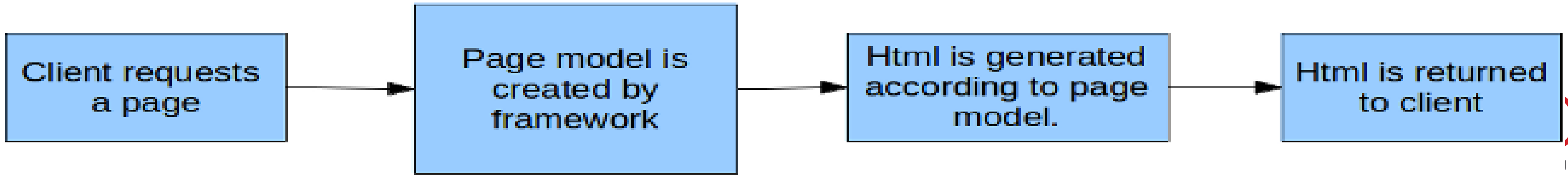
APACHE WICKET™

Introducing Apache Wicket

What is Wicket?

- Wicket is a web framework. More precisely is a *component-oriented* framework.
- Component oriented frameworks differ from classic web frameworks by building a model of requested page on the server side and generating the HTML to send back according to this model. You can think of the model as if it was an "inverse" JavaScript DOM, meaning that:

1) is built on server-side
2) is built before HTML is sent to client
3) HTML code is generated using this model and not vice versa.


Client requests a page
Page model is created by framework
Html is generated according to page model.
Html is returned to client

TextField component

- Text fields elements are handled in Wicket with class *TextField*
- *Example for a TextField:*

Java

```
new TextField<String>(“username”,Model<String>.of(“Insert   username”));
```

HTML

```
Username: <input type="text" wicket:id="username"/>
```

- Now we gonna look at a classic example of form to sign in a user with a username and password.

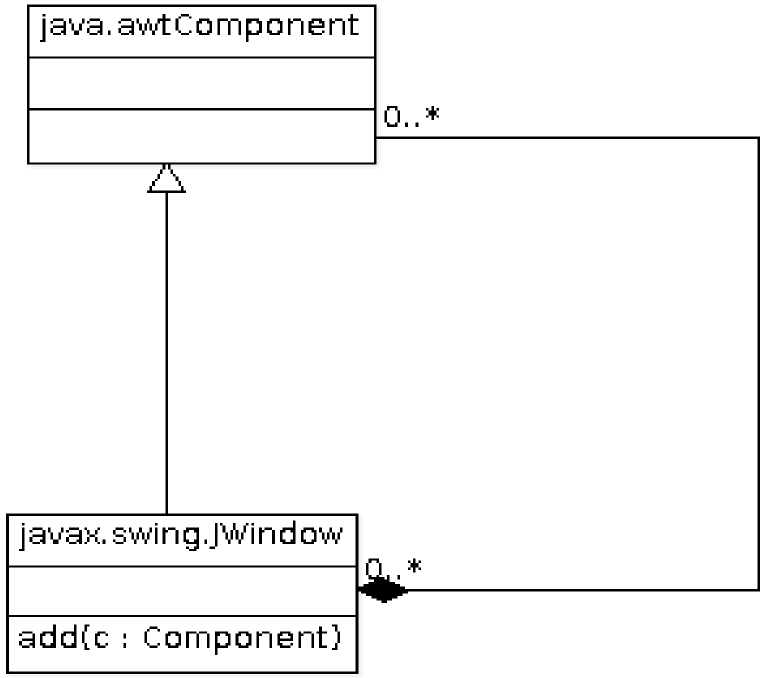
* The example code for the two versions of the form is available on GitHub under module LoginForm and LoginFormRevisited.

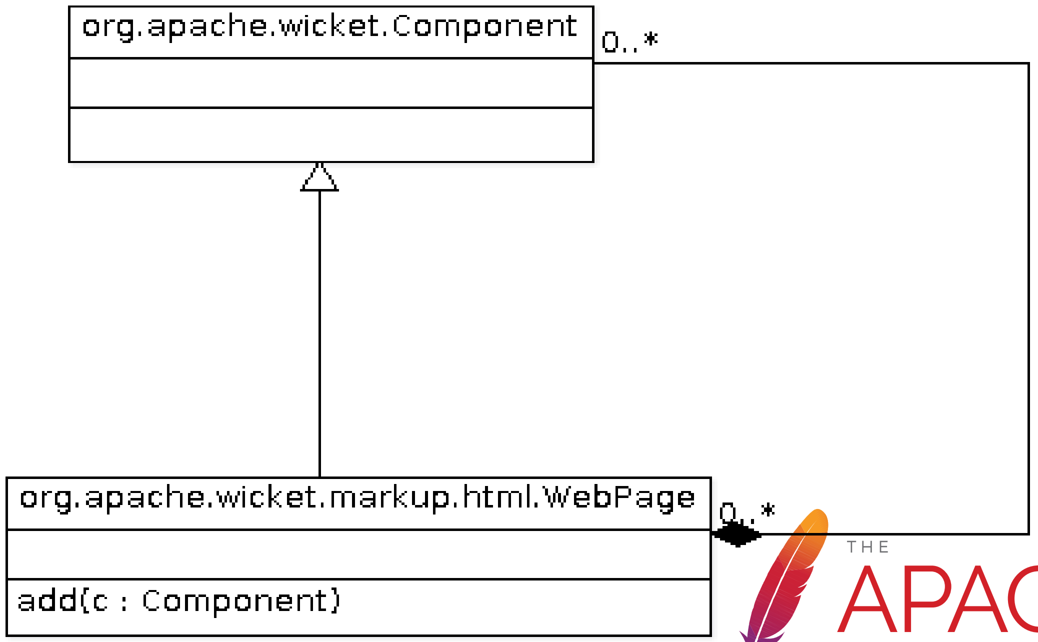
Component oriented framework

- With this kind of framework our web pages and their HTML components (forms, input controls, links, etc...), are pure class instances. Since pages are class instances they live inside the JVM heap and we can handle them as we do with any other Java class.
- This approach is very similar to what GUI frameworks (like Swing or SWT) do with desktop windows and their components. Wicket and the other component oriented frameworks bring to web development the same kind of abstraction that GUI frameworks offer when we build a desktop application.
- This kind of framework hides the details of the HTTP protocol and naturally solves the problem of its stateless nature.

