Apache Click

Apache Click User Guide

V 2.2.0

Copyright © 2009 The Apache Software Foundation

Table of Contents

Chapter 1. Introduction to Apache Click

Apache Click is a simple JEE web application framework for commercial Java developers.

Apache Click is an open source project, licensed under the *Apache license* .

Click uses an event based programming model for processing Servlet requests and Velocity for rendering the response. (Note other template engines such as JSP and Freemarker are also supported)

This framework uses a single servlet, called ClickServlet, to act as a request dispatcher. When a request arrives ClickServlet creates a Page object to process the request and then uses the page's Velocity template to render the results.

Pages provide a simple thread safe programming environment, with a new page instance created for each servlet request.

Possibly the best way to see how Click works is to dive right in and look at some examples. (The examples are also available online at http://www.avoka.com/click-examples/ under the menu "Intro Examples".)

1.1. Hello World Example

A Hello World example in Click would look something like this.

First we create a `HelloWorld` page class:

```java
package examples.page;

import java.util.Date;
import org.apache.click.Page;

public HelloWorld extends Page {

    private Date time = new Date(); ❶

    public HelloWorld() {
        addModel("time", time); ❷
    }

}
```

❶   Assign a new Date instance to the `time` variable.

❷   Add the `time` variable to the Page model under the name `"time"`. Click ensures all objects added to the Page model is automatically available in the Page template.

Next we have a page template hello-world.htm, where we can access the Page's `time` variable using the reference $time:

```html
<html>
  <body>

    <h2>Hello World</h2>
```

```
    Hello world from Click at $time

  </body>
</html>
```

Click is smart enough to figure out that the `HelloWorld` page class maps to the template hello-world.htm. We only have to inform Click of the package of the HelloWorld class, in this case examples.page. We do that through the click.xml [51] configuration file which allows Click to map hello-world.htm requests to the `examples.page.HelloWorld` page class.

```
<click-app>
  <pages package="examples.page"/>
</click-app>
```
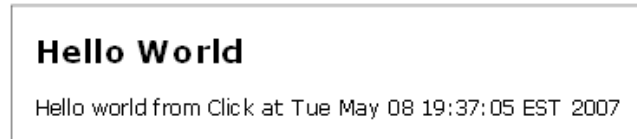
At runtime the following sequence of events occur. The ClickSerlvet maps a GET hello-world.htm request to our page class `example.page.HelloWorld` and creates a new instance. The HelloWorld page creates a new private *Date* object, which is added to the page's model under the name time.

The page model is then merged with the template which substitutes the $time reference with the *Date* object. Velocity then renders the merged template as:

Hello World

Hello world from Click at Tue May 08 19:37:05 EST 2007

Figure 1.1. Hello World Screenshot

1.2. Control Listener Type 1 Example

Click includes a library of Controls [34] which provide user interface functionality.

One of the commonly used controls is the ActionLink, which you can use to have an HTML link call a method on a Page object. For example:

```
public class ControlListenerType1Page extends Page {

    /* Set the listener to this object's "onLinkClick" method. */
    @Bindable protected ActionLink myLink = new ActionLink(this, "onLinkClick"); ❶

    @Bindable protected String msg;

    // -------------------------------------------------------- Event Handlers

    /**
     * Handle the ActionLink control click event.
     */
    public boolean onLinkClick() {
        msg = "ControlListenerPage#" + hashCode()
            + " object method <tt>onLinkClick()</tt> invoked.";

        return true;
    }
```

```
}
```

❶ Variables annotated with `@Bindable` is a short-hand way of instructing Click to automatically bind request parameters to Page variables. In addition these variables will automatically be added to the Page model. For more information please see the section Request Parameter Auto Binding [19].

In the Page class we create an ActionLink called myLink and define the control's listener to be the page method `onLinkClick()`. When a user clicks on myLink control it will invoke the listener method `onLinkClick()`.

In Click a control listener method can have any name but it must return a boolean value. The boolean return value specifies whether processing of page events should continue. This control listener pattern provides a short hand way for wiring up action listener methods without having to define anonymous inner classes.

The advantage of this style of control listener binding is that you have to write fewer lines of code. The disadvantage of this type of control listener binding is that no compile time safety is provided, and you miss out on the compiler refactoring capabilities provided with modern IDEs.

Back to our example, in the page template we define an HTML link and have the myLink control render the link's href attribute:

```html
<html>
  <head>
    <link type="text/css" rel="stylesheet" href="style.css"></link>
  </head>
  <body>

  Click myLink control <a href="$myLink.href">here</a>.

  #if ($msg)
    <div id="msgDiv"> $msg </div>
  #end

  </body>
</html>
```

At runtime this page would be rendered as:

Click myLink control here.

