JSF 2: Integrated Ajax Support

JSF 2.2 Version

Originals of slides and source code for examples: http://www.coreservlets.com/JSF-Tutorial/jsf2/

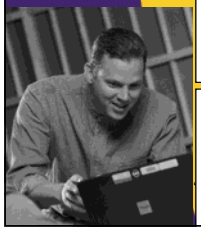
Also see the PrimeFaces tutorial – http://www.coreservlets.com/JSF-Tutorial/primefaces/

and customized JSF2 and PrimeFaces training courses – http://courses.coreservlets.com/jsf-training.html

For live training on JSF 2, PrimeFaces, or other Java EE topics, email hall@coreservlets.com

Marty is also available for consulting and development support

Taught by the author of *Core Servlets and JSP*, this tutorial, and JSF 2.2 version of *Core JSF*. Available at public venues, or customized versions can be held on-site at your organization.



- Courses developed and taught by Marty Hall
  - JSF 2, PrimeFaces, Ajax, jQuery, Spring MVC, JSP, Android, general Java, Java 8 lambdas/streams, GWT, custom topic mix
  - Courses available in any location worldwide. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Hadoop, Spring, Hibernate/JPA, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Motivation**
  - Web apps in general
  - Ajax in general
  - Ajax integrated with JSF 2
- **Using f:ajax**
  - Overview
  - render: specifying elements to update on client
  - execute: specifying elements to process on server
  - event: understanding default of valueChange
  - event: changing the event that Ajax responds to
  - onevent: specifying JavaScript side effects
  - Limitations on the use of h:outputText with Ajax

Motivation

Why Web Apps?

- **Downsides to browser-based apps**
  - GUI is poor
    - HTML is OK for static documents, but lousy for programs
  - Communication is inefficient
    - HTTP is poor protocol for the way we now use Web apps

Why Web Apps? (Continued)

- **So why does everyone want Web apps?**
  - Universal access
    - Everyone already has a browser installed
    - Any computer on the network can access content
  - Automatic "updates"
    - Content comes from server, so is never out of date

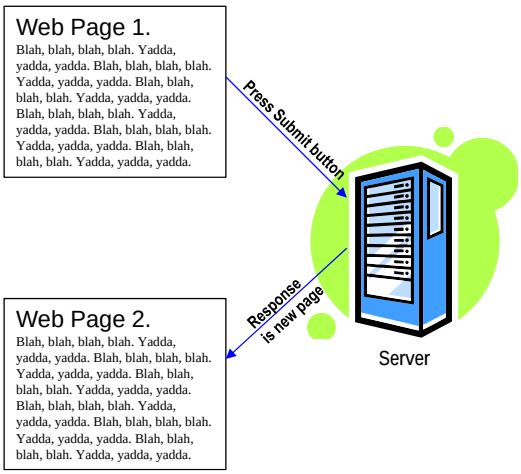
Why Ajax?

- **Solve three key problems of Web apps**
  - Coarse-grained updates
  - Synchronous: you are frozen while waiting for result
  - Extremely limited options for widgets (GUI elements)
- **Still browser based**
  - Ajax is about "what is the best you can do with what everyone *already* has in their browser?"
- **"Real" browser-based active content**
  - Failed: Java Applets
    - Not universally supported; can't interact with the HTML
  - Serious alternative: Flash/Flex
    - Not preinstalled on all PCs; not available for iPhone/iPad
  - Less proven
    - Microsoft Silverlight
    - JavaFX

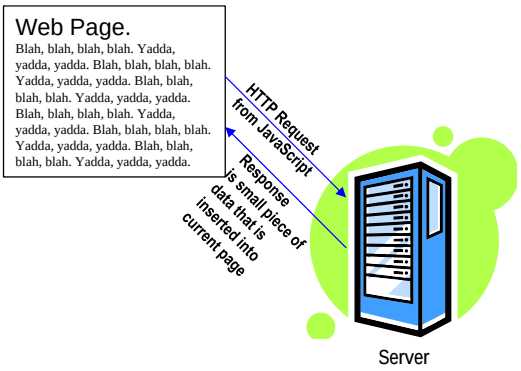
From Bill Amend and foxtrot.com. © Universal Press Syndicate.

Traditional Web Apps vs. Ajax Apps

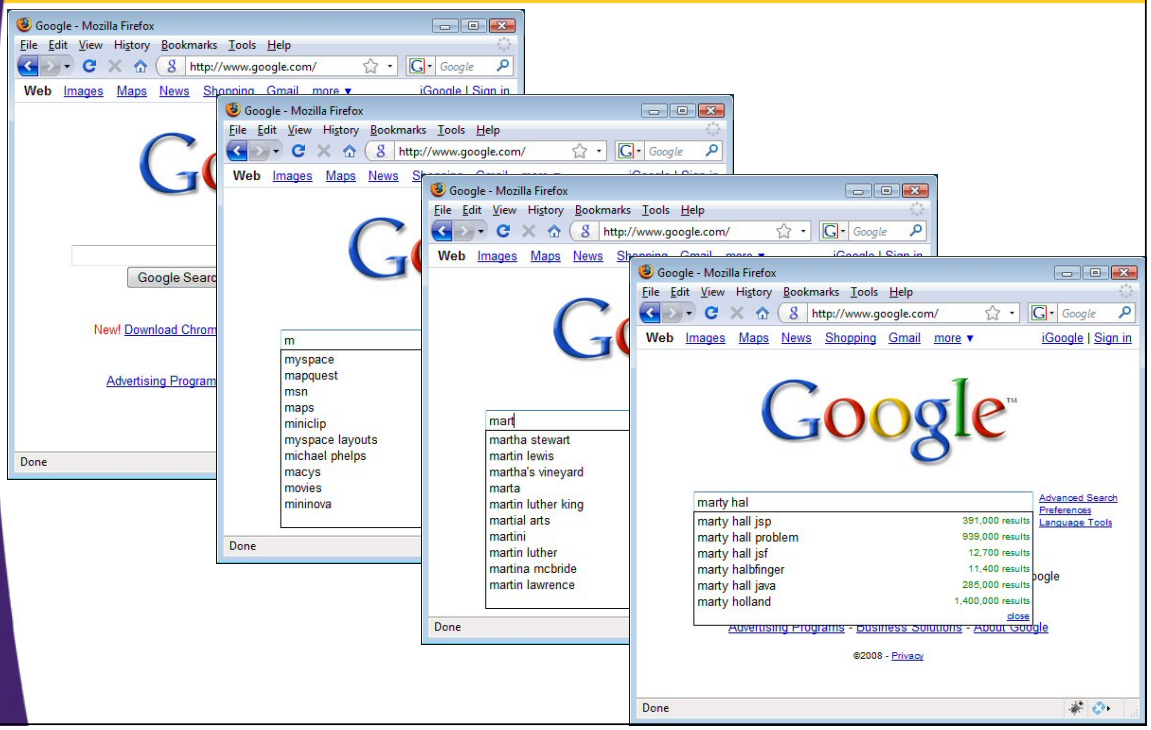
- Traditional Web Apps: Infrequent Large Updates