Component oriented framework

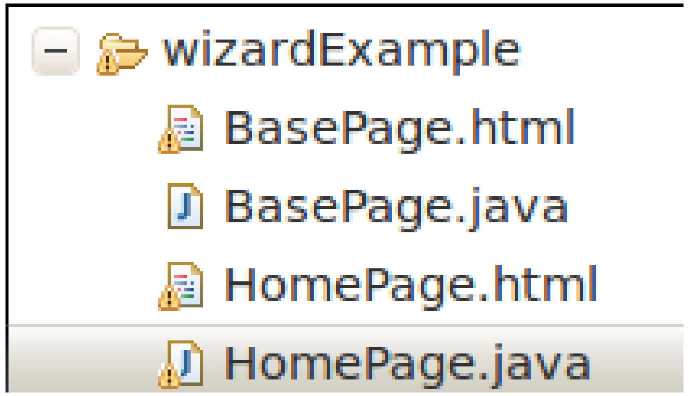
- Wicket allows us to design our web pages in terms of components and containers, just like AWT does with desktop windows. Both frameworks share the same component-based architecture: in AWT we have a Windows instance which represents the physical windows containing GUI components (like text fields, radio buttons, drawing areas, etc...), in Wicket we have a WebPage instance which represents the physical web page containing HTML components (pictures, buttons, forms, etc...) .




org.apache.wicket.Component
0..*
org.apache.wicket.markup.html.WebPage
add(c : Component)
0..*

Where is the HTML?

- By default this HTML file must have the same name of the related page class and must be in the same package:



```
wizardExample
    BasePage.html
    BasePage.java
    HomePage.html
    HomePage.java
```

- In Wicket we can use page classpath to put any kind of resource, not just HTML (pictures, properties file, etc...)

The web.xml file

- A Wicket application is a standard Java EE web application, hence it is deployed through a web.xml file placed inside folder WEB-INF:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
      "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <display-name>Wicket HelloWorld</display-name>
    <filter>
        <filter-name>WizardApplication</filter-name>
        <filter-class>
           org.apache.wicket.protocol.http.WicketFilter
        </filter-class>
        <init-param>
          <param-name>applicationClassName</param-name>
          <param-value>helloWorld.WicketApplication</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>WizardApplication</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Class WicketApplication

- If we look back at web.xml we can see that we have provided the Wicket filter with a parameter called applicationClassName. This subclass represents our web application built with Wicket and it's responsible for configuring it when the server is starting up.
- Class Application comes with a set of configuration methods that we can override to customize our application's settings. One of these methods is getHomePage() that must be overridden as it is declared abstract:

```
package helloWorld;

import org.apache.wicket.Page;
import org.apache.wicket.protocol.http.WebApplication;

public class WicketApplication extends WebApplication {

    @Override
    public Class<? extends Page> getHomePage() {
        return HomePage.class;

    }

   @Override
    Public init(){...}
}
```

HelloWorld page

- To map Wicket components to HTML tags we must use attribute **wicket:id**.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <h1 wicket:id="label"></h1>
</body>
</html>
```

HelloWorld.html

- In the Wicket page we must *add* a component with the id equal to the value of the **wicket:id** previously set:

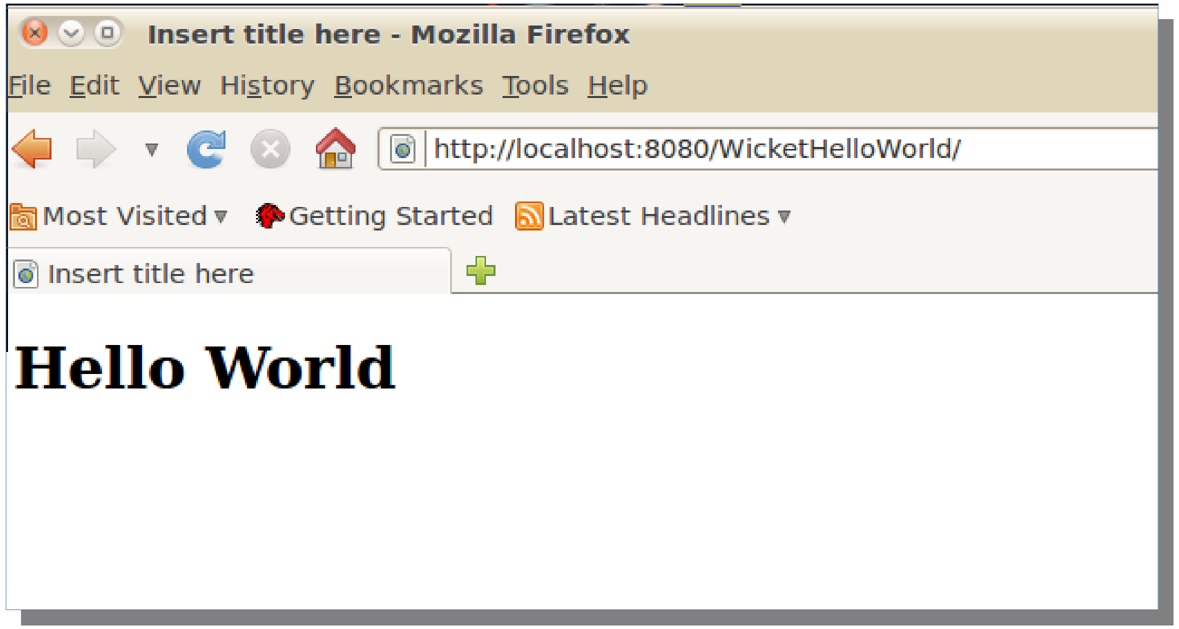
```
package helloWorld;