When a user clicks on the link the `onLinkClick()` method is invoked. This method then creates the msg model value, which is rendered in the page as:

Click myLink control here.

```
ControlListenerPage#12767107 object method onLinkClick() invoked.
```

1.3. Control Listener Type 2 Example

The second type of control listener binding uses the ActionListener interface to provide compile time safety. This compile time binding also supports code refactoring using modern IDE tools.

```java
public class ControlListenerType2Page extends Page {

    /* Public scope controls are automatically added to the page. */
    @Bindable protected ActionLink myLink = new ActionLink();

    @Bindable protected String msg;

    // ----------------------------------------------------------- Constructor

    /**
     * Create a new Page instance.
     */
    public ControlListenerType2Page() {
        myLink.setActionListener(new ActionListener() {
            public boolean onAction(Control control) {
                msg = "ControlListenerPage#" + hashCode()
                + " object method <tt>onAction()</tt> invoked.";

                return true;
            }
        });
    }

}
```

In the Page class we create an ActionLink called myLink. In the Page constructor we set the control's action listener to an anonymous inner class which implements the onAction(). When a user clicks on myLink control it will invoke the action listener method onAction().

As with our previous example, in the page template we define a HTML link and have the myLink control render the link's href attribute:

```html
<html>
  <head>
    <link type="text/css" rel="stylesheet" href="style.css"></link>
  </head>
  <body>

  Click myLink control <a href="$myLink.href">here</a>.

  #if ($msg)
    <div id="msgDiv"> $msg </div>
  #end

  </body>
</html>
```

At runtime this page would be rendered as:

Click myLink control here.

When a user clicks on the link the onAction() method is invoked. This method then creates the msg model value, which is rendered in the page as:

Click myLink control here.

```
ControlListenerPage#12767107 object method onAction() invoked.
```

1.4. Simple Table Example

One of the most useful Click controls is the Table control.

An example usage of the Table control in a customers Page is provided below:

```java
public class SimpleTablePage extends Page {

    @Bindable protected Table table = new Table();

    // -------------------------------------------------------- Constructor

    public SimpleTablePage() {
        table.setClass(Table.CLASS_ITS);

        table.addColumn(new Column("id"));
        table.addColumn(new Column("name"));
        table.addColumn(new Column("email"));
        table.addColumn(new Column("investments"));
    }

    // ---------------------------------------------------- Event Handlers


    /**
     * @see Page#onRender()
     */
    @Override
    public void onRender() {
        List list = getCustomerService().getCustomersSortedByName(10);
        table.setRowList(list);
    }
}
```

In this Page code example a Table control is declared, we set the table's HTML class, and then define a number of table Column objects. In the column definitions we specify the name of the column in the constructor, which is used for the table column header and also to specify the row object property to render.

The last thing we need to do is populate the table with data. To do this we override the Page onRender() method and set the table row list before it is rendered.

In our Page template we simply reference the $table object which is rendered when its toString() method is called.

```html
<html>
  <head>
    $headElements
  </head>
  <body>

    $table

    $jsElements
```

```
    </body>
</html>
```

Note from the example above that we specify the $headElements reference so that the table can include any HEAD elements, which includes Css imports and styles, in the header. Also note we specify the $jsElements reference which include any JavaScript imports and scripts at the bottom. At runtime Click automatically makes the variables $headElements and $jsElements available to the template.

At runtime the Table would be rendered in the page as:

Id	Name	Email	Investments
231	Albert Master	albert.master@gmail.com	Bonds
210	Alfred Alan	aalan@gmail.com	Stocks
256	Alison Smart	asmart@biztalk.com	Residential Property
211	Ally Emery	allye@easymail.com	Stocks
248	Andrew Phips	andyp@mycorp.com	Stocks
234	Andy Mitchel	andym@hotmail.com	Stocks
226	Angus Robins	arobins@robins.com	Bonds
241	Ann Melan	ann_melan@iinet.com	Residential Property
225	Ben Bessel	benb@hotmail.com	Stocks
235	Bensen Romanolf	benr@albert.net	Bonds

Figure 1.2. Simple Table

1.5. Advanced Table Example

The Table control also provides support for:

