Web Applications 2017

Week 4, Slides 1

Content

- ES6 Overview
- Components
- Server vs client side components
- Web components
- Shadow DOM
- Custom elements
- Template tag
- Slots
- Babel, Polyfills and Webpack
- React components
- Java Server Faces

eqweq

ES6 (aka EcmaScript 2015)

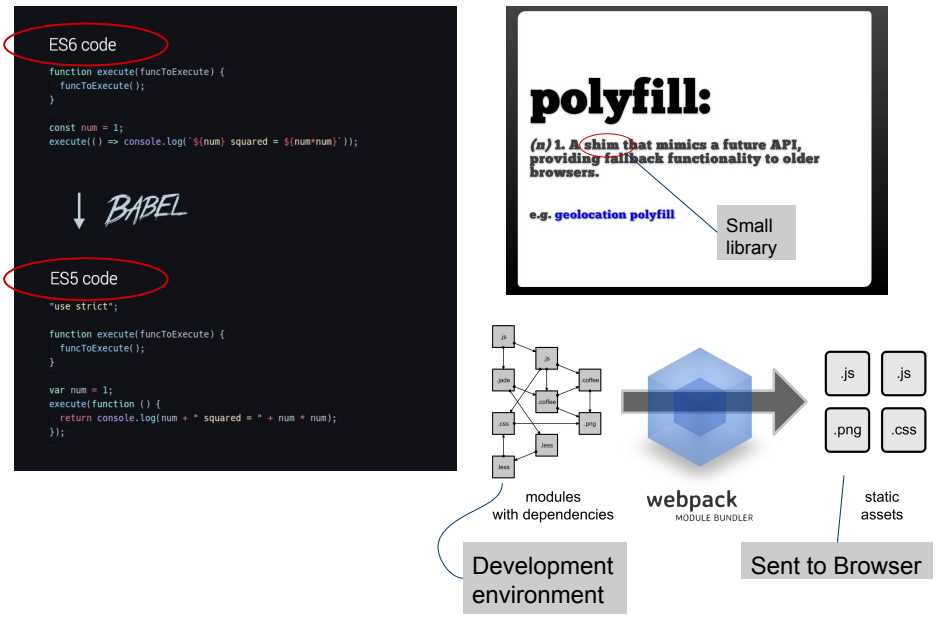
Of most interest right now …

- Modules
- Classes
- Template literals
- Arrow functions
- Block scope

Readings

- ES6 Overview features

Babel, Polyfill and Webpack



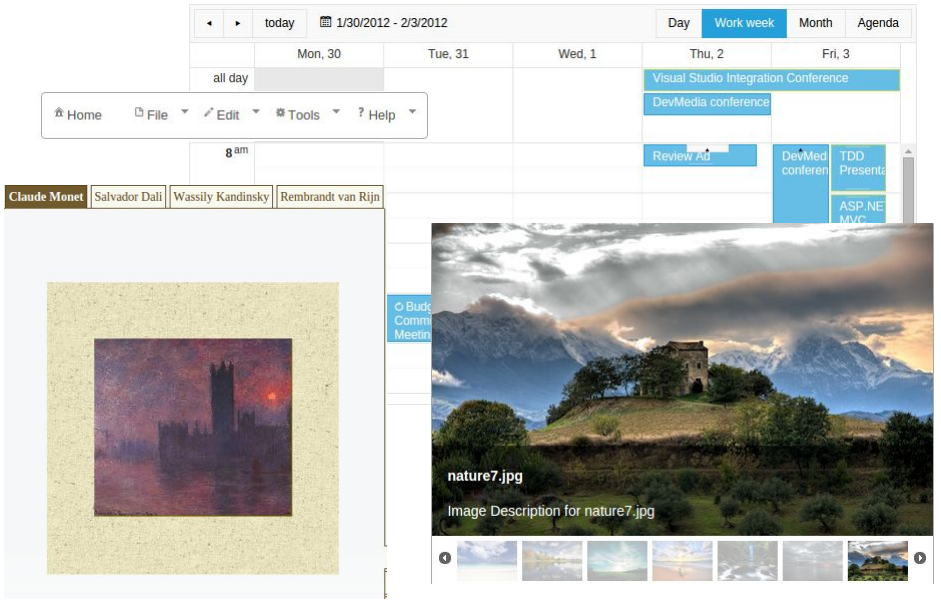
Tools needed.

- We need babel and webpack (because React uses them, upcoming).

Readings

- Babel
- Polyfill
- Webpack

Components: Basic Ideas



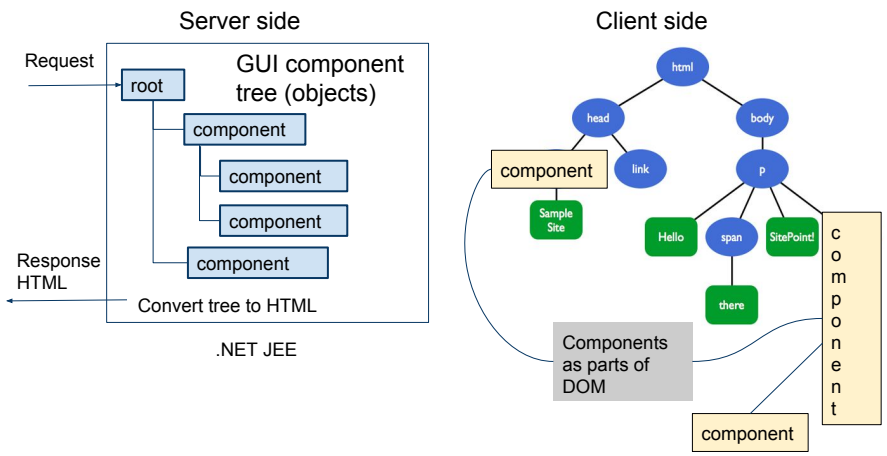
Basic idea is to do desktop programming on the web (like Swing, JavaFX)