import org.apache.wicket.markup.html.WebPage;
import org.apache.wicket.markup.html.basic.Label;

public class HomePage extends WebPage {
    public HomePage(){
        super();
        add(new Label("label", "Hello World"));
    }
}
```

HelloWorld.java

Wicket “Hello world”


Insert title here - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/WicketHelloWorld/
Most Visited
Getting Started
Latest Headlines
Insert title here

Hello World

- Inside <h1> tag you will find the value expressed as second parameter for Label’s constructor.

```
...
<body>
    <h1 wicket:id="label">
    </h1>
</body>
...
...
    public HomePage(){
        super();
        add(new Label("label",
                  "Hello World"));
    }
...
```

Wicket links

- In HTML a link is basically a pointer to another resource that most of the time is another page. Wicket implements links with component *org.apache.wicket.markup.html.link.Link*, but it's more like *onClick* event listener than a link:

```java
public class HomePage extends WebPage {
    public HomePage(){
        add(new Link("id"){
            @Override
            public void onClick() {

            }
        });
    }
}
```

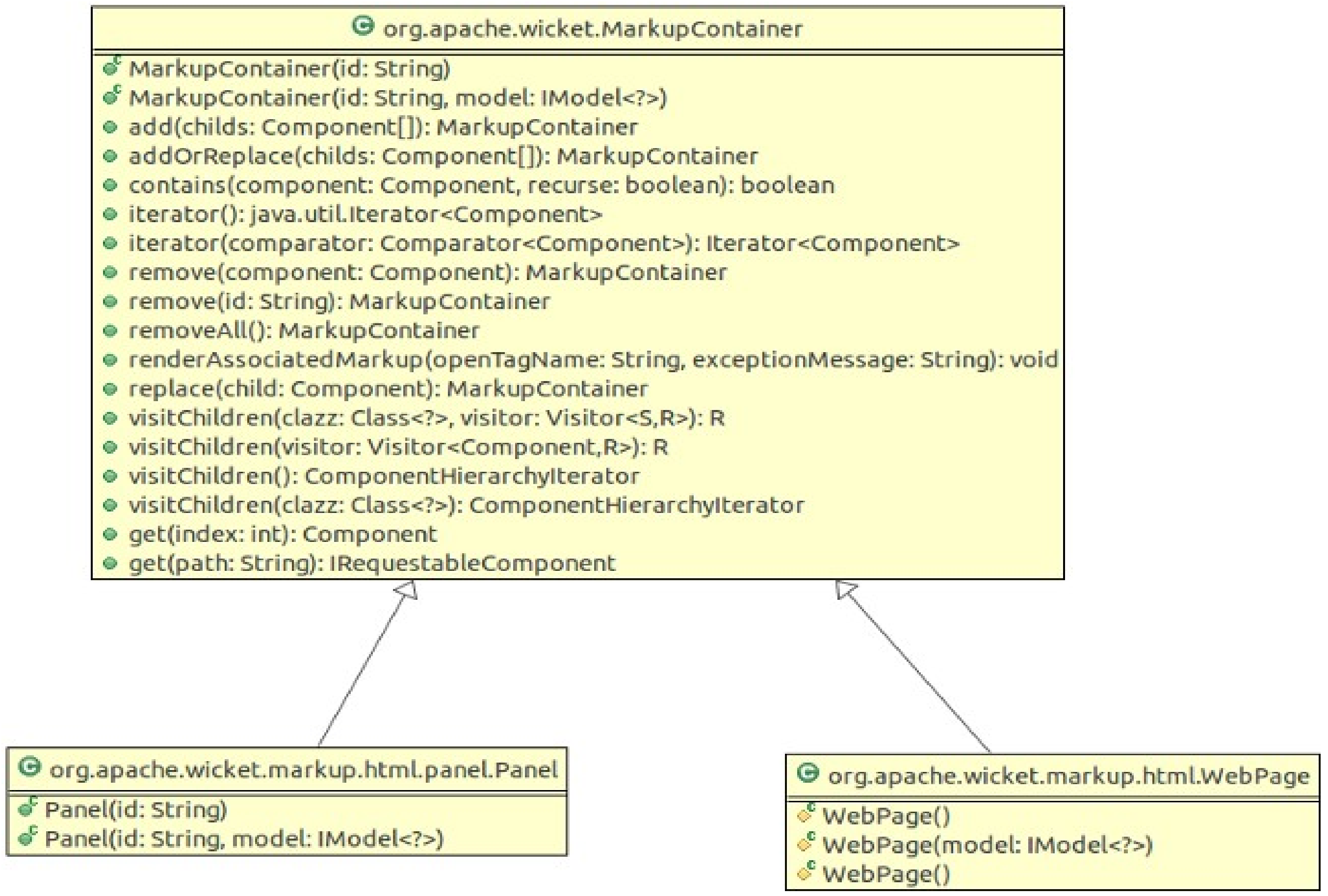
Wicket links

- By default after *onClick* has been executed, Wicket will send back to the current page to the client web browser. If we want to navigate to another page we must use method *setResponsePage* of class *Component*:

