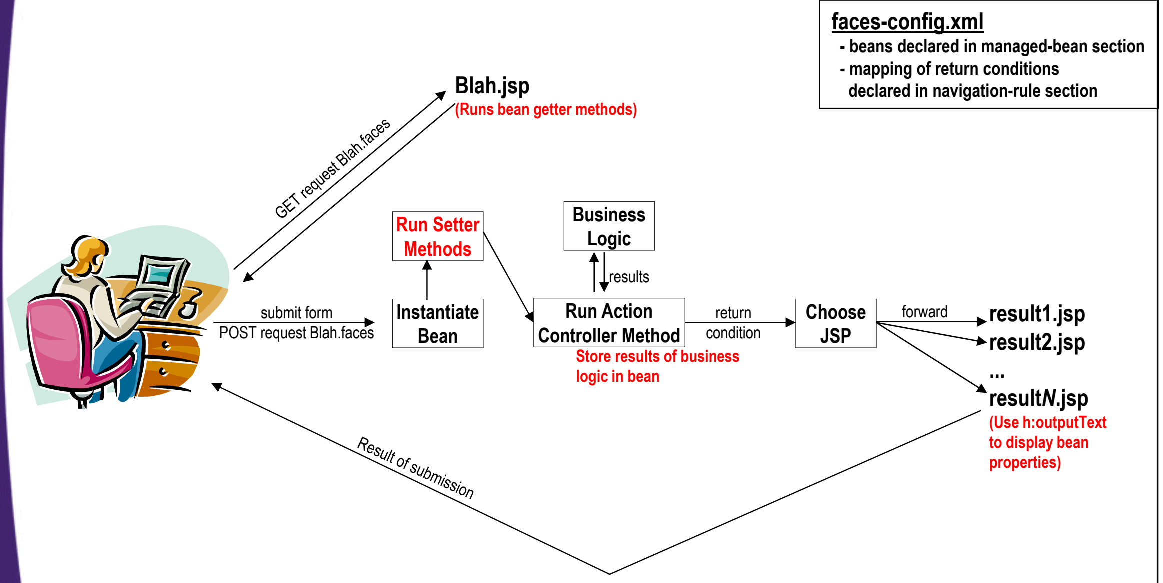
- **Some servers support distributed Web applications**
  – Load balancing used to send different requests to different machines. Sessions should still work even if different hosts are hit.
- **Some servers suport persistent sessions**
  – Session data written to disk and reloaded when server is restarted (as long as browser stays open).
    - Tomcat 5 and 6 support this
- **To support both, beans that will be session-scoped should implement the java.io.Serializable interface**
  – There are no methods in this interface; it is just a flag:

```
public class MyBean implements Serializable
   ...
}
```

  – Builtin classes like String and ArrayList are already Serializable

Updated Flow

JSF Flow of Control (Updated)


faces-config.xml
- beans declared in managed-bean section
- mapping of return conditions declared in navigation-rule section
Blah.jsp
(Runs bean getter methods)
GET request Blah.faces
submit form
POST request Blah.faces
Instantiate Bean
Run Setter Methods
Business Logic
results
Run Action Controller Method
Store results of business logic in bean
return condition
Choose JSP
forward
result1.jsp
result2.jsp
...
resultN.jsp
(Use h:outputText to display bean properties)
Result of submission

JSF Flow of Control (Simplified)

- **A form is displayed**
  - Form uses f:view and h:form
    - Bean instantiated†. If bean getter methods return non-empty, values filled in textfield
- **The form is submitted to itself**
  - Original URL and `ACTION` URL are http://…/*blah*.faces
- **A bean is instantiated†**
  - Listed in the managed-beans section of faces-config.xml
  - The setter methods given in h:inputText (etc.) are executed
    - Values passed to setter methods are the values in textfields when form is submitted
- **The action controller method is invoked**
  - Listed in the action attribute of h:commandButton
- **The action method returns a condition**
  - A string that matches from-outcome in the navigation rules in faces-config.xml
- **A results page is displayed**
  - Page uses h:outputText to output bean properties