- Ajax Apps: Frequent Small Updates

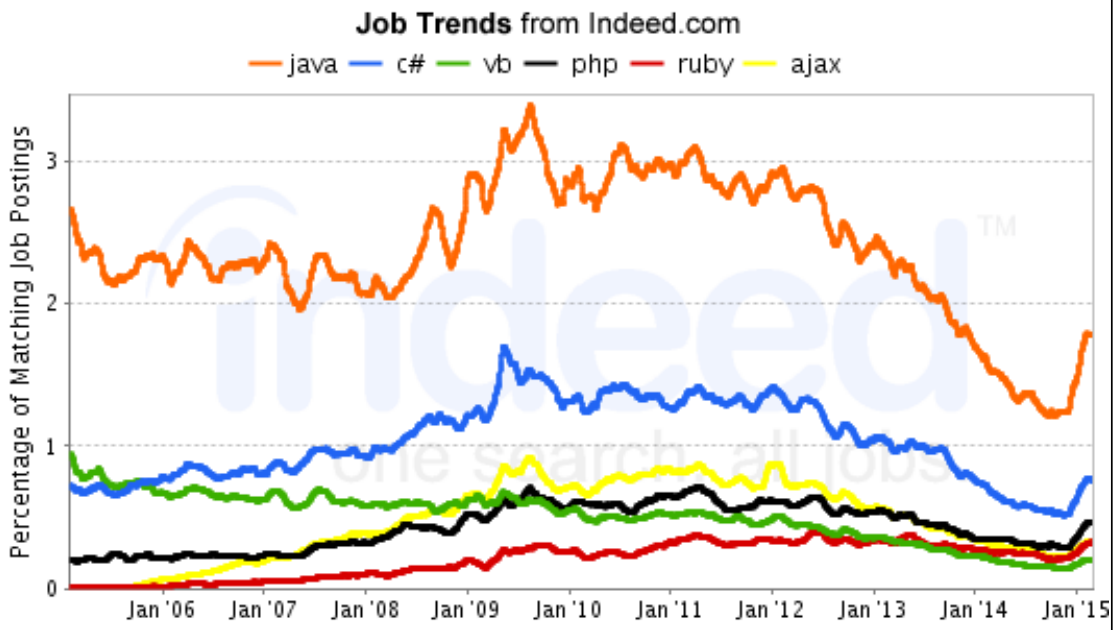


Google Home Page (formerly Google Suggest)



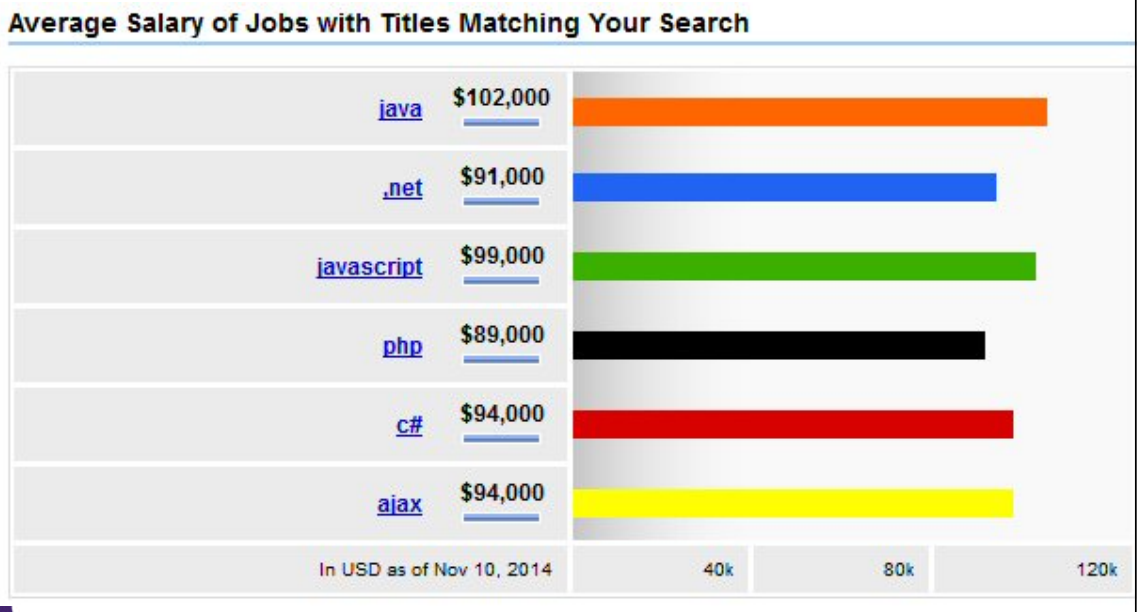
Ajax Jobs

Indeed.com compiles data from multiple jobs sites



Ajax Jobs

Data for US jobs, averaged over all states. From indeed.com as of 11/2014



Why Ajax in JSF?

- **Why a JSF-specific Ajax library?**
  - There are tons of Ajax libraries already (jQuery, DWR, GWT, etc.). Why invent a new one for JSF?
- **Advantages of a JSF-specific Ajax approach**
  - Client side
    - You can update *JSF* elements (h:outputText, h:inputText, h:selectOneMenu, etc.)
      - It would be very unwieldy if Ajax updates were entirely separate elements from the Ajax UI elements
    - You don't have to write JavaScript
  - Server side
    - Ajax calls know about JSF managed beans
      - Including reading form fields and setting bean properties
    - You don't have to write servlets and parse parameters

f:ajax – Overview

Simplest Form

- **Code**
  - <h:commandButton … action="…">
    <f:ajax render="id1"/>
    </h:commandButton>
    <h:outputText … value="#{…}" id="id1"/>
- **Interpretation**
  - When the pushbutton is pressed, go to server, run the action, compute the value of the JSF element whose id is "id1", send that value back to the client, then replace that element in the DOM with the new value.
    - If the "value" attribute computes a new result each time, then the "action" of the button can be omitted

```
<h:commandButton value="Update Time">    <!-- No action -->
  <f:ajax render="timeResult"/>
</h:commandButton>
<h:outputText value="#{dateBean.time}"  id="timeResult"/>
```

General Form

- **Code**
  - <h:commandButton … action="…">
    <f:ajax render="id1 id2" execute="id3 id4"
    event="blah" onevent="javaScriptHandler"/>
    </h:commandButton>
- **Attributes**
  - render
    - The elements to redisplay on the page. Often h:outputText
      - The target of the render must be inside the same h:form
  - execute
    - The elements to send to server to process. Generally input elements such as h:inputText or h:selectOneMenu.
  - event
    - The DOM event to respond to (e.g., keyup, blur)
  - onevent
    - A JavaScript function to run when event is fired

Structure of Facelets Pages

- **Declare the f:namespace**
  - In the <html …> start tag, you must declare the namespace for the f: tags
    - We saw the same requirement in other tutorial sections where we used f: tags (e.g., f:selectItems and f:param)
- **Use h:head**
  - When you use f:ajax, the system automatically inserts a <script> tag into the <head> section of the page. It cannot find the head section unless you use h:head
  - As discussed in early tutorial sections, it is a good standard practice to *always* use h:head and h:body.
  - If browser has JavaScript disabled, Ajax buttons will automatically become normal buttons with form submissions and then with the same page redisplayed

Structure of Facelets Pages

```
<!DOCTYPE …>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:head>
…
</h:head>
<h:body>
…
</h:body>
</html>
```


Signature for Action Controller Methods

- **Non-Ajax**

```
public String myActionControllerMethod() {
  …
  return("some outcome");
}
```

Technically, the return value of an action controller method is an arbitrary Object, the toString() of which will be used for navigation.