```
public class HomePage extends WebPage {
    public HomePage(){
        add(new Link("id"){
            @Override
            public void onClick() {
                //we redirect browser to another page.
                setResponsePage(AnotherPage.class);
            }
        });
    }
}
```

Wicket pages and panels

- Class *org.apache.wicket.markup.html.panel.Panel* is a special component which lets us reuse GUI code and HTML markup across different pages and different web applications. It shares a common ancestor class with *WebPage* class, which is *org.apache.wicket.MarkupContainer*:


org.apache.wicket.MarkupContainer
MarkupContainer(id: String)
MarkupContainer(id: String, model: IModel<?>)
add(childs: Component[]): MarkupContainer
addOrReplace(childs: Component[]): MarkupContainer
contains(component: Component, recurse: boolean): boolean
iterator(): java.util.Iterator<Component>
iterator(comparator: Comparator<Component>): Iterator<Component>
remove(component: Component): MarkupContainer
remove(id: String): MarkupContainer
removeAll(): MarkupContainer
renderAssociatedMarkup(openTagName: String, exceptionMessage: String): void
replace(child: Component): MarkupContainer
visitChildren(clazz: Class<?>, visitor: Visitor<S,R>): R
visitChildren(visitor: Visitor<Component,R>): R
visitChildren(): ComponentHierarchyIterator
visitChildren(clazz: Class<?>): ComponentHierarchyIterator
get(index: int): Component
get(path: String): IRequestableComponent
org.apache.wicket.markup.html.panel.Panel
Panel(id: String)
Panel(id: String, model: IModel<?>)
org.apache.wicket.markup.html.WebPage
WebPage()
WebPage(model: IModel<?>)
WebPage()

Wicket panels

- Both Panel and WebPage have their own associated markup file which is used to render the corresponding component. When a panel is attached to a container, the content of its markup file is inserted into its related tag.

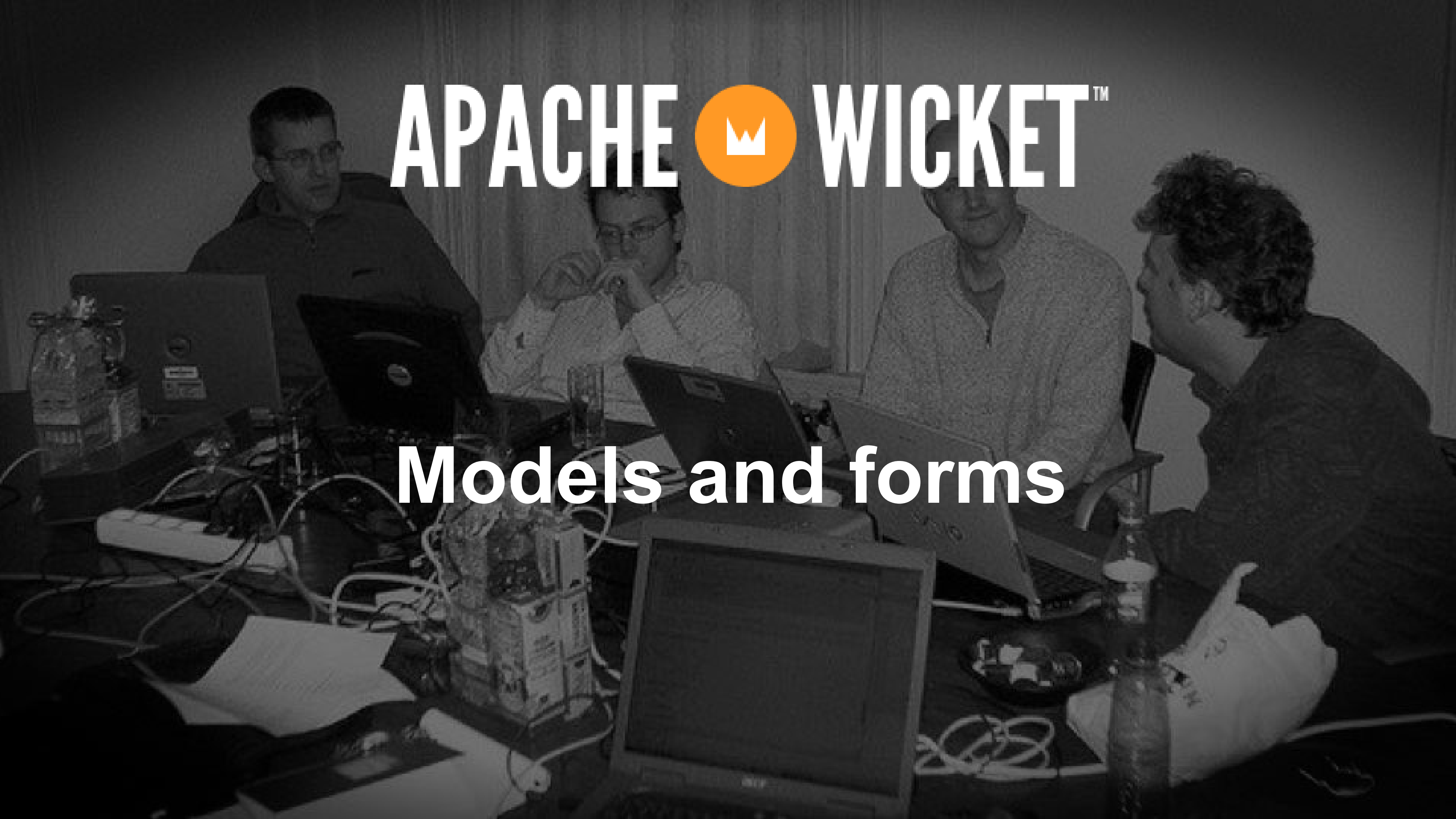
```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
…
</head>
<body>
   <wicket:panel>
      <!-- Your markup goes here -->
   </wicket:panel>
</body>
</html>
```

- The HTML outside tag <wicket:panel> will be removed during rendering phase. The space outside this tag can be used by both web developers and web designers to place some mock HTML to show how the final panel should look like.

JavaScript and CSS in panels

- If we want to add header resources to a panel (for example a css), we can use tag ***<wicket:head>***. It’s content will be included in the <head> tag of the page. **NOTE:** resources are added *once per panel class* to avoid duplicates.

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Insert title here</title>
<wicket:head>
   <script>...</script>
<wicket:head>
...
</head>
<body>
...
<wicket:panel>
…
</wicket:panel>
...
```