• automatic rendering

• column formatting and custom rendering

• automatic pagination

• link control support

A more advanced Table example is provided below:

```
public class CustomerPage extends Page {

    @Bindable protected Table table = new Table();
    @Bindable protected PageLink editLink = new PageLink("Edit", EditCustomer.class);
    @Bindable protected ActionLink deleteLink = new ActionLink("Delete", this, "onDeleteClick");

    // ----------------------------------- Constructor

    public CustomersPage() {
        // Setting Page to stateful to preserve Table sort and paging state while
        // editing customers
        setStateful(true); ❶

        table.setClass(Table.CLASS_ITS);
        table.setPageSize(10);
```

```
        table.setShowBanner(true);
        table.setSortable(true);

        table.addColumn(new Column("id"));

        table.addColumn(new Column("name"));

        Column column = new Column("email");
        column.setAutolink(true);
        column.setTitleProperty("name");
        table.addColumn(column);

        table.addColumn(new Column("investments"));

        editLink.setImageSrc("/images/table-edit.png");
        editLink.setTitle("Edit customer details");
        editLink.setParameter("referrer", "/introduction/advanced-table.htm");

        deleteLink.setImageSrc("/images/table-delete.png");
        deleteLink.setTitle("Delete customer record");
        deleteLink.setAttribute("onclick",
            "return window.confirm('Are you sure you want to delete this record?');");

        column = new Column("Action");
        column.setTextAlign("center");
        AbstractLink[] links = new AbstractLink[] { editLink, deleteLink };
        column.setDecorator(new LinkDecorator(table, links, "id"));
        column.setSortable(false);
        table.addColumn(column);

        // Table rowList will be populated through a DataProvider which loads
        // data on demand.
        table.setDataProvider(new DataProvider() {

            public List getData() {
                return getCustomerService().getCustomers();
            }
        });
    }

    // --------------------------------- Event Handlers

    /**
     * Handle the delete row click event.
     */
    public boolean onDeleteClick() {
        Integer id = deleteLink.getValueInteger();
        getCustomerService().deleteCustomer(id);
        return true;
    }
}
```

❶   Setting Page to stateful instructs Click to store the Page in the <u>HttpSession</u>. This ensures the Table's sort and paging state is preserved while editing customers. See the <u>Stateful Page [29]</u> section for more details.

In this Page code example a Table control is declared and a number of <u>Column</u> objects are added. We set the Table's <u>DataProvider</u> instance which provides data on demand to the table. Data retrieved from the dataProvider will be used

to populate the Table rowList before it is rendered. An editLink PageLink control is used as decorator for the "Action" column. This control navigates to the `EditCustomer` page. A deleteLink ActionLink control is also used as a decorator for the "Action" column. This control will invoke the Page `onDeleteClick()` method when it is clicked.

In our Page template we simply reference the $table object which is rendered when its `toString()` method is called.

```html
<html>
  <head>
    $headElements
  </head>
  <body>

    $table

    $jsElements

  </body>
</html>
```

At runtime the Table would be rendered in the page as:

Figure 1.3. Advanced Table

In this example, clicking on the Edit link will navigate the user to the `EditCustomer` page where the selected customer can be edited. When the user click on the Delete link, the `onDeleteClick()` method will be called on the Page deleting the customer record.

1.6. Simple Form Example

The Form and Field controls are also some of the most commonly used controls in the Click Framework.

The SimpleForm page below provides a demonstration of using these controls.

In our example code we have the page's constructor adding a TextField field and a Submit button to the form. A page method is also set as a control listener on the form. Also note in this example the page's public form field is automatically added to its list of controls.

```java
public class SimpleForm extends Page {

    @Bindable protected Form form = new Form();
    @Bindable protected String msg;
```

```
    // ---------------------------------------------------------- Constructor

    public SimpleForm() {
        form.add(new TextField("name", true));
        form.add(new Submit("OK"));

        form.setListener(this, "onSubmit");
    }

    // -------------------------------------------------- Event Handlers

    /**
     * Handle the form submit event.
     */
    public boolean onSubmit() {
        if (form.isValid()) {
            msg = "Your name is " + form.getFieldValue("name");
        }
        return true;
    }
}
```

Next we have the SimpleForm template simple-form.htm. The Click application automatically associates the simple-form.htm template with the SimpleForm class.

```
<html>
  <head>
    $headElements
  </head>
  <body>

    $form

    #if ($msg)
      <div id="msgDiv"> $msg </div>
    #end

    $jsElements

  </body>
</html>
```

When the SimpleForm page is first requested the $form object will automatically render itself as:
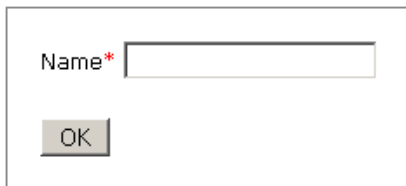
Figure 1.4. Simple Form

Say the user does not enter their name and presses the OK button to submit the form. The ClickServlet creates a new SimpleForm page and processes the form control.