- Skip all (own) CSS, JS, components hiding it all

Demo:

- Mock OSX (JEE, PrimeFaces)
- Todo list (React, client side components)

Server vs Client side Components



Server side: Let GUI live as a component tree (tree of objects) on server side

- One for each unique client, so tree must have unique id
- Translate to HTML (CSS, JS) and send to client for rendering

Client side

- Components attached to DOM
- Components (normally) created client side

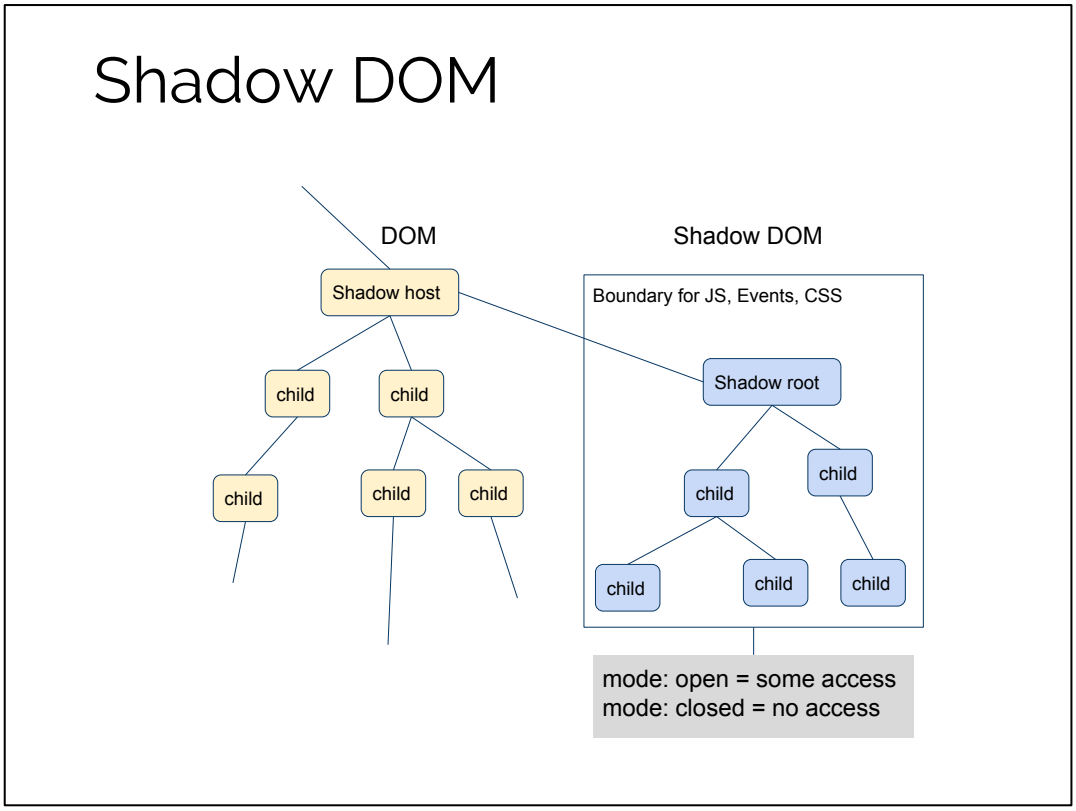
Web Components

Specification from W3c (client side)

- Shadow DOM
- Custom Elements
- HTML Templates
- HTML Imports (cancelled?)

Readings

- Current status W3C
- Web components MDB



Readings

- [Shadow DOM](#) Google

Create a Shadow DOM

```
<!-- Not all elements can act as hosts -->
    <p id="hostElement"></p>
    <script>
        var host = document.querySelector('#hostElement');
        // create shadow DOM on the <p> element above
        var shadow = host.attachShadow({
            mode: 'open'
        });
        console.log(host);
        var shadowRoot = host.shadowRoot;
        console.log(shadowRoot);
        console.log(shadowRoot.host);

        shadow.innerHTML = '<p>Here is some new text</p>';

        var found = document.querySelector('#hostElement p');
        console.log(found); // NOT Found!
        var found = document.querySelector('p');
        console.log(found); // NOT Found!
        // Just affect p in shadow (not outer p's)
        shadow.innerHTML += '<style>p { color: red; }</style>';
    </script>
```

Create shadow DOM

Shadow DOM encapsulated

Custom Element

A name

A valid custom element name

A local name

A local name

A constructor

A custom element constructor

A prototype

A JavaScript object

A list of observed attributes

A sequence<DOMString>

A collection of lifecycle callbacks

A map, whose four keys are the strings "connectedCallback", "disconnectedCallback", "adoptedCallback", and "attributeChangedCallback". The corresponding values are either a Web IDL Function callback function type value, or null. By default the value of each entry is null.