APACHE WICKET™

Models and forms

Model in Wicket

- A model is a *facade* for the data object that a component must display (for example with a label) or set (for example with a textfield).
- Every component has a related model that can be accessed with method *getModel()*. A model can be shared among different components.
- Up to Wicket 7, a model was a simple interface that just defined a method to *read* an object and another one to *write* it:

```
public interface IModel
{
        public Object getObject();
        public void setObject(final Object object);
}
```

- With models our components can handle data without knowing how they are physically persisted or retrieved.

Model in Wicket

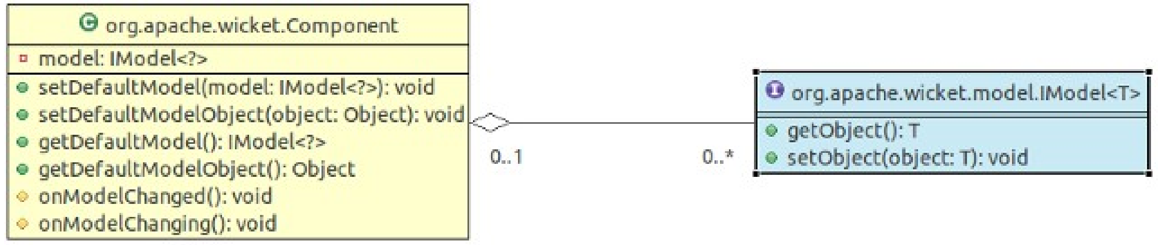
- Also Label has a model, but it's “hidden” inside the component and contains the second parameter of the constructor:

```
add(new Label("label", "Hello World"));
...
```

- Wicket comes with a set of models suited for different needs. The most basic one is class Model. We can wrap any object in this model and we can use factory method *of* to avoid explicit instantiations:

```
new Model<String>("label");
…
Model<Persona>.of(new Person(“Mario”, “Rossi”));
…
```

- Every component has a set of method to access its model and the object inside it:



org.apache.wicket.Component
model: IModel<?>
setDefaultModel(model: IModel<?>): void
setDefaultModelObject(object: Object): void
getDefaultModel(): IModel<?>
getDefaultModelObject(): Object
onModelChanged(): void
onModelChanging(): void
0..1
0..*
org.apache.wicket.model.IModel<T>
getObject(): T
setObject(object: T): void

Model in Wicket 8

- With Wicket 8 model interface has changed to take advantage of the new syntax features introduced in Java 8.
- In short, *IModel* has become a functional interface and provides a default empty implementation for *setObject*:

```
public interface IModel
{
         public Object getObject();
         default void setObject(final T object) {
                 throw new UnsupportedOperationException(
             "Override this method to support setObject(Object)");
          }
}
```

- In this way models are *read-only* by default, and can be implemented with lambda expressions:

```
add(new Label("timeStamp", () -> LocalDate.now()));
```

Models and forms

- In Wicket the concept of model is probably the most important topic of the entire framework and it is strictly related to the usage of its components.
- In addition, models are also an important element for localization support (*see user guide for more details*).
- However, despite their fundamental role, models are not difficult to understand but the best way to get acquainted with them is to use them with forms.
- Hence, to continue our introduction to Wicket models, in the next slides we will introduce Wicket forms a very basic form component, the *TextField*.

Forms in Wicket

- Following its component oriented nature, Wicket offers a *Form* component to handle HTML forms.
- Forms are special container for input components (representing text fields, radio buttons, check boxes, etc...) which are subclasses of *org.apache.wicket.markup.html.form.FormComponent*.
- Class Form comes with a callback method *onSubmit* which is invoked when form is submitted.

```
public class LoginForm extends Form {

   public final void onSubmit() {
      ...
}
```

TextField component

- Text fields elements are handled in Wicket with class *TextField*

- *Example for a TextField:*

  Java

  ```
  new TextField<String>(“username”,Model<String>.of(“Insert
            username”));
  ```

  HTML

  ```
  Username: <input type="text" wicket:id="username"/>
  ```

- Now we gonna look at a classic example of form to sign in a user with a username and password.

* The example code for the two versions of the form is available on GitHub under module LoginForm and LoginFormRevisited.

Example form: the markup

```
<html>
…
<div style="margin: auto; width: 40%" class="">
<form  id="search" method="get" wicket:id="form">
<fieldset class="center">
    <legend >Search</legend>
    <p wicket:id="loginStatus" style=""></p>
    <span>Username: </span>
        <input wicket:id="username" type="text" id="username" />
<br/>
    <span >Password: </span>
        <input wicket:id="password" type="password" id="password" />
<p>
       <input type="submit" name="login" value="login"/>
</p>

</fieldset>
</form>
</div>
…
</html>
```

4 components: a form, a label, a text field and a password field.

Wicket comes with a specific component for password fields.