- **Ajax: preferred**

```
public String myActionControllerMethod() {
  …
  return(null);  // In non-Ajax apps, means to redisplay form
}
```

  – The return value is ignored in Ajax apps, but by returning null, method will work for *both* Ajax and non-Ajax forms. For non-Ajax apps, it will do a full page reload, but will still show the correct final result.

Ajax Testing

- **JavaScript is notoriously inconsistent**
  - You hope that the JSF implementation took this into account, and hid the browser differences. Nevertheless, JSF 2.0 is a specification, not an implementation, so different implementations could be better or worse at this.
- **Test on multiple browsers**
  - If you field an internal application, test on all officially sanctioned browsers on all supported operating systems.
  - If you field an external application, test on as many browsers as possible. Preferably: IE 6-9, a recent Firefox implementation, Chrome, and Safari.
    - Test regularly on latest versions of Chrome, IE, and Firefox. Test on a wider set of browsers before deploying.
      - Usage: http://www.w3schools.com/browsers/browsers_stats.asp
    - This is needed for any Ajax application, not just for JSF 2.

render: Specifying Elements to Update on Client

The render attribute of f:ajax

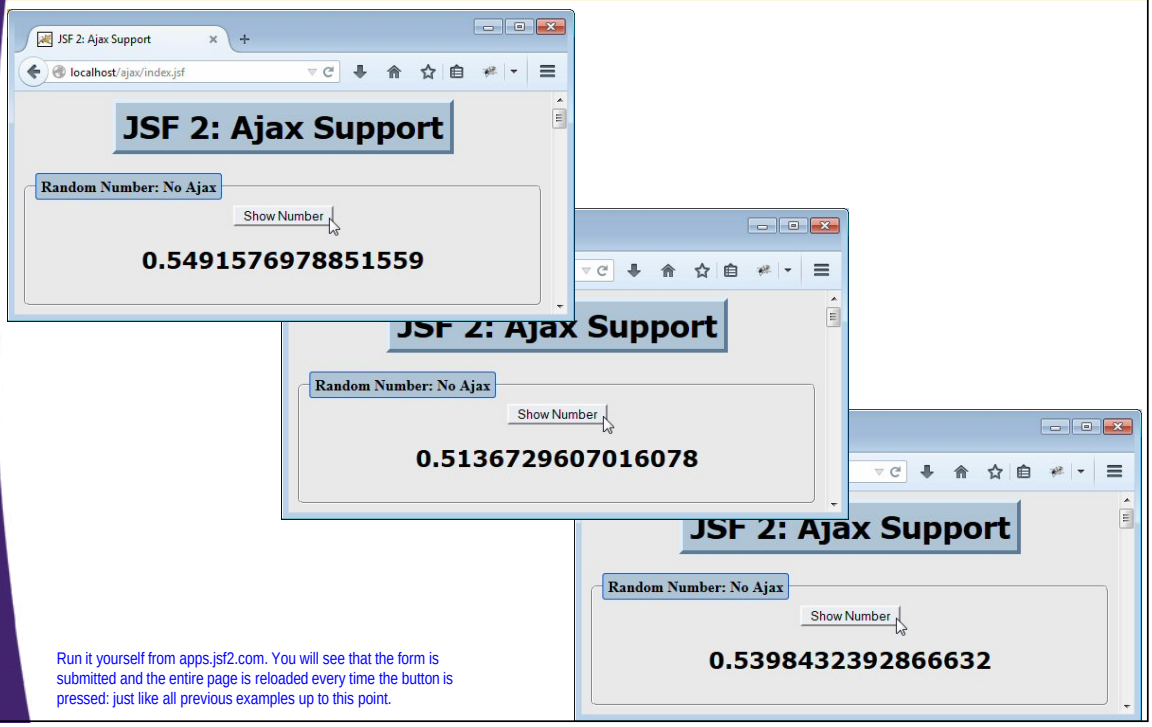
- **Code summary**
  - <f:ajax render="elementId" … />
- **Idea**
  - Id or space-separated list of ids of JSF elements whose values should be returned from server and replaced in DOM
    - These JSF elements should be in same h:form as f:ajax
- **Details**
  - There are four special values: @this, @form, @none, and @all. However, these are more often used for the execute attribute than the render attribute. See execute section.
  - Values for render (and execute and event) can be JSF expressions instead of literal strings

Facelets Code: Non-Ajax Version

```
<h:form>
<fieldset>
  <legend>Random Number: No Ajax</legend>
  <h:commandButton value="Show Number"
                   action="#{numberGenerator.randomize}"/>
  <h2><h:outputText value="#{numberGenerator.number}"/>
   </h2>
</fieldset>
</h:form>
```

When the pushbutton is pressed, submit form and on server, get bean whose name is numberGenerator (if it is session or applicatoin scoped, use existing instance if available, otherwise instantiate it). Then, run the randomize method. Then, compute the value of the getNumber method and insert it into the page where the h:outputText is. Note that h:outputText is not needed here; it is just planning ahead for the next slide. In this specific example, we could have just replaced the entire h:outputText with #{numberGenerator.number}.

Results: Non-Ajax Version



Run it yourself from apps.jsf2.com. You will see that the form is submitted and the entire page is reloaded every time the button is pressed: just like all previous examples up to this point.

Facelets Code: Ajax Version

```
<h:form>
<fieldset>
  <legend>Random Number</legend>
  <h:commandButton value="Show Number"
                   action="#{numberGenerator.randomize}">
    <f:ajax render="numField1"/>
  </h:commandButton><br/>
  <h2><h:outputText value="#{numberGenerator.number}"
                    id="numField1"/></h2>
</fieldset>
</h:form>
```

When the pushbutton is pressed, have JavaScript make an XMLHttpRequest to the server without submitting the form. On the server, get bean whose name is numberGenerator, then, run the randomize method. Then, compute the value of the getNumber method. Send that value back to the client and insert it into the DOM in the place where the h:outputText is. Note that the thing being updated (i.e., the target of render) must be inside the same h:form as the f:ajax tag that refers to it.

If JavaScript is disabled in the browser, then this form will work exactly like the one on the previous slide (i.e., with a normal form submission and then page redisplay).