A construction stack

A list, initially empty, that is manipulated by the upgrade an element algorithm and the HTML element constructors. Each entry in the list will be either an element or an *already constructed* marker.

Also custom elements registry references as JS customElements

A Custom Element

```
class LoginElement extends HTMLElement {
    constructor() {
        super(); // Always first, establish prototype chain
    }

    connectedCallback() {
        …
        this.attachShadow({mode: 'open'});
        …
    }

    render(){
        this.shadowRoot.innerHTML = `<div id='content'>
                    <p>Please log in </p>
                <table>
                    <tr><td> …`
} // End class
// Register
customElements.define('log-in', LoginElement);
// In page
<log-in></log-in>
```

Extend existing element

This is ES6

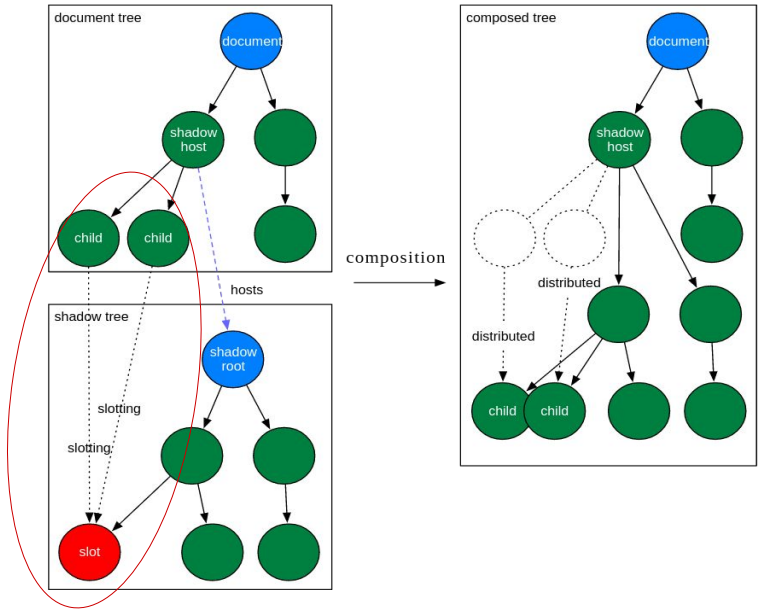
Register (must have “-” in name)

Template tag

```
<template id="login-template">
      <style>
      button {
        color: var(--button-text-color, pink);  <!-- Problems -->
        font-family: var(--button-font);
        /*font-size: 24px;*/
      }
      </style>
      <div id='log-in'>
      <p>Please log in </p>
      <table>
      <tr>
        <td>Email</td>
        <td><input id='email' type='text'/></td>
      </tr>
      <tr>
        <td>Password</td>
        <td><input id='passwd' type='text'/></td>
      </tr>
      <tr><td><button id="btnLogin" type='button' />Login</td></tr>
    </table>
    </div>
    </template>
```

-

Slots


document tree
document
shadow host
child
child
hosts
shadow tree
shadow root
slotting
slotting
slot
composition
composed tree
document
shadow host
distributed
distributed
child
child

–

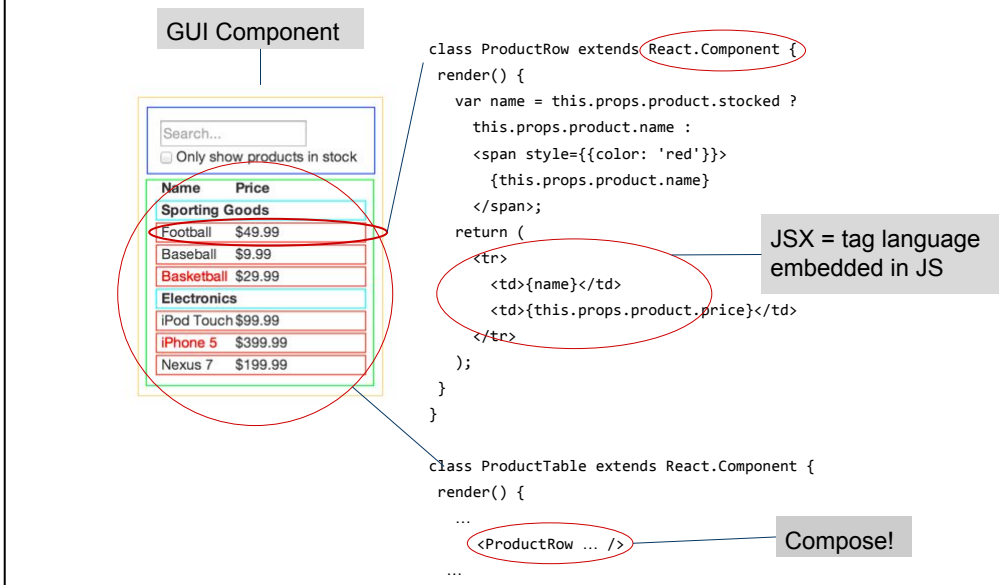
Slots

```
# shadowroot
<div id="tabs">
   <slot id="tabsSlot" name="title"></slot>
</div>
<div id="panels">
    <slot id="panelsSlot"></slot>
</div>

<!-- In page -->
<fancy-tabs>
  <button slot="title">Title</button>
  <button slot="title" selected>Title 2</button>
  <button slot="title">Title 3</button>
  <section>content panel 1</section>
  <section>content panel 2</section>
  <section>content panel 3</section>
</fancy-tabs>
```