Example form: first version

```
public class LoginForm extends Form {
    private TextField usernameField;
    private PasswordTextField passwordField;
    private Label loginStatus;

    public LoginForm(final String componentName) {
        super(componentName);

        usernameField = new TextField("username", new Model<String>(""));
        passwordField = new PasswordTextField("password", new
              Model<String>(""));
        loginStatus = new Label("loginStatus");

        add(usernameField);
        add(passwordField);
        add(new Label("message", "Login"));
        add(loginStatus);

    }

    public final void onSubmit() {
        String username = usernameField.getDefaultModelObject();
        String password = passwordField.getDefaultModelObject();

        if((username.equals("Mario") && password.equals("Rossi")))
            loginStatus.setDefaultModel(new Model<String>("Good!"));
        else
            loginStatus.setDefaultModel(new Model<String>("Username or password invalid!"));
}
```

Wicket comes with a specific component for password fields.

onSubmit is triggered when form is submitted.

Model CompoundPropertyModel

- *CompoundPropertyModel* is a particular kind of model which uses components ids to resolve properties on its model object.
- This can save us a lot boilerplate code if we choose components ids according to properties name:

Data model:

```java
public class Person {
 private String name;
 private String surname;
 private String address;
 private String email;
}
```

Display data model:

```java
//Create a person named 'John Smith'
Person person = new Person("John", "Smith");
setDefaultModel(new
CompoundPropertyModel(person));
add(new Label("name"));
add(new Label("surname"));
add(new Label("address"));
add(new Label("email"));
```

Example form: second version

- *CompoundPropertyModel* can drastically improve the code of our example form if we use the form *itself* as data object.
- We just have to add to the form class the fields we need to use inside *onSubmit()*:

```java
class LoginForm extends Form{
    private String username;
    private String password;
    private String loginStatus;

    public LoginForm(String id) {
        super(id);
        setDefaultModel(new CompoundPropertyModel(this));

        add(new TextField("username"));
        add(new PasswordTextField("password"));
        add(new Label("loginStatus"));
    }

    public final void onSubmit() {
        if(username.equals("test") && password.equals("test"))
            loginStatus = "Congratulations!";
        else
            loginStatus = "Wrong username or password !";
    }
}
```

The form itself is the model object!

Components ids and fields names are the same.

Model has become completely *transparent* to the developer.

More about forms and models

- Forms are a wide-ranging topic and can not be fully covered with this presentation.
- There’s also much more to say about models, especially if you use them in Wicket 8 with lambda support.
- For a full coverage of forms and models see the user guide:

https://wicket.apache.org/learn/#guide

APACHE WICKET™

Resource handling

Resource handling

- With “resource” we indicate both static resources (such as JS and CSS files) and dynamic resources (those who returns they value on the fly [ex: a RSS]).
- From a technical point of view, in Wicket a resource is just an implementation of interface *org.apache.wicket.request. resource.IResource*.
- Working with dynamic resources is less frequent in Wicket and it implies a custom implementation of *IResource*.
- On the contrary handling static resources is a very common task. With Wicket we can specify not just the static resource to load, but also its dependencies and its loading priority.

Static resources

- In general static resources are loaded from 3 possible sources:
  - A generic file from local filesystem
  - A file from classpath (via ClassLoader)
  - An URL
- In Wicket resources are instantiated using a *reference* to them rather than directly. In this way they can be lazy-loaded the first time they are requested.
- Resource references are instances of class *org.apache.wicket. request.resource.ResourceReference*.
- Most of the time static resources are JS or CSS files which can be referred to as *header items.*

Header items

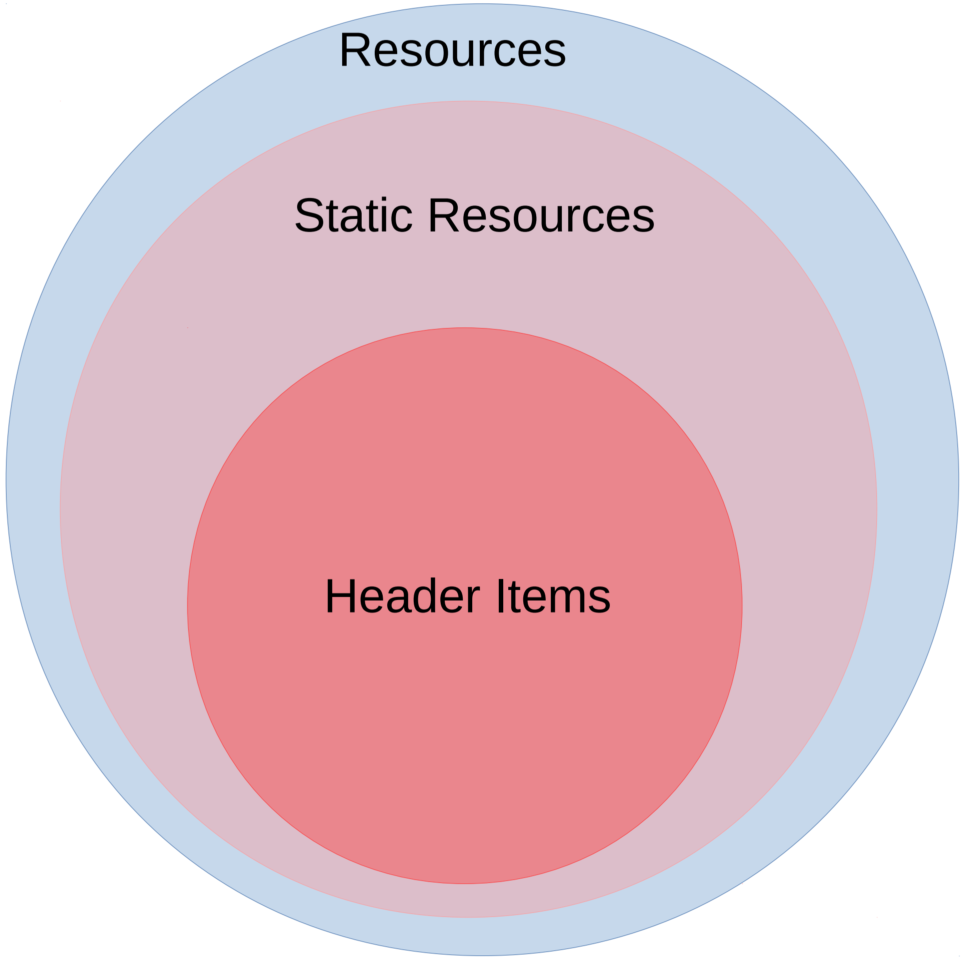
- As the name suggests, an header item is simply an element that is placed inside the <head> tag of the page. In Wicket header items are usually built from a resource reference and are instances of class *org.apache.wicket.markup.head.HeaderItem* (for example *JavaScriptHeaderItem*)