The form control processes its fields and determines that it is invalid. The form then invokes the listener method `onSubmit()`. As the form is not valid this method simply returns true and the form renders the field validation errors.
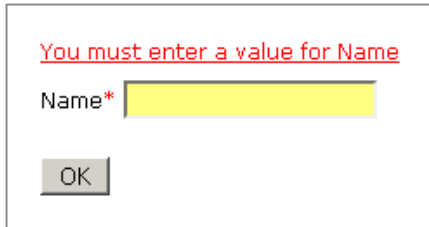
Figure 1.5. Form after an invalid request

Note the form will automatically maintain the entered state during the post and validate cycle.

Now if the user enters their name and clicks the OK button, the form will be valid and the `onSubmit()` add a msg to the Pages model. This will be rendered as:

Figure 1.6. Form after a valid request

1.7. Advanced Form Example

The `AdvancedForm` page below provides a more advanced demonstration of using Form, Field and FieldSet controls.

First we have an `AdvancedForm` class which setups up a Form in its constructor. The form's investment Select list is populated in the page's `onInit()` method. At this point any page dependencies such as the CustomerService should be available.

Note in this example the page's public form field is automatically added to its list of controls. The msg field is added to the page's model.

```java
public class AdvancedForm extends Page {

    @Bindable protected Form form = new Form();
    @Bindable protected String msg;

    private Select investmentSelect = new Select("investment");

    // -------------------------------------------------------- Constructor

    public AdvancedForm() {
        FieldSet fieldSet = new FieldSet("Customer");
        form.add(fieldSet);
```

```java
        TextField nameField = new TextField("name", true);
        nameField.setMinLength(5);
        nameField.setFocus(true);
        fieldSet.add(nameField);

        fieldSet.add(new EmailField("email", true));

        fieldSet.add(investmentSelect);

        fieldSet.add(new DateField("dateJoined", true));
        fieldSet.add(new Checkbox("active"));

        form.add(new Submit("ok", " OK ", this, "onOkClicked"));
        form.add(new Submit("cancel", this, "onCancelClicked"));
    }

    // ----------------------------------------------------- Event Handlers

    /**
     * @see Page#onInit()
     */
    @Override
    public void onInit() {
        CustomerService customerService = getCustomerService();
        investmentSelect.add(Option.EMPTY_OPTION);
        investmentSelect.addAll(customerService.getInvestmentCategories());
    }

    /**
     * Handle the OK button click event.
     *
     * @return true
     */
    public boolean onOkClicked() {
        if (form.isValid()) {
            Customer customer = new Customer();
            form.copyTo(customer);

            getCustomerService().saveCustomer(customer);

            form.clearValues();

            msg = "A new customer record has been created.";
        }
        return true;
    }

    /**
     * Handle the Cancel button click event.
     *
     * @return false
     */
    public boolean onCancelClicked() {
        setRedirect(HomePage.class);
        return false;
    }
}
```

Next we have the AdvancedForm template `advanced-form.htm`. The Click application automatically associates the `advanced-form.htm` template with the `AdvancedForm` class.

```html
<html>
  <head>
    $headElements
  </head>
  <body>

    #if ($msg)
      <div id="msgDiv"> $msg </div>
    #end

    $form

    $headElements

  </body>
</html>
```

When the AdvancedForm page is first requested the $form object will automatically render itself as:
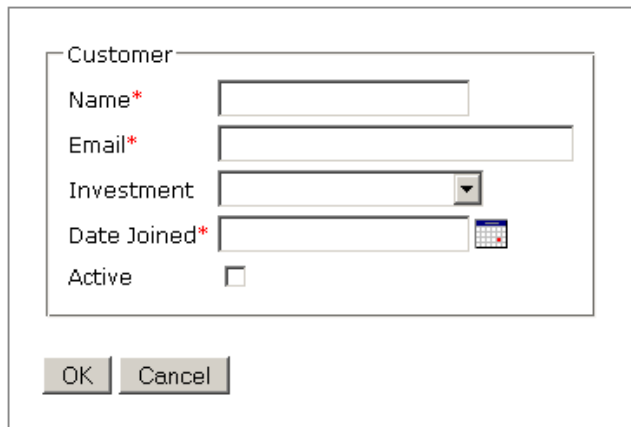
Figure 1.7. Advanced Form

In this example when the OK button is clicked the `onOkClicked()` method is invoked. If the form is valid a new customer object is created and the forms field values are copied to the new object using the Form copyTo() method. The customer object is then saved, the form's field values are cleared and an info message is presented to the user.