-

React GUI Components



React is a GUI component framework (not full MVC framework)

Readings

- [React]

Props and State

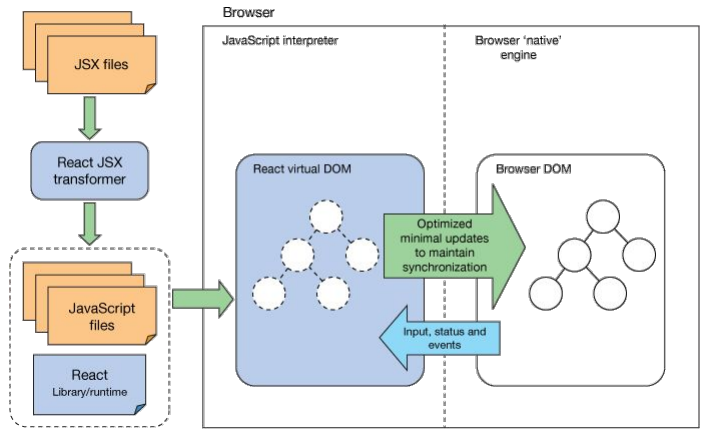
```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is
            {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Parameters to component (not used here)

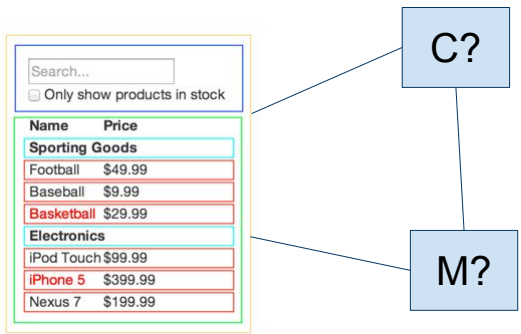
State of component (view model)

React Internals



-

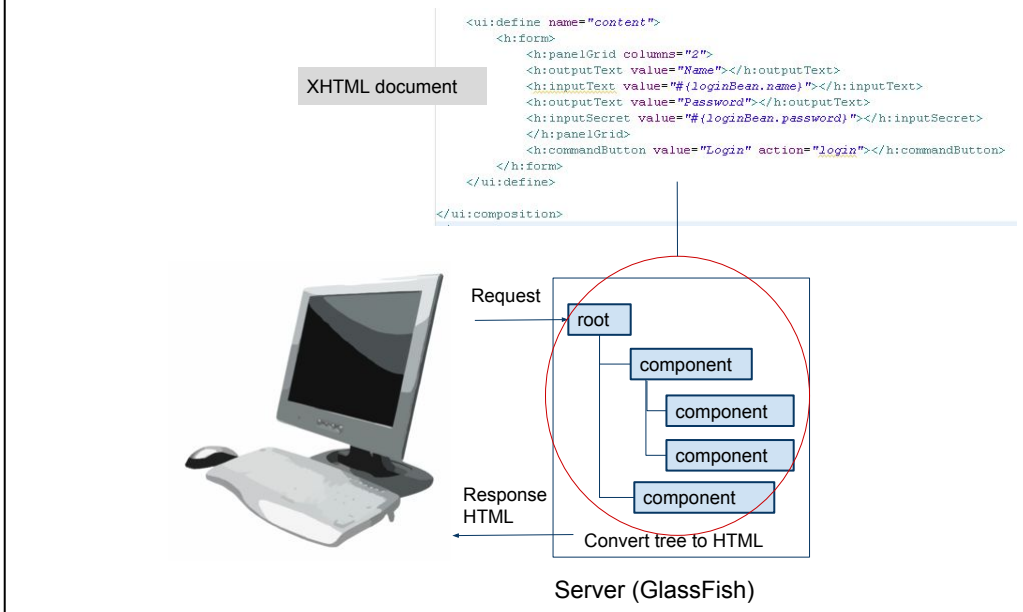
React MVC



Probably problems with Backbone (events)?

- React and Backbone
- React and .NET
- Interesting critical article

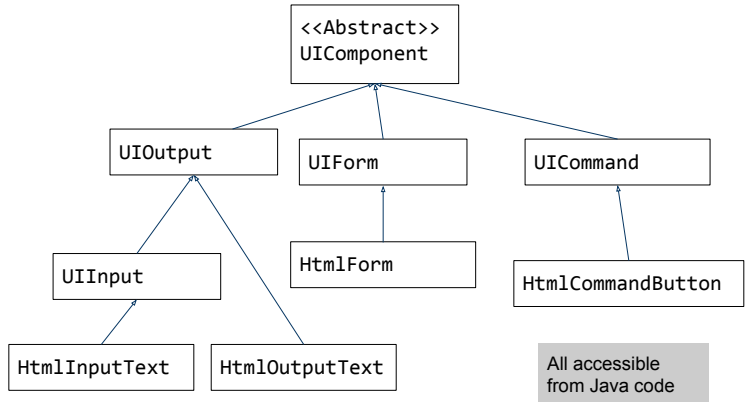
Java Server Faces



JSF is a server side (client) GUI technique

- Component tree “coded” as xhtml document (tedious to code in pure Java, … but possible)
- NOTE: xhtml pages NOT returned!

JSF Component Classes

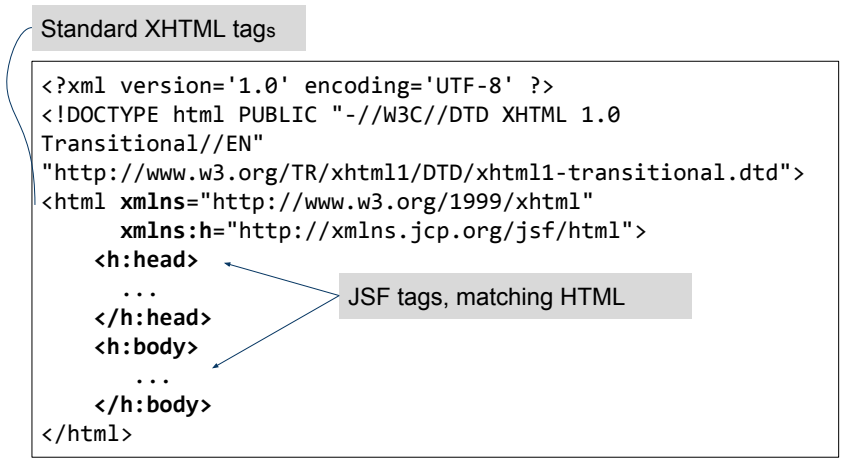


Many more classes ….

Server side component classes (tree of component objects),

- Toplevel is UIComponent (similar to Swing)
- Incoming data from HTTP-request will be converted and stored in the components
- Outgoing data retrieved from components
- A renderkit to translate components (and data) to renderable format (spec. requires a HTML renderkit).
  - Handled transparently by JSF
  - Example: HtmlCommandButton object → <input type="submit" … > (String)

A JSF Page



Readings

- JSF HTML tag reference
- JSF implicit objects

Post and Get

POST