†Assumes bean is request-scoped

Steps in Using JSF

1) Create a bean

A) Properties for form data

B) Action controller method

C) Placeholders for results data

2) Create an input form

A) Input fields refer to bean properties

B) Button specifies action controller method that will return condition

3) Edit faces-config.xml

A) Declare the bean

B) Specify navigation rules

4) Create results pages

– Output form data and results data with h:outputText

5) Prevent direct access to JSP pages

– Use a filter that redirects blah.jsp to blah.faces

Example

Example: Using Beans

- **Original URL:**
  - http://*hostname*/jsf-beans/register.faces
- **When form submitted, three possible results**
  - Error message re illegal email address
  - Error message re illegal password
  - Success
- **New features**
  - Action controller obtains request data from within bean
  - Output pages access bean properties
- **Main points**
  - Defining a bean with properties for the form data
  - Declaring beans in faces-config.xml
  - Outputting bean properties

Main Points of This Example

- **Add two new sections to the beans**
  - Properties (getter/setter pairs) for request parameters
  - Placeholders for results data
  - (Still have action controller method as before)

```
public class MyBean {
  public String getCustomerId() {…}
  public void setCustomerId(String id) {…}
  public String doBusinessLogic() {…}
  public String getBalance() {…}
}
```

Assume textfield refers to customerId property.

Assume balance is calculated by business logic based on the customer id.

- **Use h:inputText to associate textfield with property**

  `<h:inputText value="#{beanName.propertyName}"/>`
- **Use h:outputText to output bean properties**

  `<h:outputText value="#{beanName.propertyName}"/>`

Step 1: Create a Bean

(A) Properties for form data

- Pair of getter/setter methods for each request parameter
  - If input form says `value="#{name.foo}"`, then bean should have getFoo and setFoo methods.
- When form first displayed
  - Bean instantiated (assuming request scope)
  - Getter methods called (e.g., getFoo in above example)
  - If result is something other than null or empty String, value is placed into textfield
    - I.e., textfields are prepopulated with bean default values
- When form submitted
  - A new copy of the bean is instantiated
    - (assuming request scope)
  - Values from textfields passed to setter methods
    - E.g., setFoo in above example
  - Strings converted to other types as with jsp:setProperty
  - Form redisplayed if there are errors: see validation section

Step 1: Create a Bean (Continued)

(B) Action controller method

- Method can directly access bean properties
  - Since controller is inside same class that stores the request parameters
    - Different from Struts, where one object stores the request data (the form bean that extends ActionForm) and a different object has the controller (the class that extends Action and has execute)
- Method also invokes business logic, takes results, and stores them in placeholders reserved for output values

(C) Additional properties for output values

- Filled in by the action controller method

Step 1: Example Code

(1A) Form data

```
public class RegistrationBean implements Serializable {
  private String email = "user@host";
  private String password = "";

  public String getEmail() {
    return(email);
  }

  public void setEmail(String email) {
    this.email = email;
  }

  public String getPassword() {
    return(password);
  }
  public void setPassword(String password) {
    this.password = password;
  }
```

If you expect to ever make bean session-scoped

Step 1: Example Code

(1B) Action controller method

```
public String register() {
  if ((email == null) ||
      (email.trim().length() < 3) ||
      (email.indexOf("@") == -1)) {
    suggestion = SuggestionUtils.getSuggestionBean();
    return("bad-address");
  } else if ((password == null) ||
             (password.trim().length() < 6)) {
    suggestion = SuggestionUtils.getSuggestionBean();
    return("bad-password");
  } else {
    return("success");
  }
}
```

Step 1: Example Code

(1C) Placeholder for storing results

– Note that action controller method called business logic and placed the result in this placeholder

```
private SuggestionBean suggestion;

public SuggestionBean getSuggestion() {
  return(suggestion);
}
```