- A page or one of its component can add an header item overriding its method *renderHead*:

```
class MyPanel extends Panel {

  public void renderHead(IHeaderResponse response) {
    response.render(JavaScriptHeaderItem.forUrl("https://code.jquery.com/"
            + "jquery.min.js"));
    response.render(JavaScriptHeaderItem.forScript("alert('page loaded!');"));
  }
}
```

Ok, let’s recap….



Resources: ANY kind of resources. i.e. both dynamic (RSS, dynamic PDF, etc…) and static (JS, CSS, pictures, etc…)

Static Resources: usually loaded from files (JS, CSS, pictures, etc…)

Header Items: those resources that must be placed in the header section (aka <head> tag). JS, CSS, script sections, etc...

Built-in Header Items and dependencies

- **CssHeaderItem**: for CSS content.
- **JavaScriptHeaderItem**: for JavaScript content.
- **StringHeaderItem**: render free text in the header section.

- As we said before, we can declare dependencies on header items and resources:

```
Url jqueyuiUrl = Url.parse("https://ajax.googleapis.com/ajax/libs/jqueryui/" +
                "1.10.2/jquery-ui.min.js");

UrlResourceReference jqueryuiRef = new UrlResourceReference(jqueyuiUrl){
  @Override
  public List<HeaderItem> getDependencies() {
    Application application = Application.get();
    ResourceReference jqueryRef = …;

    return Arrays.asList(JavaScriptHeaderItem.forReference(jqueryRef));
  }
};

JavaScriptReferenceHeaderItem javaScriptHeaderItem =
                                        JavaScriptHeaderItem.forReference(jqueryuiRef);
```

Priority Header Item

- **PriorityHeaderItem**: wraps another header item and ensures that it will have the priority over the other items.

```
Url jqueyuiUrl = Url.parse("https://ajax.googleapis.com/ajax/libs/"
                     + "jqueryui/1.10.2/jquery-ui.min.js");
UrlResourceReference jqueryuiRef = new
                     UrlResourceReference(jqueyuiUrl);
JavaScriptReferenceHeaderItem javaScriptHeaderItem =
                     JavaScriptHeaderItem.forReference(jqueryuiRef);

PriorityHeaderItem item = new
                     PriorityHeaderItem(javaScriptHeaderItem);
```

The item wrapped by PriorityHeaderItem will be contributed before any other non-priority item, including its dependencies.

Header Items for JavaScript

There are also header items meant to work with JavaScript events. In this way we can execute our code only when a specific event occurs.

- **OnDomReadyHeaderItem**: JavaScript code that will be executed after the DOM has been built, but before external files will be loaded.

```
OnDomReadyHeaderItem item = new OnDomReadyHeaderItem(";alert('hello!');");
```

- **OnLoadHeaderItem**: execute JavaScript code after the whole page is loaded.

```
OnLoadHeaderItem item = new OnLoadHeaderItem(";alert('hello!');");
```

- **OnEventHeaderItem**: execute JavaScript code when a specific event is triggered.

```
OnEventHeaderItem item = new OnEventHeaderItem("elementId",
                                               "eventName", ";alert('Hello!');");
```

Bundle resources

- To reduce the number of requests to the server, resources can be aggregated in *bundles*. A resource bundle can be declared during application initialization listing all the resources that compose it:

```
@Override
public void init()
{

   getResourceBundles()
        .addJavaScriptBundle(getClass(), "plugins-bundle.js",
            jqueryPlugin1, jqueryPlugin2, jqueryPlugin3
        );

}
```

Now, when one of the resources included in the bundle is requested, the entire bundle is served, i.e. the page will contain the JavaScript entry *plugins-bundle.js* , which includes all the bundle resources.

APACHE WICKET™

AJAX support

“Transparent” AJAX

Wicket simplifies AJAX development controlling via Java the following basic operations:

- Generate a page unique id for the DOM element of a component.
- Write a callback for an event triggered on the page or on a specific component (with AJAX behaviors).
- Refresh the HTML of a component.
- Execute JavaScript code as response to a specific event.

For the first three operations we won’t write a single line of JavaScript!

Handle AJAX events

- Now we can “ajaxify” components adding AJAX *behaviors*. In Wicket a behaviors are quite like *plug-ins* that can enrich a component with new features.
- For example *org.apache.wicket.ajax.AjaxEventBehavior* provides the means to handle an event on server side via AJAX:

Java Code

```java
Label label = new Label("label","Hello!");
label.setOutputMarkupId(true);

label.add(new AjaxEventBehavior("click") {
  @Override
  protected void onEvent(AjaxRequestTarget
                  target) {
    //Do my stuff...
  }
});
```

HTML Template

```html
<html>
<body>
    <span wicket:id="message">
      Message goes here
    </span>
</body>
</html>
```

AjaxRequestTarget is the main entity for AJAX development.