```
<h:form>
   ...
   <h:commandButton ... />
   <h:commandLink action=.../>
</h:form>
```

GET

```
<h:link outcome=.../>
<h:button outcome=.../>
```

To send a POST request use

- commandButton or commandLink
- Must be inside <h:form…>
  - And so must the input controls too (<h:inputText…>, etc.)
- NOTE: <h:form> has no action attribute. Will “post back” to same page

To send a GET use

- link or button
- No need for <h:form …>

Important, POST and GET will control behaviour of JSF, see JSF Request Cycle, later …

Expression Language in JSF

EL as read

```
<h:outputText value="#{person.name}"/>
```

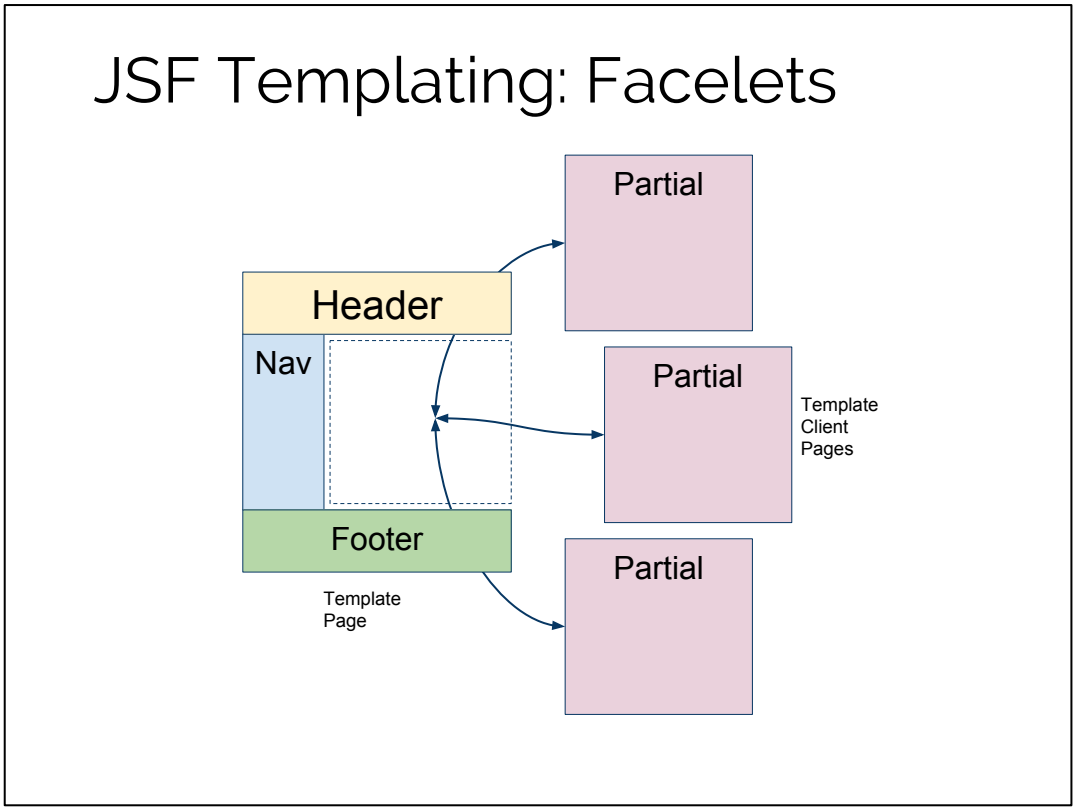
EL as write