Step 1: Example Code (Result returned by business logic)

```
package coreservlets;
import java.io.*;

public class SuggestionBean implements Serializable {
  private String email;
  private String password;

  public SuggestionBean(String email, String password) {
    this.email = email;
    this.password = password;
  }

  public String getEmail() {
    return(email);
  }

  public String getPassword() {
    return(password);
  }
}
```

Step 1: Example Code (Business Logic)

```
package coreservlets;

public class SuggestionUtils {
  private static String[] suggestedAddresses =
    { "president@whitehouse.gov",
      "gates@microsoft.com",
      "palmisano@ibm.com",
      "ellison@oracle.com" };
  private static String chars =
    "abcdefghijklmnopqrstuvwxyz0123456789#@$%^&*?!";

  public static SuggestionBean getSuggestionBean() {
    String address = randomString(suggestedAddresses);
    String password = randomString(chars, 8);
    return(new SuggestionBean(address, password));
  }
  ...
}
```

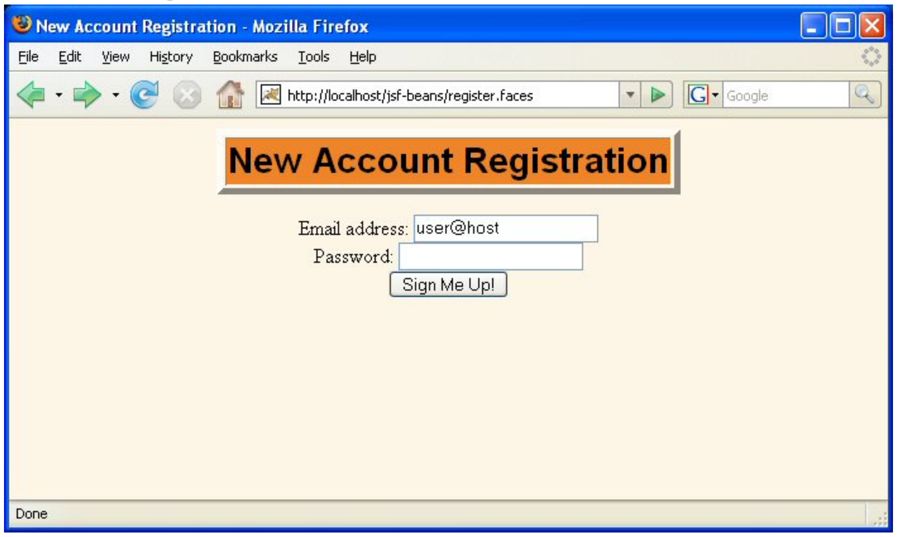
Step 2: Create Input Form

- **Similar to previous example, except**
  - h:input*Blah* tags given a `value` attribute identifying the corresponding bean property
- **Example code**

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>…
<h:form>
  Email address:
  <h:inputText value="#{registrationBean.email}"/><BR>
  Password:
  <h:inputSecret value="#{registrationBean.password}"/><BR>
  <h:commandButton value="Sign Me Up!"
                   action="#{registrationBean.register}"/>
</h:form>…
</f:view>
```

Step 2: Result

- **File is *tomcat_dir*/webapps/jsf-beans/register.jsp**
- **URL is http://localhost/jsf-beans/register.faces**
- **The user@host value comes from the bean**


New Account Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/jsf-beans/register.faces
Google
Done

New Account Registration

Email address: user@host
Password:
Sign Me Up!

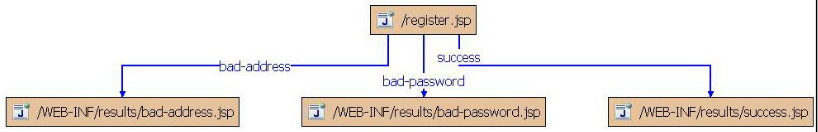
Step 3: Edit faces-config.xml

(A) Declare bean

```
…
<faces-config>
  <managed-bean>
    <managed-bean-name>
      registrationBean
    </managed-bean-name>
    <managed-bean-class>
      coreservlets.RegistrationBean
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
…
</faces-config>
```

Step 3: Edit faces-config.xml

- (B) Define navigation rules


/register.jsp
bad-address
success
bad-password
/WEB-INF/results/bad-address.jsp
/WEB-INF/results/bad-password.jsp
/WEB-INF/results/success.jsp