If the user clicks on the Cancel button the request is redirected to the applications HomePage.

1.7.1. Form Layout

In the example above the Form control automatically renders the form and the fields HTML markup. This is a great feature for quickly building screens, and the form control provides a number of layout options. See the Click Examples for an interactive Form Properties demo.

For fine grained page design you can specifically layout form and fields in your page template. See the Template Layout [44] section and Form Javadoc for more details.

An alternative to page template design is using a programmatic approach. See for more details.

Chapter 2. Pages

Pages are the heart of web applications. In Apache Click, Pages encapsulate the processing of HTML requests and the rendering of HTML responses. This chapter discusses Apache Click pages in detail.

In Click, a logical page is composed of a Java class and a Velocity template, with these components being defined in page elements of the click.xml [51] file:

```
<page path="search.htm" classname="com.mycorp.page.Search"/>
```

The path attribute specifies the location of the page Velocity template, and the classname attribute specifies the page Java class name. If you use the Freemarker template engine instead of Velocity, the setup is the same.

The template path should have an .htm extension which is specified in web.xml [50] to route *.htm requests to the `ClickServlet`.

Please note if you want Click to process templates with a different extension e.g. .xml, you need to implement the method isTemplate(String path) and specify the extra extensions. The simplest way is to subclass `XmlConfigService` and override the default implementation as described here. Also remember to map the new extensions in `web.xml`.

If you use JSP pages for rendering, the .jsp extension must be used. For example:

```
<page path="search.jsp" classname="com.mycorp.page.Search"/>
```

At runtime Click automatically converts the page path from .jsp to .htm and back, so there is no need to map the .jsp extension in `web.xml`.

2.1. Classes

All custom Click pages must subclass the Page base class. The Page class and its associated companion classes, Context and Control, are depicted in the figure below.
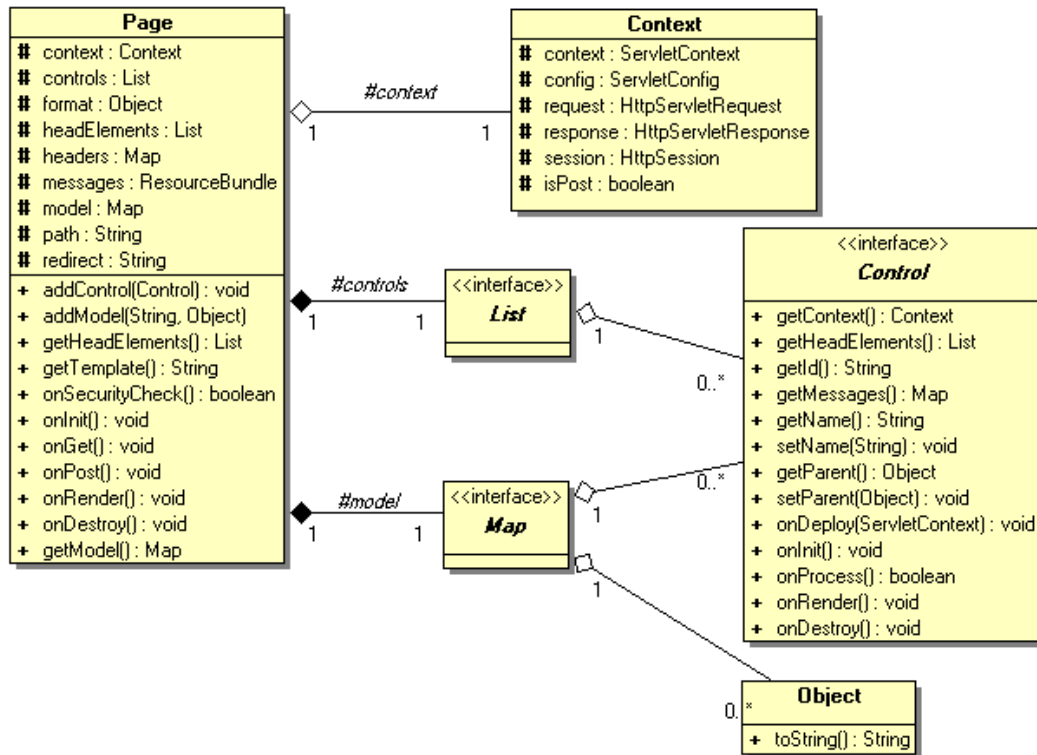
Figure 2.1. Page Class Diagram

The Page class provides a model attribute which is used to hold all the objects that are rendered in the page's Velocity template. The model may also contain Control objects, which provide user interface controls on the Page.