```
<h:inputText value="#{person.name}"/>
```

A CDI bean (Java object handled by Context and dependency injection)

Same idea as JSP. Used as in JSPs but

- … uses deferred syntax
- **#**{ … expression … }
  - Deferred is both read and write (depends on expression location)
  - Input vs output component
- JSP used immediate syntax ${…expression…}
  - Immediate syntax is read only

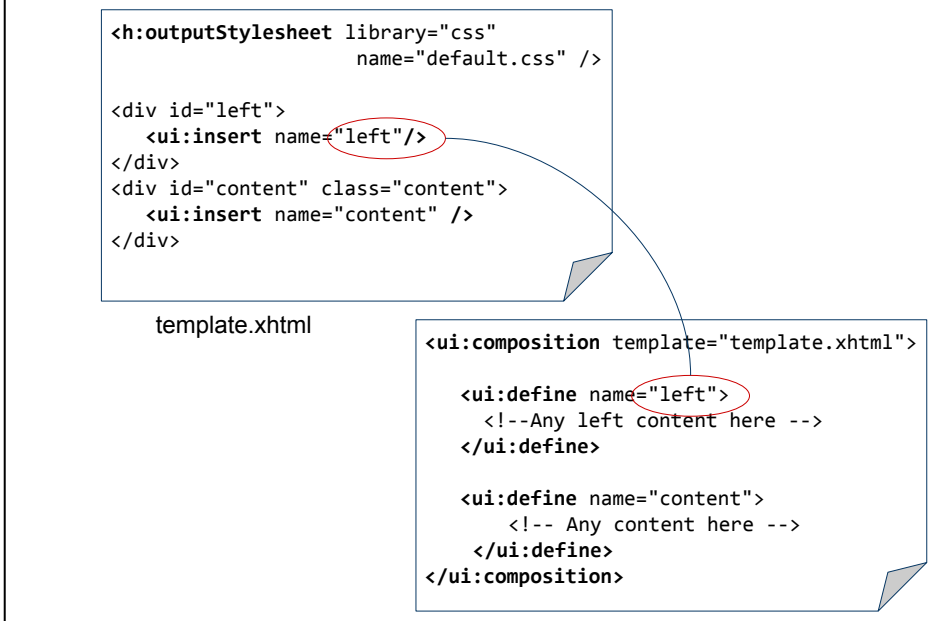


Facelets refers to the templating technique (view declaration language) for JavaServer Faces technology

Readings

- Facelets tag reference

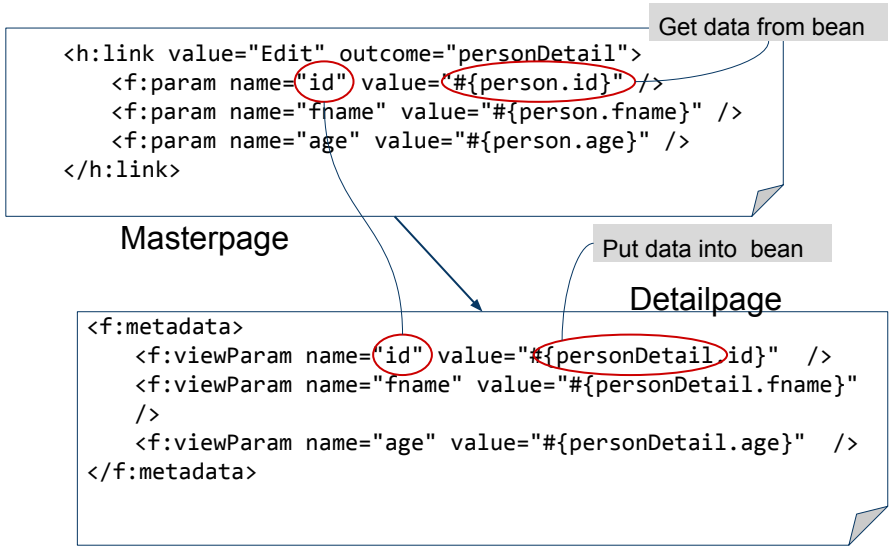
Template and Client



User never access template file (page) directly

- Request URL is the client file (page)

Master Detail Interface



HTML and JSF

Pass through elements

```
<input type="email" placeholder="Enter email"
    jsf:value="#{htmlbean.email}" required="required"/>