```
…
<faces-config>
  …
  <navigation-rule>
    <from-view-id>/register.jsp</from-view-id>
    <navigation-case>
      <from-outcome>bad-address</from-outcome>
      <to-view-id>/WEB-INF/results/bad-address.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>bad-password</from-outcome>
      <to-view-id>/WEB-INF/results/bad-password.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/WEB-INF/results/success.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Step 4: Create Results Pages

- Use h:outputText to access bean properties

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<!DOCTYPE …>
<HTML>
…
<h:outputText value="#{beanName.propertyName}"/>
…
</HTML>
</f:view>
```

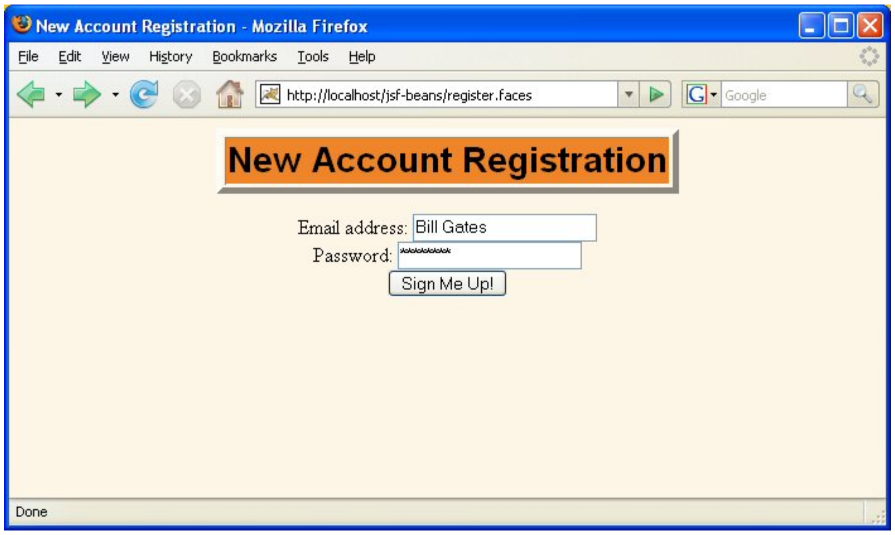
Step 4: Create Results Pages

- …/jsf-beans/WEB-INF/results/bad-address.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">Illegal Email Address</TH></TR>
</TABLE>
<P>
The address
"<h:outputText value="#{registrationBean.email}"/>"
is not of the form username@hostname (e.g.,
<h:outputText
     value="#{registrationBean.suggestion.email}"/>).
<P>
Please <A HREF="register.faces">try again</A>.
…
</HTML>
</f:view>
```

Step 4: Example Result for Bad Email Address

- **Input**


New Account Registration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/jsf-beans/register.faces
Google
Done