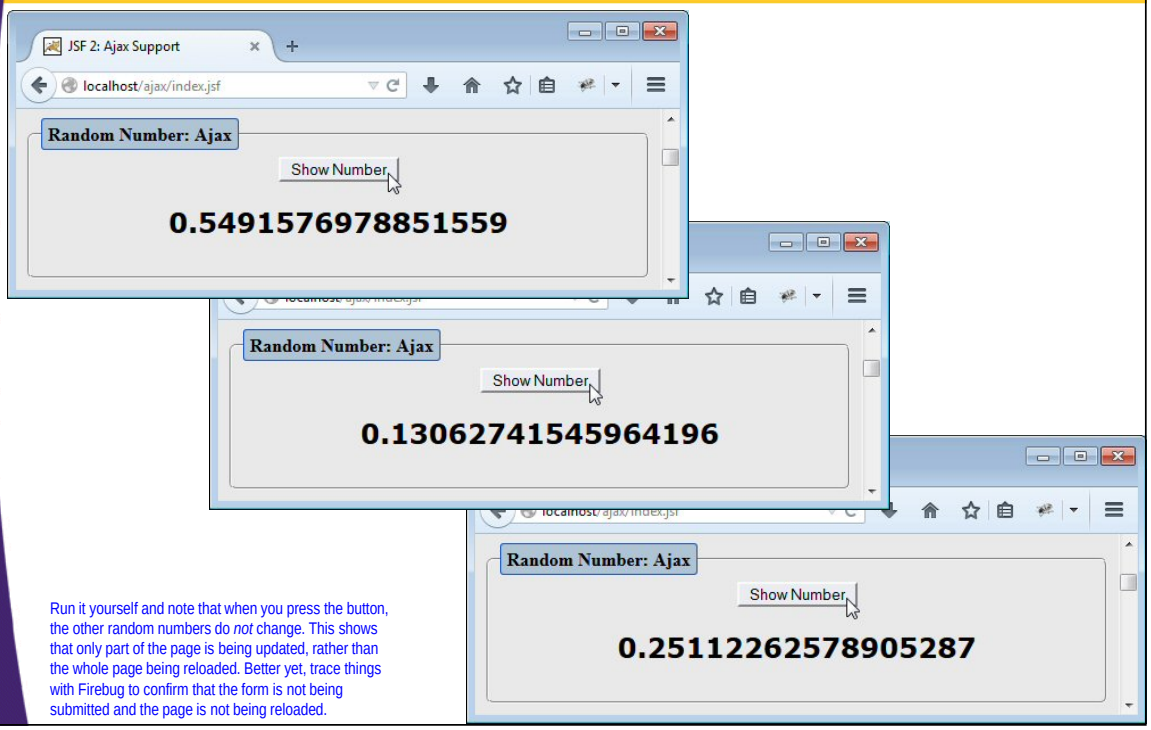
Bean Code

```
@ManagedBean
public class NumberGenerator {
  private double number = Math.random();
  private double range = 1.0;

  public double getRange() {  return(range); }

  public void setRange(double range) {
    this.range = range;
  }
  public double getNumber() { return(number); }

  public String randomize() {
    number = range * Math.random();
    return(null);
  }
}
```

getRange and setRange aren't used in this example, but are used in an upcoming one.

In Ajax apps, the return value of the action controller method is ignored. In non-Ajax apps, returning null means to redisplay the current page. By using null here, the app will still work (albeit with a full page reload) if you remove f:ajax.

Results: Ajax Version



Run it yourself and note that when you press the button, the other random numbers do *not* change. This shows that only part of the page is being updated, rather than the whole page being reloaded. Better yet, trace things with Firebug to confirm that the form is not being submitted and the page is not being reloaded.

Minor Variation: Omitting the action

The render attribute of f:ajax

- **Code summary**

```
<h:commandButton value="Button Label">        <!-- No action -->
   <f:ajax render="someId" … />
</h:commandButton>
<h:outputText id="someId"
              value="#{someBean.propertyThatComputesValue}"/>
```

- **Idea**
  - In last example, the action controller method computed the random number and stored it in an instance variable. Then, the getter method retrieved the instance variable. We could have just had the getter method directly return a random number, and skipped the action controller method.
    - This will still work in non-Ajax apps, resulting in a full page reload
- **Warning**
  - If the getter is called more than once in the page, this approach will be much less efficient. In most real-life apps, the action controller method calls the business logic and stores the result in an instance variable. The getter than retrieves that value.

Facelets Code

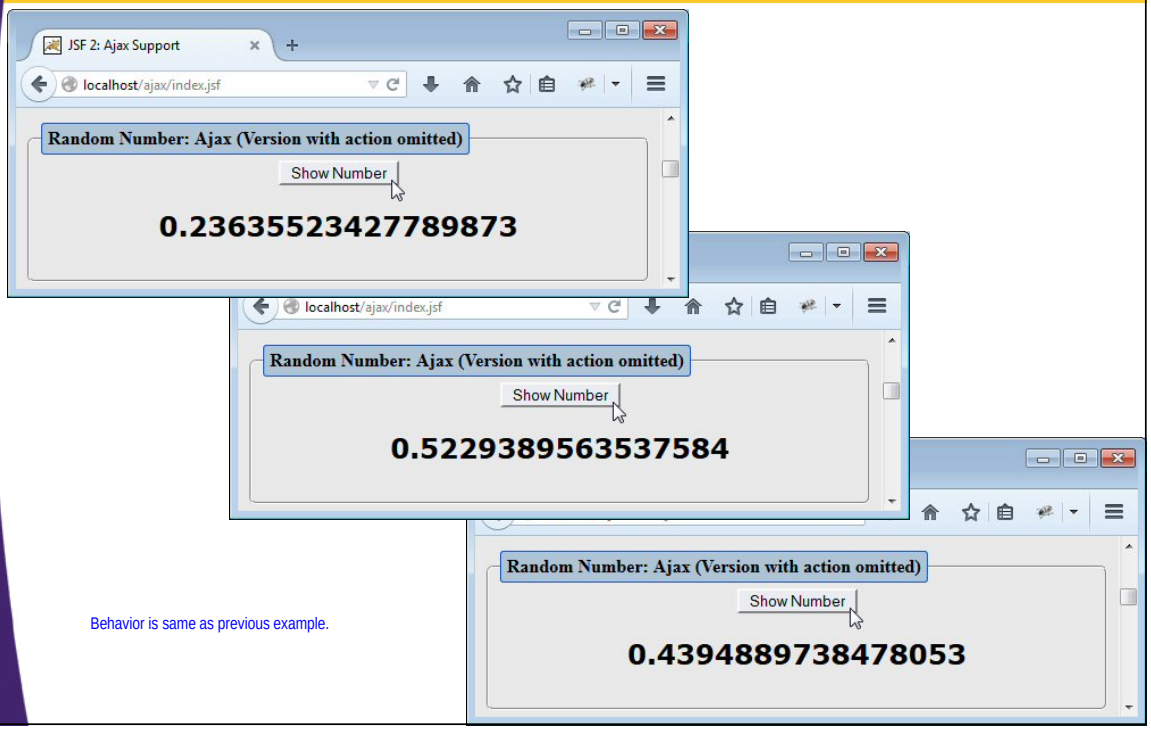
```
<fieldset>
  <legend>Random Number: Ajax
         (Version with action omitted)</legend>
  <h:commandButton value="Show Number"> <!-- No action -->
    <f:ajax render="numField2"/>
  </h:commandButton>
  <h2><h:outputText value="#{numberGenerator.number2}"
                    id="numField2"/></h2>
</fieldset>
</h:form>
```

Since getNumber2 directly computes and returns a new random number, there is no need to run an action controller method first. Omitting action like this is slightly easier in simple examples, but is less used in real-life examples.