<label jsf:value="#{htmlbean.email}" />
```

Pass through attributes

```
<h:inputText id="email" value="#{htmlbean.email}"
        p:type="email" p:placeholder="Enter email"/>
```

Possible to let JSF and HTML work in conjunction (HTML5 friendly JSF)

Pass-through elements

- In HTML5 (i.e. no JSF-elements) use JSF attributes (and EL expressions)
- Make it possible to build the view with HTML elements that can access components and beans
  - Will convert HTML to JSF
  - Mapping between HTML-elements and JSF in TagDecorator
- To make an element a pass-through element, at least one of its attributes must be in the namespace http://xmlns.jcp.org/jsf.

Pass-through attributes

- In JSF (using JSF-elements), use HTML5 attributes
- Attributes passed through JSF directly to render phase
- Prefix attributes with namespace http://xmlns.jcp.org/jsf/passthrough

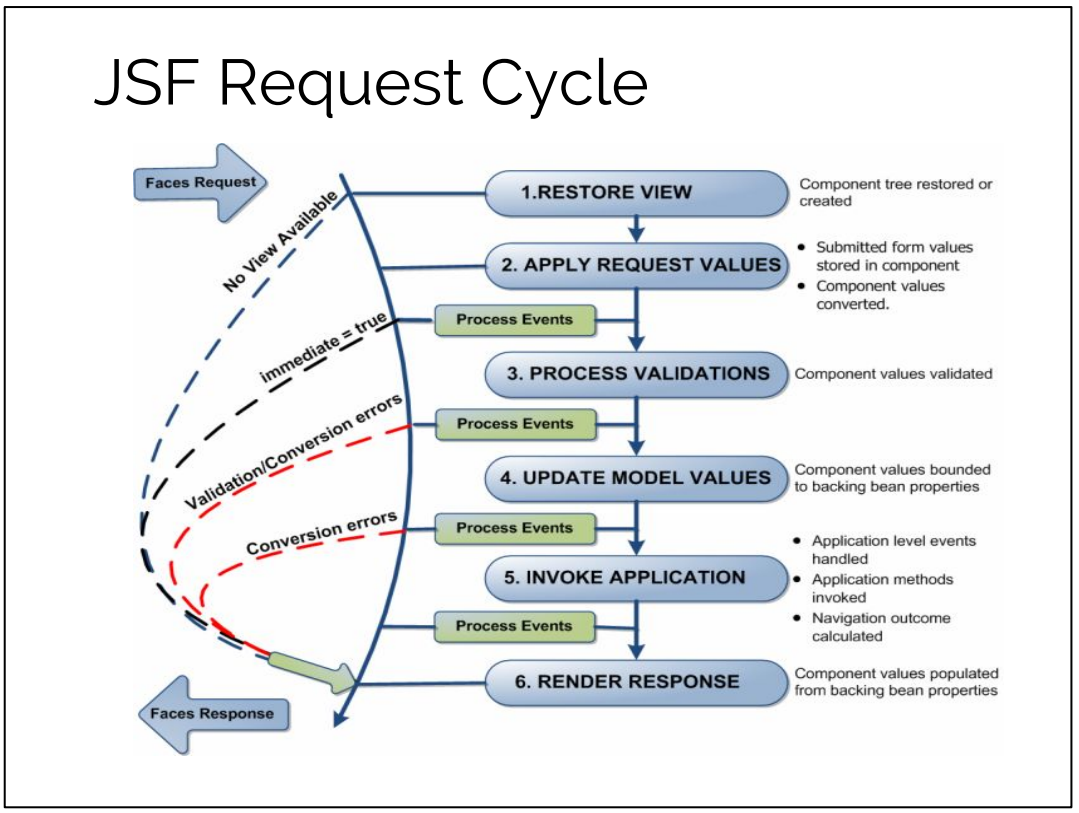
AJAX Behaviour

The AJAX element

```
<h:form>
    <h:inputText value="#{ajaxbean.bound}" />
    <h:commandButton … >
        <f:ajax render="output" execute="@form" />
    </h:commandButton>
    <h:outputText id="output" value="#{ajaxbean.randnumb}" />
</h:form>
```

JSF is default “full page load”

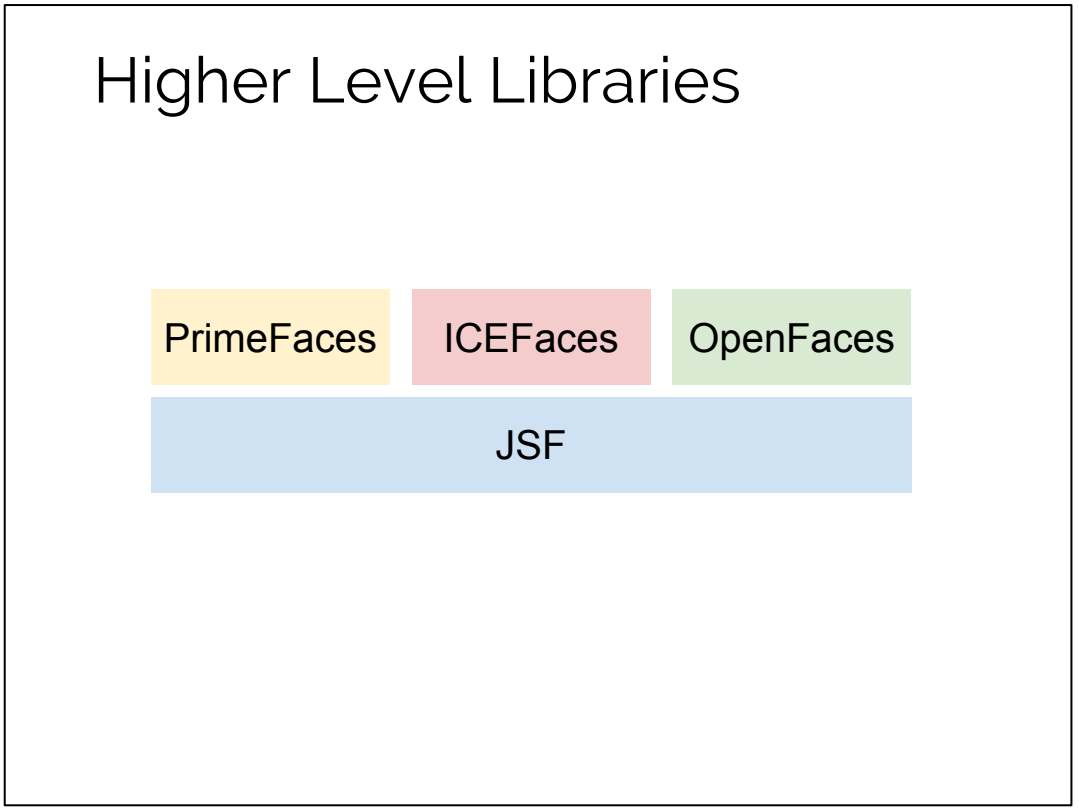
- Need for AJAX …?? If so … <f:ajax>-element
- There are 4 special values for attributes execute and render
  - @this. The element enclosing f:ajax.
  - @form. The h:form enclosing f:ajax.
    - Very convenient if you have multiple fields to send
  - @none. Nothing sent.
    - Useful if the element you render changes values each time you evaluate it.
  - @all. All JSF UI elements on page.



To emulate desktop programming (but having a network in between) things must be done in an orderly fashion

Readings

- JSF request cycle
- more about the same,



Readings

- PrimeFaces
- ICEFaces
- OpenFaces

Context and Dependency Injection