New Account Registration

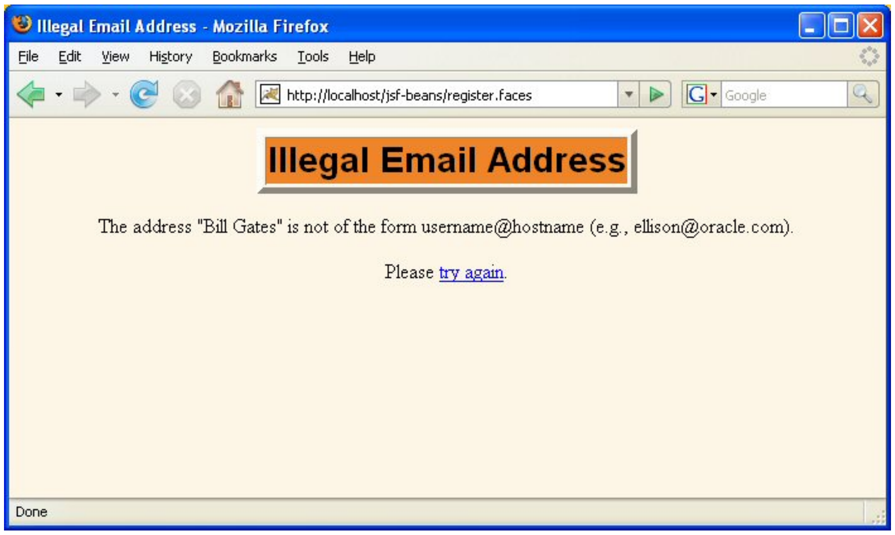
Email address: Bill Gates

Password: ********

Sign Me Up!

Step 4: Example Result for Bad Email Address

- **Output**


Illegal Email Address - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/jsf-beans/register.faces
Google
Done

Illegal Email Address

The address "Bill Gates" is not of the form username@hostname (e.g., ellison@oracle.com).

Please try again.