Refresh HTML and add JavaScript to response

- A common parameter for AJAX handler is *org.apache.wicket .ajax.AjaxRequestTarget*, which can be used to refresh component HTML:

```
protected void onEvent(AjaxRequestTarget
                   target) {
    target.add(panel);
}
```

- *AjaxRequestTarget* can be used also to enrich AJAX response with JavaScript code and header items:

```
protected void onEvent(AjaxRequestTarget
                   target) {
   target.appendJavaScript(";alert('hello!');");
   target.getHeaderResponse().render(headerItem);
}
```

Built-in AJAX components and behaviors

A number of ready-to-use components and behaviors are provided out of the box:

- AjaxLink
- AjaxButton
- AjaxCheckBox
- AutoCompleteTextField
- AjaxEventBehavior
- AjaxFormSubmitBehavior
- AbstractAjaxTimerBehavior
- ....

More examples at http://examples7x.wicket.apache.org/ajax/

Java lambdas: a match made in heaven

- Java 8 lambdas are quite suited for writing callback code, which is what we do to handle AJAX events.
- With Wicket 8 an AJAX *click* handler can be written leveraging lambdas in the following way:

```
AjaxEventBehavior.onEvent("click", target -> target.add(component));
```

APACHE WICKET™

Testing with Wicket

Test in isolation

- Test Driven Development (and unit testing) has become a fundamental activity in our everyday-job. Wicket offers a rich set of helper classes that allows us to test our applications in isolation using just JUnit.

- With “just JUnit” we mean:

1) We don’t need to have a running server
2) We don’t need to run tests for a specific browser (like we do with Karma)
3) No additional library required, just Wicket and JUnit (no need of browser automation tools like Selenium)

Test in isolation

- The central class in a Wicket testing is *org.apache.wicket.util.tester.WicketTester*. This utility class provides a set of methods to render a component, click links, check page content, etc...

```
public class TestHomePage {
    private WicketTester tester;

    @Before
    public void setUp() {
        tester = new WicketTester(new WicketApplication());
    }

    @Test
    public void testHomePageLink() {
            //start and render the test page
            tester.startPage(HomePage.class);
            //assert rendered page class
            tester.assertRenderedPage(HomePage.class);
            //move to an application link
            tester.executeUrl("./foo/bar");
            //test expected page for link
            tester.assertRenderedPage(AnotherHomePage.class);
    }
}
```

Testing the response

- *WicketTester* allows us to access to the last response generated during testing with method *getLastResponse*. Utility class *Mock-HttpServletResponse* is returned to extract informations from mocked request.

```
String responseContent = tester.getLastResponse().getDocument();
tester.assertContains("regExp");
```

- Resulting markup can be tested at tag-level with *TagTester*:

Testing the response

Response content:

```
<html xmlns:wicket="http://wicket.apache.org">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>
        <span class="myClass"></span>
        <div class="myClass"></div>
    </body>
</html>
```

JUnit code:

```
String responseContent = tester.getLastResponse().getDocument();
//look for a tag with 'class="myClass"'
TagTester tagTester = TagTester.createTagByAttribute(responseTxt,
                                "class", "myClass");

assertEquals("span", tagTester.getName());
List<TagTester> tagTesterList = TagTester.createTagsByAttribute(responseTxt,
                           "class", "myClass", false);

assertEquals(2, tagTesterList.size());
```

Testing AJAX events

- AJAX components can be tested as well “triggering” the JavaScript event they handle:

Page Code

```
Label label = new Label("label", "Hello World!");
Label otherLabel = new Label("otherLabel", "hola!");
label.setOutputMarkupId(true);

label.add(new AjaxEventBehavior("click") {
  @Override
  protected void onEvent(AjaxRequestTarget target) {
    target.add(otherLabel);
  }
});
```

Test Code

```
//simulate an AJAX "click" event
tester.executeAjaxEvent("label", "click");

//test other assertions…
```

Testing AJAX response

- The AJAX response can be tested with *WicketTester* to ensure that a specific component has been added (i.e. we want to refresh its markup):

Page Code

```
Label label = new Label("label", "Hello World!");
Label otherLabel = new Label("otherLabel", "hola!");
label.setOutputMarkupId(true);

label.add(new AjaxEventBehavior("click") {
  @Override
  protected void onEvent(AjaxRequestTarget target) {
    target.add(otherLabel);
  }
});
```

Test Code

```
//simulate an AJAX "click" event
tester.executeAjaxEvent("label", "click");
//test if AjaxRequestTarget contains a component (using its path)
tester.assertComponentOnAjaxResponse("otherLabel");
```

Testing AJAX behaviors

- AJAX behaviors can also be tested in isolation, relying only on *WicketTester:*

Test Code

```
AjaxFormComponentUpdatingBehavior ajaxBehavior =
             new AjaxFormComponentUpdatingBehavior("change"){
          @Override
          protected void onUpdate(AjaxRequestTarget target) {
             //...
          }
};

component.add(ajaxBehavior);

//execute AJAX behavior, i.e. onUpdate will be invoked
tester.executeBehavior(ajaxBehavior);
```

Other test goodies

WicketTester offers many more utilities for unit testing:

- Check component status (*assertEnabled, assertDisabled, assetVisible, assertInvisible*)
- Check component's model value (*assertModelValue*)
- Test forms with *FormTester*:

```
 FormTester formTester = tester.newFormTester("form");
//set credentials
formTester.setValue("username", username);
formTester.setValue("password", password);
//submit form
formTester.submit();
```

- …

Summary and references

- We had a quick “journey” through the life of Wicket.
- As for any other Open Source project, the health of its community is fundamental.
- We have seen some of the most appealing features, still there is lot more to discover!
- Learn more and keep in contact with us!
  - Main site: http://wicket.apache.org/
  - Tweeter account: https://twitter.com/apache_wicket/
  - User guide: http://wicket.apache.org/learn/#guide
  - User guide live examples: https://wicket-guide.herokuapp.com/

APACHE WICKET™

Thank you!