```
import javax.inject.Inject;

private PersonBean pb;

@Inject
public void setPersonBean(PersonBean pb) {
        this.pb = pb;
}
```

Get an instance "for free"

Readings

- Contexts and Dependency Injection (CDI)
- Weld reference implementation

CDI Bean

```
import javax.enterprise.context.RequestScoped;

@Named("personbean")
@RequestScoped
public class PersonBean implements
                              Serializable {

    private int age;

    public int getAge(){ … }
    public void setAge(int age){ … }
}
```

Name in pages

CDI Beans as Listeners

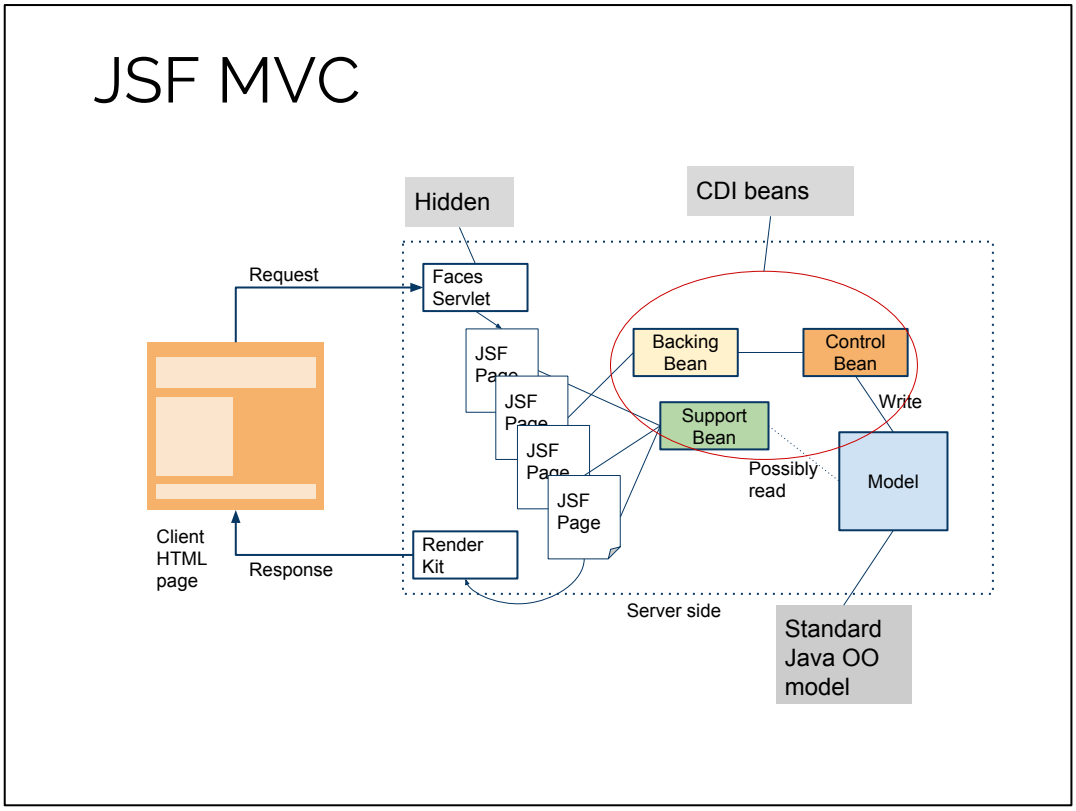
Call methods on CDI beans

```
<h:commandButton ...
         actionListener="#{bean.doActionListener}".../>


<h:commandButton …
               action="#{bean.anyMethod}" />

<h:inputText … valueChangeListener="#{bean.doValueChange}" />
```

Must return String, used for navigation



JEE MVC Java Server Faces for GUI

- Backing bean (Viewmodel): Holder for data. 1:1 with some page, request scoped
- Control bean: In control layer, transfer data from/to model. Act as listener (action attribute in element) , request scoped
- Support bean: Supply data common to more pages, session or application scoped

Readings

- CDI bean roles