Pages also have an associated Context object which references all the javax.servlet objects associated with the request. When programming in Click you use the Context object to access HttpServletRequest attributes, parameters and the HttpSession object.

2.2. Execution

The Page class provide a number of empty handler methods which subclasses can override to provide functionality:

- onSecurityCheck()

- onInit()

- onGet()

- onPost()

- onRender()

- onDestroy()

The ClickServlet relies on instantiating Pages using a public no arguments constructor, so when you create Page subclasses you must ensure you don't add an incompatible constructor. The GET request execution sequence for Pages is summarized below in the Figure 2.
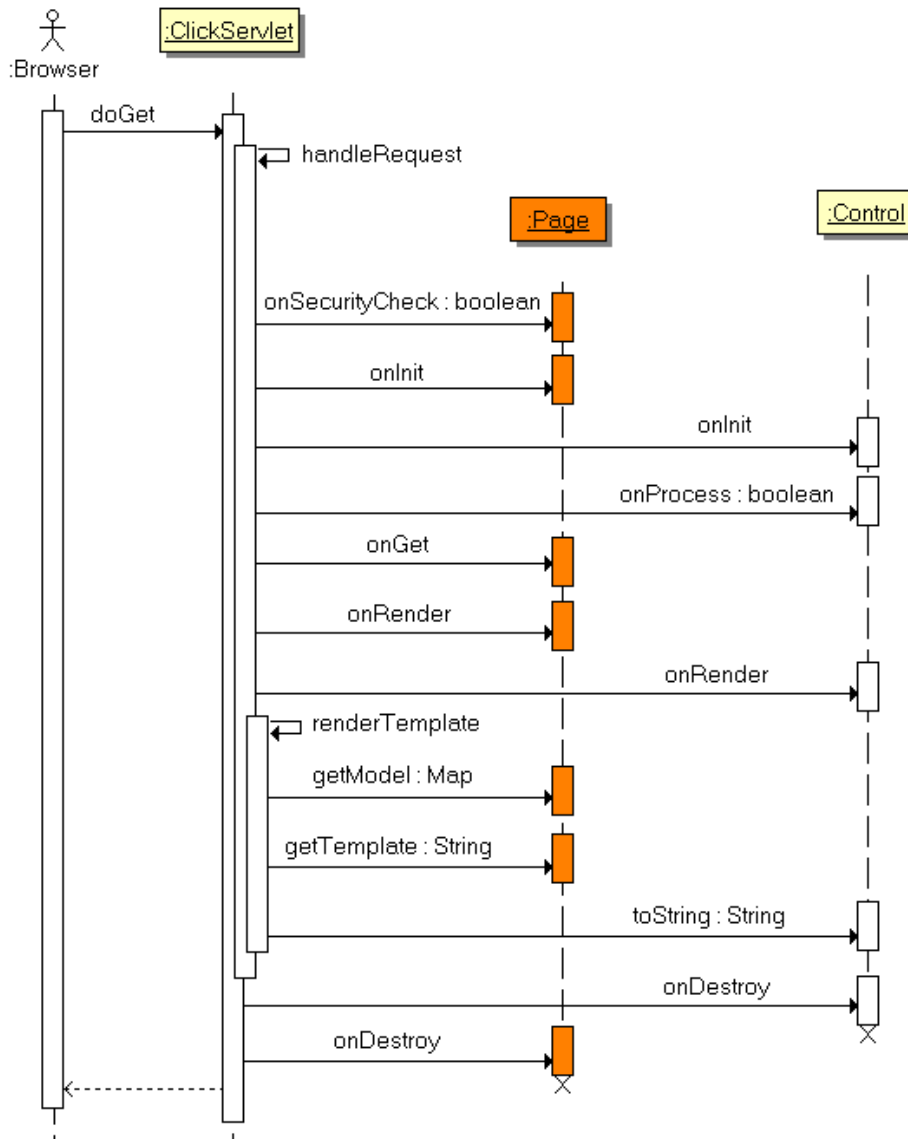
Figure 2.2. GET Request Sequence Diagram

Stepping through this GET request sequence, a new Page instance is created and the attributes for the Page are set (context, format, headers, path). Next, request parameter values are bound to any matching public Page fields.

Then the `onSecurityCheck()` handler is executed. This method can be used to ensure the user is authorized to access the page, and if necessary abort any further processing.