Bean Code

```
@ManagedBean
public class NumberGenerator {
  ...

  public double getNumber2() {
    return(Math.random() * range);
  }
}
```

Results: Ajax Version with "action" Omitted



Behavior is same as previous example.

execute: Specifying Elements to Process on Server

The execute attribute of f:ajax

- **Code summary**
  - <f:ajax render="…" execute="…" … />
- **Idea**
  - An id or space-separated list of ids of JSF elements that should be sent to the server for execution.
    - h:inputText processed normally (setters, validation, etc.)
- **Details**
  - There are 4 special values: @this, @form, @none, @all
    - @this. The element enclosing f:ajax. Default.
    - @form. The h:form enclosing f:ajax. Very convenient if you have multiple fields to send. Shown in later example.
    - @none. Nothing sent. Useful if the element you render changes values each time you evaluate it.
    - @all. All JSF UI elements on page.

Facelets Code

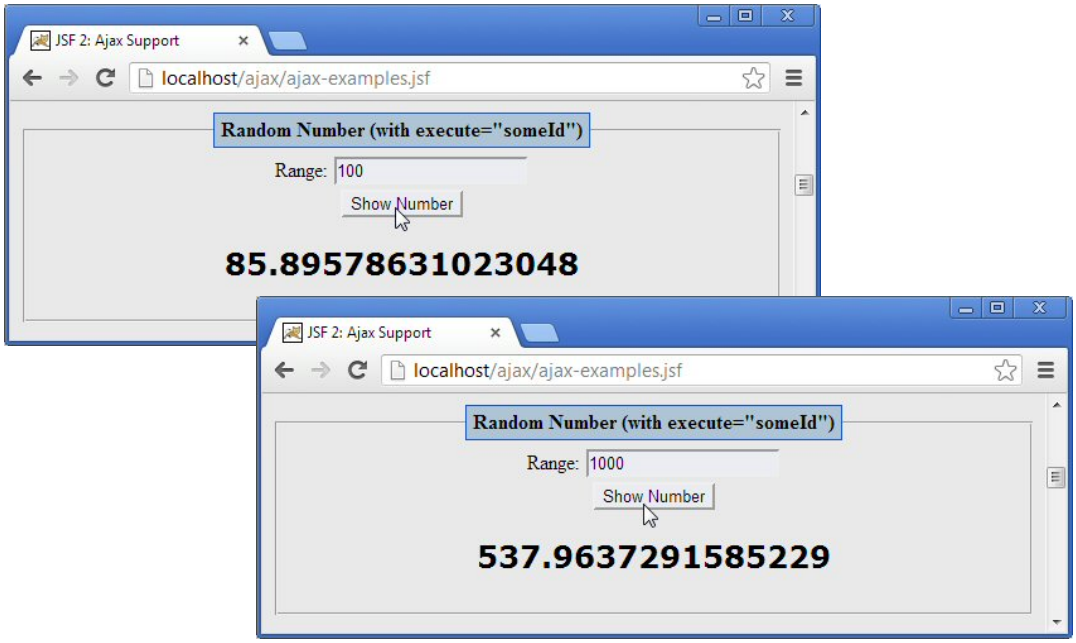
```
<h:form>
<fieldset>
  <legend>Random Number (with execute="someId")</legend>
  Range:
  <h:inputText value="#{numberGenerator.range}"
               id="rangeField"/><br/>
  <h:commandButton value="Show Number"
                   action="#{numberGenerator.randomize}">
    <f:ajax execute="rangeField"
            render="numField3"/>
  </h:commandButton><br/>
  <h2><h:outputText value="#{numberGenerator.number}"
                    id="numField3"/></h2>
</fieldset>
</h:form>
```

Bean Code

```
@ManagedBean
public class NumberGenerator {
  private double number = Math.random();
  private double range = 1.0;

  public double getRange() {
    return(range);
  }
  public void setRange(double range) {
    this.range = range;
  }
  public double getNumber() {
    return(number);
  }
  public String randomize() {
    number = range * Math.random();
    return(null);
  }
}
```

This is the same bean code as in previous examples. But shown again to emphasize that setRange (corresponding to the h:inputText element specified with execute) will be called before randomize (the action of the h:commandButton that surrounds f:ajax).

Results



Using execute="@form"

- **Code summary**
  - <h:form />

    …

    <f:ajax render="elementId" execute="@form" />

    …

    </h:form>
- **Idea**
  - Send all elements of current form to server to process.
    - Again, processes form elements in normal JSF manner (performs validation, calls setter methods, etc.)
- **Details**
  - Convenient if you have several input fields and don't want to list each one in "render". Also, input fields don't need explicit ids when you use @form.

Facelets Code

```
<h:form>
<fieldset>
  <legend>Bank Customer Lookup (with execute="@form")</legend>
  Customer ID:
  <h:inputText value="#{bankingBeanAjax.customerId}"/><br/>
  Password:
  <h:inputSecret value="#{bankingBeanAjax.password}"/><br/>
  <h:commandButton value="Show Current Balance"
                   action="#{bankingBeanAjax.showBalance}">
    <f:ajax execute="@form"
            render="ajaxMessage1"/>
  </h:commandButton>
  <br/>
  <h2><h:outputText value="#{bankingBeanAjax.message}"
                    id="ajaxMessage1"/></h2>
</fieldset>
</h:form>
```

I didn't need to give explicit ids to the input fields. JSF generates ids automatically when no id specified.
Since @form was given, JSF finds all elements in current form and sends them to server for processing.

Bean Code

```
@ManagedBean
public class BankingBeanAjax extends BankingBeanBase {
  private String message = "";

  public String getMessage() {
    return(message);
  }

  public void setMessage(String message) {
    this.message = message;
  }
```


Bean Code (Continued)

```
  public String showBalance() {
    if (!password.equals("secret")) {
      message = "Incorrect password";
    } else {
      CustomerLookupService service =
        new CustomerSimpleMap();
      customer = service.findCustomer(customerId);
      if (customer == null) {
        message = "Unknown customer";
      } else {
        message =
          String.format("Balance for %s %s is $%,.2f",
                        customer.getFirstName(),
                        customer.getLastName(),
                        customer.getBalance());
      }
    }
    return(null);
  }
}
```

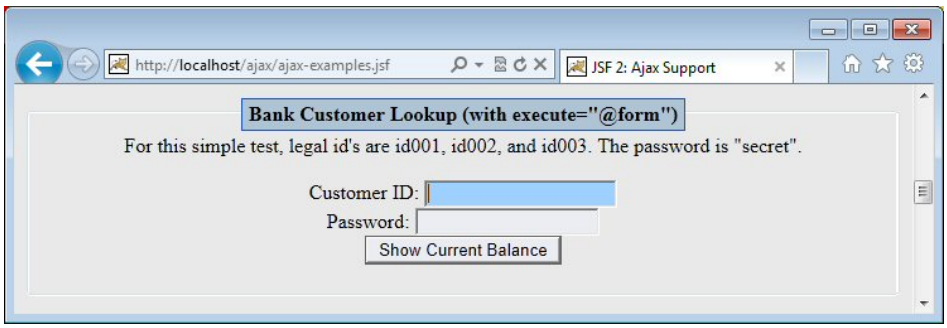
The message starts off as an empty String. Once the button is pressed, showBalance changes the message to either an error message or a string showing the balance. Then JSF replaces the h:outputText value in the DOM with the new message. This is same supporting class as shown in previous section on annotations. CustomerSimpleMap is just a lookup table of a few customers, and is shown in the previous section (plus is in the downloadable source code).