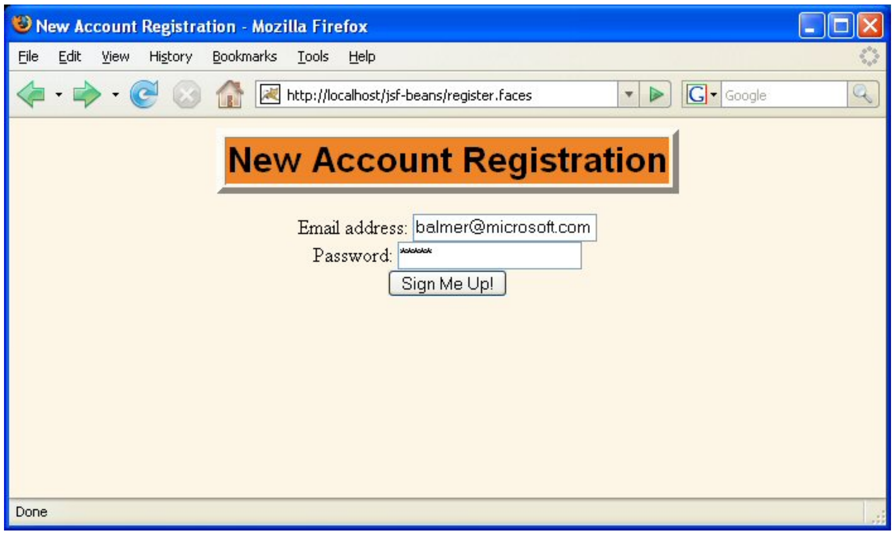
Step 4: Create Results Pages

- …/jsf-beans/WEB-INF/results/bad-password.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">Illegal Password</TH></TR>
</TABLE>
<P>
The password
"<h:outputText value="#{registrationBean.password}"/>"
is too short; it must contain at least six characters.
Here is a possible password:
<h:outputText
    value="#{registrationBean.suggestion.password}"/>.
<P>
Please <A HREF="register.faces">try again</A>.
…
</HTML>
</f:view>
```

Step 6: Example Result for Bad Password

- Input


New Account Registration - Mozilla Firefox

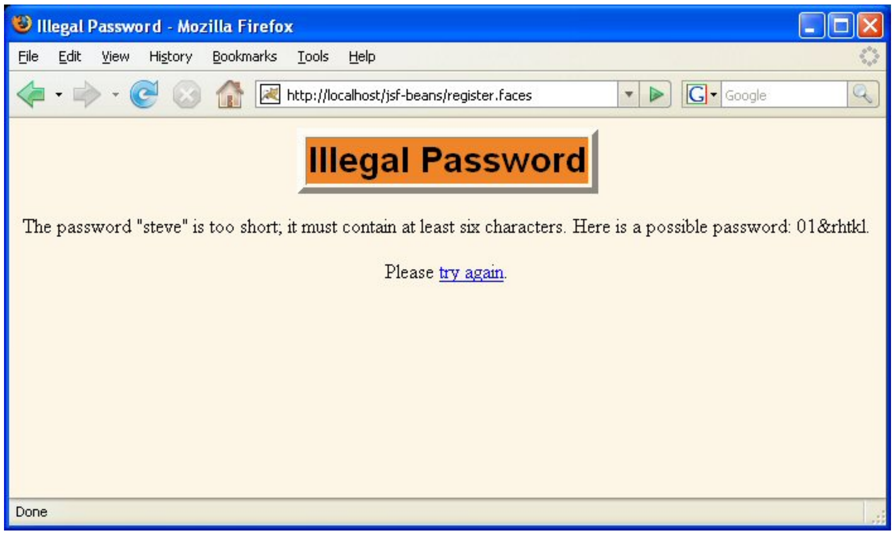
http://localhost/jsf-beans/register.faces

New Account Registration

Email address: balmer@microsoft.com
Password: *****

Step 4: Example Result for Bad Password

- Output


Illegal Password - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost/jsf-beans/register.faces
Google
Done

Illegal Password

The password "steve" is too short; it must contain at least six characters. Here is a possible password: 01&rhtkl.

Please try again.

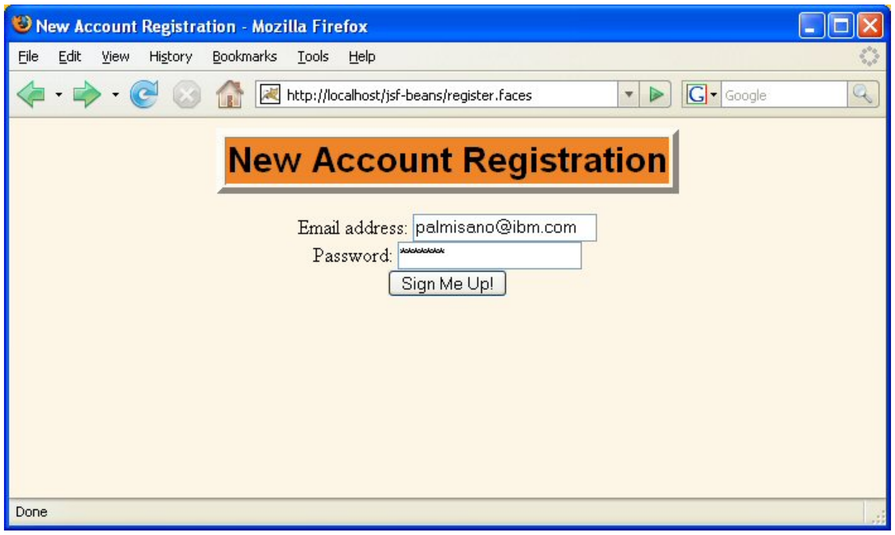
Step 4: Create Results Pages

- …/jsf-beans/WEB-INF/results/success.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">Success</TH></TR>
</TABLE>
<H2>You have registered successfully.</H2>
<UL>
  <LI>Email Address:
      <h:outputText value="#{registrationBean.email}"/>
  <LI>Password:
      <h:outputText value="#{registrationBean.password}"/>
</UL>
…
</HTML>
</f:view>
```

Step 6: Example Result for Good Input

- **Input**


New Account Registration - Mozilla Firefox

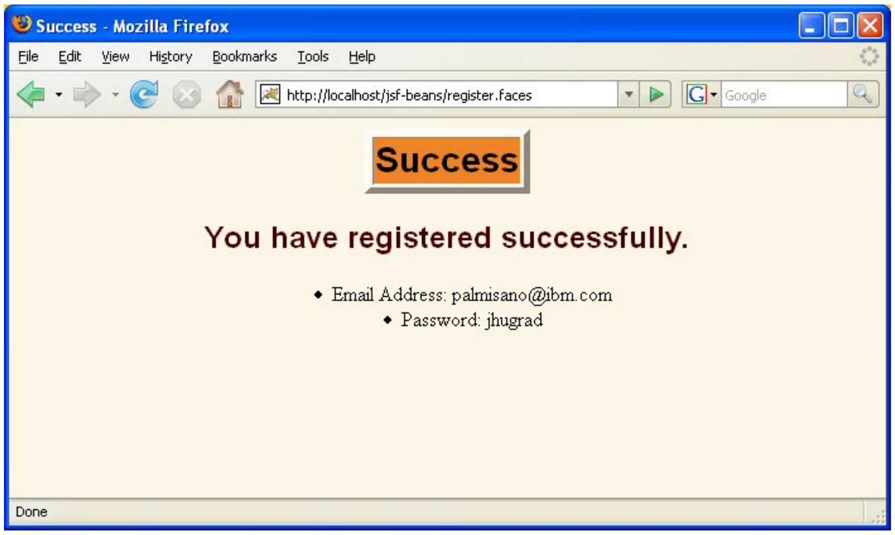
http://localhost/jsf-beans/register.faces