The next method invoked is `onInit()`, this is where you place any post constructor initialization code. `onInit()` is the ideal place to create controls such as Forms, Fields and Tables. As illustrated by the diagram, after a Page's `onInit()` is called, each Control, available at that stage, will have their `onInit()` method called.

The next step is the processing of the Page's controls. The ClickServlet gets the list of Controls from the page and then iterates through the list calling `onProcess()`. If any of the Control's `onProcess()` methods return false, processing of subsequent controls and the Page's `onGet()` method is aborted.

If everything is executing normally the Page's `onGet()` method is now called.

The next step is rendering the page template to generate the displayed HTML. The ClickServlet gets the model (`Map`) from the Page then adds the following objects to the model:

- any public Page fields using the fields name

- context - the Servlet context path, e.g. /mycorp

- format - the [Format](#) object for formatting the display of objects.

- headElements - the HEAD [elements](#), excluding JavaScript, to include in the page header. Please see [PageImports](#) for more details.

- jsElements - the JavaScript imports and script blocks to include in the pages footer. Please see [PageImports](#) for more details.

- messages - the [MessagesMap](#) adaptor for the Page [getMessage()](#) method

- path - the [path](#) of the page template to render

- request - the pages [HttpServletRequest](#) object

- response - the pages [HttpServletResponse](#) object

- session - the [SessionMap](#) adaptor for the users [HttpSession](#)

It then merges the template with the page model and writes out results to the HttpServletResponse. When the model is being merged with the template, any Controls in the model may be rendered using their `toString()` method.

The final step in this sequence is invoking each control's `onDestroy()` method and lastly invoke the Page's `onDestroy()` method. This method can be used to clean up resource associated with the Control or Page before it is garbage collected. The `onDestroy()` method is guaranteed to be called even if an exception occurs in the previous steps.

The execution sequence for POST requests is almost identical, except the `onPost()` method is invoked instead on `onGet()`. See the [POST Request Sequence Diagram](#).

Another view on the execution flow of Pages is illustrated in the Activity diagram below.
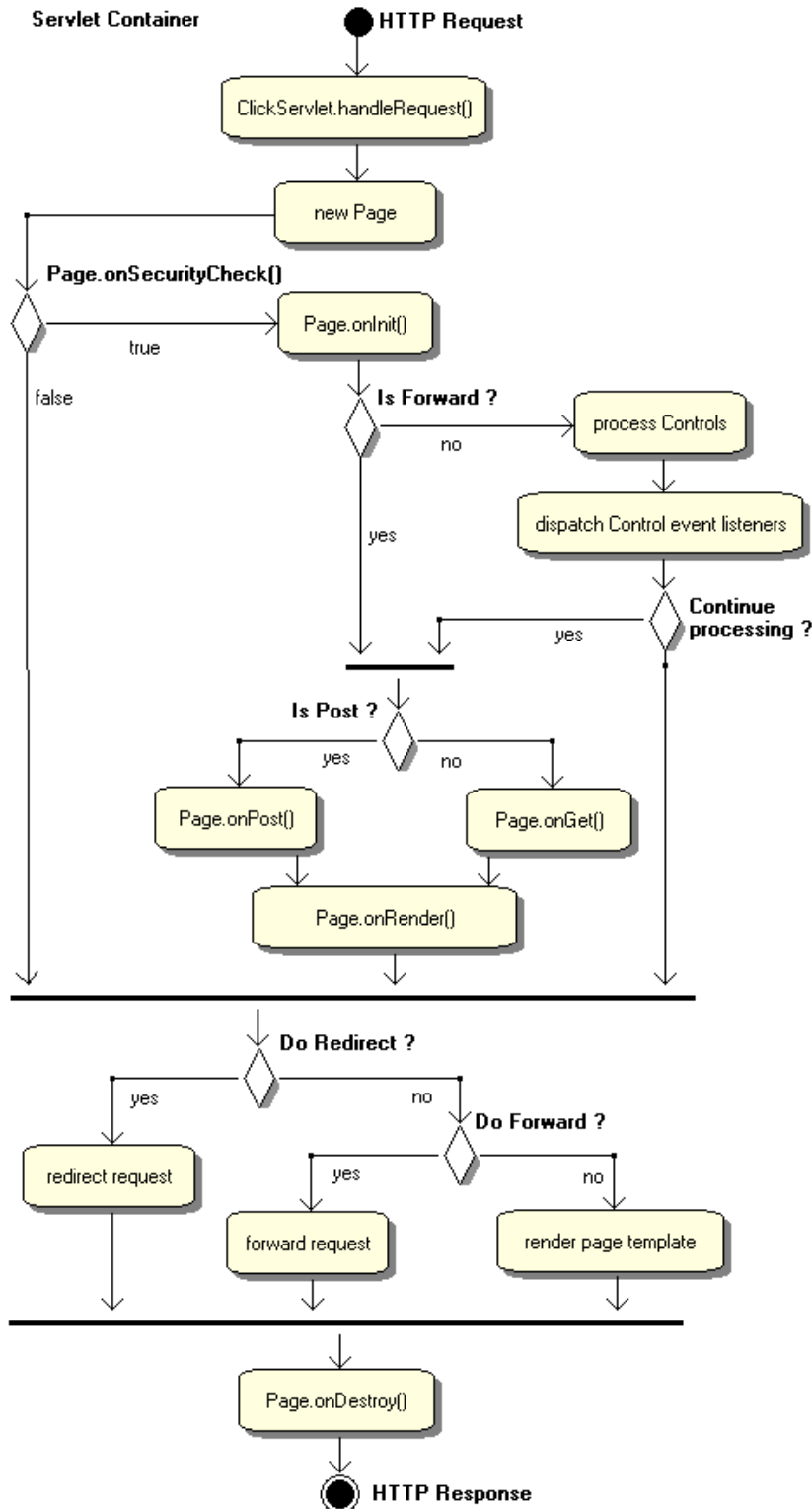
Figure 2.3. Page Execution Activity Diagram