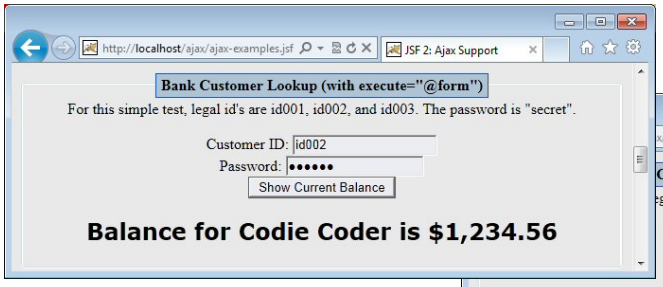
BankingBeanBase.java

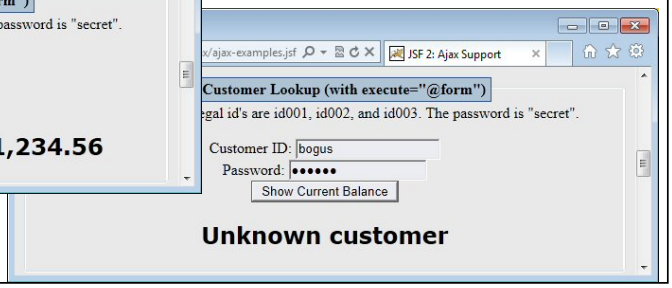
```
…
public abstract class BankingBeanBase {
  protected String customerId, password;
  protected Customer customer;

  public String getCustomerId() { return(customerId); }

  public void setCustomerId(String customerId) {
    this.customerId = customerId;
  }

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

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

These will be automatically called by JSF because the corresponding h:input elements were processed due to the execute="@form" attribute.

Results







event attribute: Understanding the default of valueChange

The event attribute of f:ajax

- **Code summary**
  - <f:ajax render="…" event="…" … />
- **Idea**
  - The name of the DOM event to respond to. You don't include "on", so it is mouseover, keyup, blur, etc.
- **Details**
  - Defaults
    - If unspecified, default event used. See next slide.
  - High-level events
    - JSF adds 2 extras: action & valueChange. See next slide.
  - Wrapping f:ajax around elements
    - <f:ajax render="…">a bunch of components</f:ajax>
      - Adds Ajax behavior on default event for each wrapped component

Default Events

- **action**
  - h:commandButton, h:commandLink
    - Note that “action” is part of JSF, and not a native JavaScript/DOM event name. It means button has been invoked in any way (clicking on it, ENTER if it has focus, keyboard shortcut, etc.)
- **valueChange**
  - h:inputText, h:inputSecret, h:inputTextarea, all radio button, checkbox, & menu items (h:selectOneMenu, etc.)
    - Again, this event is added by JSF and is not a native DOM event name. Different browsers handle “change” differently, so this unifies the behavior.
    - Also note that it is “valueChange”, not “valuechange”. The native DOM events are all lower case (mouseover, keyup, etc.)

Example: Chained Combo Boxes (Select Menus)

- **Desired behavior**
  - Use h:selectOneMenu to make a list of US states
  - When the user selects a state, a list of corresponding cities is shown
    - Again, using h:selectOneMenu
  - When city selected, population of that city is displayed
  - If a new state is selected, the list of cities changes and the population is cleared

Example: Approach

- **Ajax usage**
  - State list
    - <f:ajax render="cityList population"/>
      - When state selection changes, update the list of cities and clear displayed population if any
  - City list
    - <f:ajax render="population"/>
      - When city selection changes, show population.
      - In this example, I use a Map for the city list, where the values are the city names and the corresponding values are the populations.
  - Bean
    - Make managed bean session scoped so that it remembers which cities are currently shown

Example: Approach

- **Dummy values at top of list**
  - State list
    - <f:selectItem itemLabel="--Choose State--"/>
      - Since valueChange is default event, you need dummy value at the top so that any real state selection is a change.
      - Alternative approach is to pick a state, city, and population to show at beginning. But assuming you want city list and population empty to begin with, dummy value at top is needed.
      - This is not needed in a normal, non-Ajax Web app
  - City list
    - <f:selectItem itemLabel="--Choose City--"/>
      - Again, you must ensure that any real city selection is a change

Facelets Code

```
<h:form>...
  State:
  <h:selectOneMenu value="#{locationBean.state}">
    <f:selectItem itemLabel="--Choose State--"/>
    <f:selectItems value="#{locationBean.states}"/>
    <f:ajax render="cityList population"/>
  </h:selectOneMenu>
  <br/>City:
  <h:selectOneMenu value="#{locationBean.city}"
                   disabled="#{locationBean.cityListDisabled}"
                   id="cityList">
    <f:selectItem itemLabel="--Choose City--"/>
    <f:selectItems value="#{locationBean.cities}"/>
    <f:ajax render="population"/>
  </h:selectOneMenu>
  <br/><b>Population:
  <h:outputText value="#{locationBean.city}"
                id="population"/></b>
...</h:form>
```


Bean Code

```
@ManagedBean
@SessionScoped
public class LocationBean implements Serializable {
  private String state, cityPopulation;
  private boolean isCityListDisabled = true;

  public String getState() {
    return(state);
  }

 public void setState(String state) {
    this.state = state;
    this.cityPopulation="";        // Clear population display
    isCityListDisabled = false;  // Enable list of cities
  }
```

Make city list disabled (grayed out) initially. Enable it when the state is selected.

Bean Code (Continued)

```
public String getCityPopulation() {
  return(cityPopulation);
}

public void setCity(String cityPopulation) {
  this.cityPopulation = cityPopulation;
}

public boolean isCityListDisabled() {
  return(isCityListDisabled);
}
```

Although the end user sees a drop down menu of city names, the String passed to setCityPopulation and returned from getCityPopulation really represents the population. That is because, for drop down menues, list boxes, and radio buttons, JSF lets you have one value displayed to the end user and a different value passed to the setter method. See the discussion of use of Maps in section on menus in the Managed Beans II lecture.

Bean Code (Continued)

```
  public List<String> getStates() {
    return(StateInfo.STATE_NAMES);
  }

  public Map<String,String> getCities() {
    return(StateInfo.STATE_MAP.get(state));
  }
}
```

The "list" of cities is a Map where the display values are the city names and the return values are the corresponding populations. The STATE_MAP is another Map, where each key is a state name and each corresponding value is the city name/population Map.

Supporting Class (StateInfo)