New Account Registration

Email address: palmisano@ibm.com
Password: *******

Step 6: Example Result for Good Input

- **Output**


Success - Mozilla Firefox

http://localhost/jsf-beans/register.faces

Success

You have registered successfully.

- Email Address: palmisano@ibm.com
- Password: jhugrad

Step 5: Prevent Direct Access to JSP Pages

- **Use filter that captures url-pattern *.jsp**
  - No changes from previous example

Alternative Approaches

- **Preview: using the JSP 2.0 EL**
  - If output pages only display bean properties (rather than manipulating a form or using form elements):
    - Why bother with f:view and associated taglib declaration?
    - Why use h:outputText and associated taglib declaration when the JSP 2.0 EL is simpler?
  - If you use the JSP 2.0 EL, you must:
    - Be in a JSP 2.0 container (e.g., Oracle10g, not Oracle9i)
    - Use the JSP 2.0 declaration for web.xml (see later section)
- **Pros of sticking with JSF**
  - You might use form elements or I18N or renderers or custom components or other JSF stuff in the future
  - h:outputText escapes < and > with &lt; and &gt;
- **Pros of using EL**
  - Shorter, simpler, more readable, already familiar

Using the JSP 2.0 Expression Language

- **Standard JSF approach**

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">Success</TH></TR>
</TABLE>
<H2>You have registered successfully.</H2>
<UL>
  <LI>Email Address:
      <h:outputText value="#{registrationBean.email}"/>
  <LI>Password:
      <h:outputText value="#{registrationBean.password}"/>
</UL>
…
</HTML>
</f:view>
```

Using the JSP 2.0 Expression Language

- **JSP 2.0 approach**
  - Omit taglib declarations and f:view tags
  - Shorten expression that outputs bean properties

```
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
  <TR><TH CLASS="TITLE">Success</TH></TR>
</TABLE>
<H2>You have registered successfully.</H2>
<UL>
  <LI>Email Address: ${registrationBean.email}
  <LI>Password: ${registrationBean.password}
</UL>
…
</HTML>
```

Looking Ahead

- **Algorithm for password length was clumsy**
  - Nice to have builtin checking for textfield lengths
  - Already supported in JSF
    - See validation section
- **Algorithm for checking legal email addresses was primitive and easily fooled**
  - Nice to have builtin checking of valid addresses
  - Supported in MyFaces via Tomahawk extensions
    - See section on MyFaces extensions
- **If both password and email were wrong, only one was reported**
  - Error pages for bad input results in too many error pages
  - Better to redisplay form and say what was wrong
    - See section on validation

Summary

- **Create a bean**
  - Properties for each request parameter
  - Action controller method
  - Placeholders to hold results objects
- **Refer to bean properties in input form**
  - <h:inputText value="#{beanName.propertyName}"/>
- **Declare bean in faces-config.xml**
  - Use managed-bean declaration
  - Bean lifecycle (assuming request scope)
    - Instantiated when form first displayed
      - Getter methods called for initial textfield values
    - Instantiated again when form submitted
      - Setter methods called for each input field
      - Action controller method called after setter method
- **Use h:outputText to output bean properties**
  - The JSP 2.0 expression language is also possible

Questions?