```
public class StateInfo {
  // Maps state name to Map that has city names & associated populations
  public static final Map<String, Map<String,String>> STATE_MAP =
    new HashMap<>();
  public static final List<String> STATE_NAMES = new ArrayList<>();
  // Populations from http://www.citypopulation.de/USA.html
  public static final State[] STATES =
    { new State("Maryland",
                new City("Baltimore", "622,104"),
                new City("Columbia", "105,000"),
                new City("Frederick", "66,893"),
                new City("Gaithersburg", "65,690")),
      new State("Virginia",
                new City("Virginia Beach", "448,479"),
                new City("Norfolk", "246,139"),
                new City("Chesapeake", "230,571"),
                new City("Arlington", "224,906")),
      // More states
   };
}
```

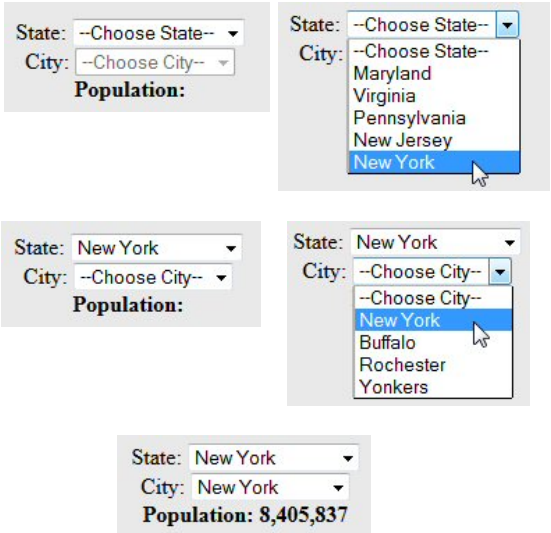

Supporting Class (State)

```
public class State {
  private final String name;
  private final Map<String,String> cityMap = new LinkedHashMap<>();

  public State(String name, City...cities) {
    this.name = name;
    for(City c: cities) {
      cityMap.put(c.getName(), c.getPopulation());
    }
    StateInfo.STATE_NAMES.add(name);
    StateInfo.STATE_MAP.put(name, cityMap);
  }

  public String getName() {
    return(name);
  }
  public Map<String, String> getCityMap() {
    return(cityMap);
  }
}
```

Supporting Class (City)

```
public class City {
  private final String name, population;

  public City(String name, String population) {
    this.name = name;
    this.population = population;
  }

  public String getName() {
    return (name);
  }

  public String getPopulation() {
    return (population);
  }
}
```


Results



event attribute: Changing the Event that Ajax Responds to

Example: On-the-Fly Temperature Converter

- **Idea**
  - The user types a temperature in Fahrenheit into textfield
  - As the value is being entered, the corresponding values in Celsius and Kelvin are displayed
- **Approach**
  - Temperature field
    - <f:ajax event="keyup"
      render="cField kField"/>
      - keyup is not the default event, so needs to be specified explicitly in "event"
      - We want to update two output fields, so list both for "render"
  - Bean
    - Temp (in F) passed each time, so use request scope

Facelets Code

```
<h:form>
<fieldset>
  <legend>On-the-Fly Temperature Converter …</legend>
  Temperature in Fahrenheit:
  <h:inputText value="#{temperatureConverter.fTemp}">
    <f:ajax event="keyup"
            render="cField kField"/>
  </h:inputText><br/>
  <h2>
    Temperature in Celsius:
    <h:outputText value="#{temperatureConverter.cTemp}"
                  id="cField"/><br/>
    Temperature in Kelvin:
    <h:outputText value="#{temperatureConverter.kTemp}"
                  id="kField"/><br/>
   </h2>
</fieldset></h:form>
```

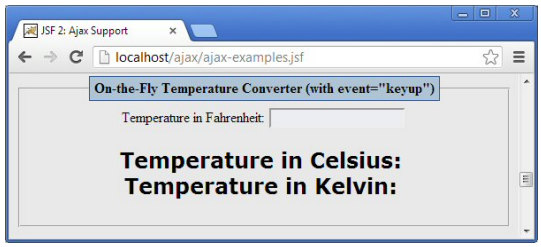

Bean Code

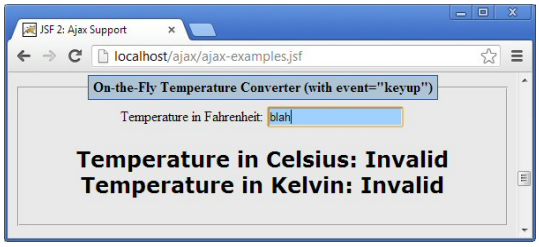
```
@ManagedBean
public class TemperatureConverter  {
  private String cTemp, kTemp;

  public String getcTemp() {
    return(cTemp);
  }

  public String getkTemp() {
    return(kTemp);
  }

  public String getfTemp() {
    return("");
  }
```

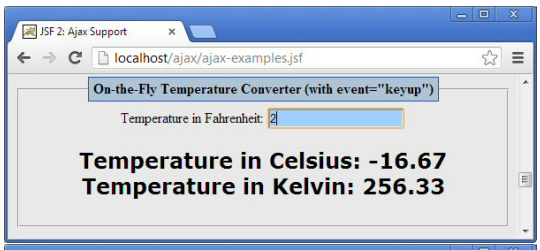
Bean Code (Continued)

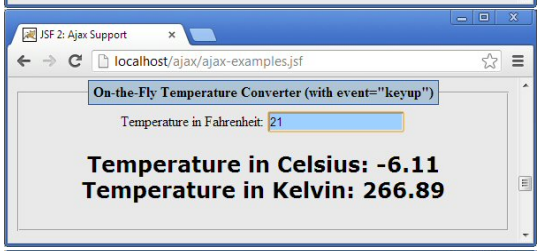
```
public void setfTemp(String fTemp) {
   double f = -500;
   try {
     f = Double.parseDouble(fTemp);
   } catch(NumberFormatException nfe) {
     cTemp = "Invalid";
     kTemp = "Invalid";
   }
   if (f >= -459.4) {
     double c = (f - 32)*(5.0/9.0);
     double k = c + 273;
     cTemp = String.format("%.2f", c);
     kTemp = String.format("%.2f", k);
   }
 }
}
```

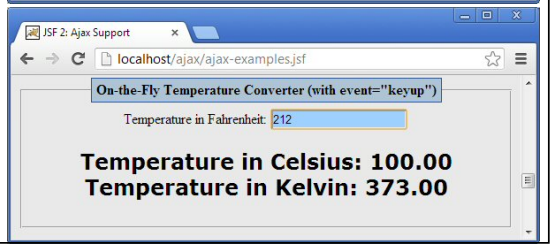

Results











onevent:
Specifying JavaScript Side Effects to Run Before/After Ajax Request

The onevent attribute of f:ajax

- **Code summary**
  - <f:ajax render="…" onevent="functionName" … />
- **Idea**
  - The name of a JavaScript function to call at various stages of the Ajax submission and response. Function should take one argument (e.g., function blah(data) {…})
- **Details on argument to JavaScript function**
  - The status property (e.g., data.status in example above)
    - Either "begin", "complete", or "success" (in that order)
  - The source property (e.g., data.source)
    - The DOM event that triggered the Ajax request
  - responseCode, responseText, responseXML
    - The values in XHR object. Omitted for “begin” event.

Example: Showing "Getting Data…" Message While Waiting

- **Idea**
  - You have slow server operation
  - Display animated GIF (& message) when request sent
  - Hide GIF/message when response completes
- **Approach**
  - Get animated GIF
    - http://ajaxload.info/ lets you build your own
  - Display image plus message in region with display: none
    - So it is hidden initially
  - When request begins, change to display: inline
    - Use onevent handler and "begin" status
  - When request finishes, make display: none
    - Use onevent handler and "success" status

Facelets Code

```
…
<h:head><title>JSF 2.0: Ajax Support</title>
<link href="./css/styles.css"
      rel="stylesheet" type="text/css"/>
<script src="./scripts/utils.js"
        type="text/javascript"></script>
</h:head>
…
```

This file defines the onevent handler that is referred to on next page.

In this simple example, we load style sheets and JavaScript files the normal XHTML way. However, if you have pages in several folders that use the same resources (especially with ui:composition), this standard approach is inconvenient since the relative URL to the resources could be different for each of the pages. So, JSF 2.0 adds h:outputScript, h:outputStyleSheet, and an option for h:graphicImage. These let you load resources (scripts, style sheets, and images) with the same syntax, regardless of where the loading page is situated in your app. These approaches will be covered in the later section on page templating.

Facelets Code (Continued)

```
<h:form>
<fieldset>
  <legend>Bank Customer Lookup (with onevent)</legend>
  Customer ID:
  <h:inputText value="#{bankingBeanSlow.customerId}"/><br/>
  Password:
  <h:inputSecret value="#{bankingBeanSlow.password}"/><br/>
  <h:commandButton value="Show Current Balance"
                   action="#{bankingBeanSlow.showBalance}">
    <f:ajax execute="@form"
            render="ajaxMessage2"
            onevent="showWorkingIndicator"/>
  </h:commandButton>
```

Calls the showWorkingIndicator JavaScript function (loaded on previous slide) with a data object at various stages in the Ajax process. The data object has a status property that lets the programmer decide whether it is before or after the Ajax request.

Facelets Code (Continued)

```
  <h2 id="workingIndicator" style="display: none">
      <img src="./images/ajax-loader.gif"/>
      Loading data from server...</h2>
  <h2><h:outputText value="#{bankingBeanSlow.message}"
                    id="ajaxMessage2"/></h2></fieldset>
</h:form>
```

This heading is hidden initially. It is made visible in the first call to the showWorkingIndicator onevent handler (for the “begin” status) and hidden again in a later call to the onevent handler (for the “complete” status).

JavaScript Code (Part 1)

```
function showWorkingIndicator(data) {
  showIndicatorRegion(data, "workingIndicator",
                            "ajaxBank:ajaxMessage2");
}
```

This is the function referred to in onevent. It is not reusable since it refers to specific region ids, but the other functions are generic and reusable.

This is the reusable function called above.

```
function showIndicatorRegion(data, spinnerRegionId,
                             messageRegionId) {
  if (data.status == "begin") {
    clearExistingText(messageRegionId);
    showElement(spinnerRegionId);
  } else if (data.status == "complete") {
    hideElement(spinnerRegionId);
  }
}
```


JavaScript Code (Part 2)

```
function clearExistingText(id) {
   document.getElementById(id).innerHTML = "";
}
```

If you resubmit a form that already has a previous answer, you do not want the previous results shown below the "Loading data from server..." message. This clears it. Note that the raw HTML ID is the if of the JSF form, then a colon, then the id of the JSF input element.

```
function showElement(id) {
  document.getElementById(id).style.display
    = "block";
}

function hideElement(id) {
  document.getElementById(id).style.display
    = "none";
}
```

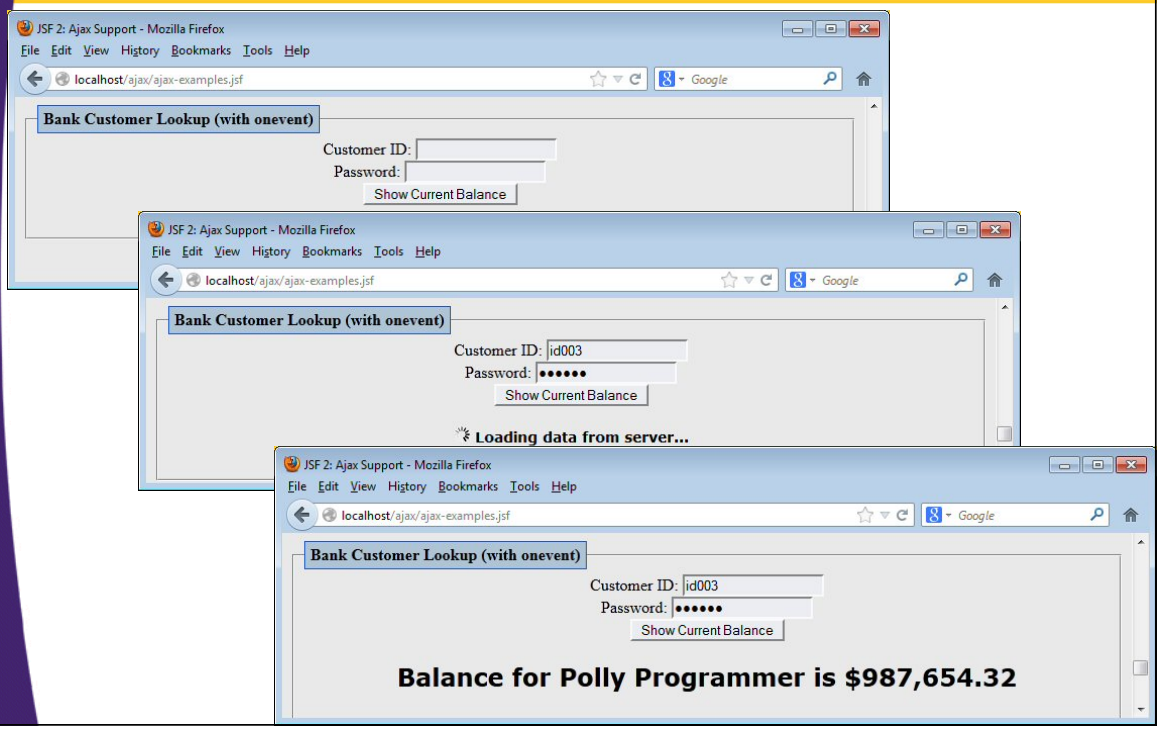
These make the hidden region visible and invisible. In real life I would probably use jQuery and $("#workingIndicator").hide() and ...show(). But I wanted to keep this as simple as possible for folks that don't know jQuery.Similarly, experienced JavaScript programmers would use namespaces, not functions at the top level. This is omitted here to keep things simpler for JavaScript newbies.

Bean Code

```
@ManagedBean
public class BankingBeanSlow extends BankingBeanAjax {
  public String showBalance() {
    try {
      Thread.sleep(5000);
    } catch(InterruptedException ie) {}
    return(super.showBalance());
  }
}
```

Works the same as the previous banking example, except for the extra five-second delay.

Results



Wrap-Up

Summary

- **Simple example**

```
<h:form>
  <h:commandButton value="Show Number"
                   action="#{numberGenerator.randomize}">
    <f:ajax render="numField1"/>
  </h:commandButton><br/>
  <h2><h:outputText value="#{numberGenerator.number}"
                    id="numField1"/></h2>
</h:form>
```

- **General format**
  - <f:ajax render="ids to update"
    execute="ids to send to server" *(or use @form)*
    event="event to respond to"
    onevent="javaScriptFunctionName"/>

Questions?

More info:
http://www.coreservlets.com/JSF-Tutorial/jsf2/ – JSF 2.2 tutorial
http://www.coreservlets.com/JSF-Tutorial/primefaces/ – PrimeFaces tutorial
http://courses.coreservlets.com/jsf-training.html – Customized JSF and PrimeFaces training courses
http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training