2.3. Request Parameter Auto Binding

Click will automatically bind any request parameter values to public Page fields with the same name. You can also use the Bindable annotation to bind private and protected Page fields. When binding these values Click will also attempt to convert them to the correct type.

The best way to understand this is to walk through an example. Our application receives a GET request:

```
http://localhost:8080/mycorp/customer-details.htm?customerId=7203
```

This request is automatically handled by our `CustomerDetails` page:

```
package com.mycorp.page;

public class CustomerDetails extends Page {

    @Bindable protected Integer customerId;
}
```

After the CustomerDetails page has been created the "customerId" request parameter value "7023" will be converted into an Integer and assigned to the public page field customerId.

Another feature of Click is that any public Page fields are automatically added to the page's model before it is rendered. This will make these values available in the page template for display. In our example the public customerId field will be added to the Page model and will be available for rendering in the page template.

Our customer-details.htm page template contains:

```
<html>
<body>

  Customer ID: $customerId

</body>
</html>
```

After processing the request our page would be rendered as:

```
Customer ID: 7203
```

2.3.1. Customizing Auto Binding

Auto binding supports the conversion of request string parameters into the Java classes: Integer, Double, Boolean, Byte, Character, Short, Long, Float, BigInteger, BigDecimal, String and the various Date classes.

By default type conversion is performed by the RequestTypeConverter class which is used by the ClickServlet method getTypeConverter().

If you need to add support for additional types, you would write your own type converter class and specify it as a ClickServlet init parameter.

For example if you wanted to automatically load a `Customer` object from the database when a customer id request parameter is specified, you could write your own type converter:

```java
public class CustomTypeConverter extends RequestTypeConverter {

    private CustomerService customerService = new CustomerService();

    /**
     * @see RequestTypeConverter#convertValue(Object, Class)
     */
    protected Object convertValue(Object value, Class toType) {
        if (toType == Customer.class) {
            return customerService.getCustomerForId(value);

        } else {
            return super.convertValue(value, toType);
        }
    }
}
```

This type converter would handle the following request:

```
http://localhost:8080/mycorp/customer-details.htm?customer=7203
```

This request will load the customer object from the database using "7203" as the customer id value. The ClickServlet would then assign this customer object to the matching page field:

```java
package com.mycorp.page;

public class CustomerDetails extends Page {

    @Bindable protected Customer customer;

}
```

To make your custom type converter available you will need to add an init parameter to ClickServlet in `web.xml`. For example:

```xml
<web-app>
  ...
  <servlet>
    <servlet-name>ClickServlet</servlet-name>
    <servlet-class>org.apache.click.ClickServlet</servlet-class>
    <init-param>
        <param-name>type-converter-class</param-name>
        <param-value>com.mycorp.util.CustomTypeConverter</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>ClickServlet</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>
  ...
```

```
</web-app>
```

2.4. Security

Pages provide an [onSecurityCheck](#) event handler which application pages can override to implement a programmatic security model.

Please note you generally don't need to use this capability, and where possible you should use the declarative JEE security model. See the Best Practices <u>Security [65]</u> topic for more details.

2.4.1. Application Authentication

Applications can use the `onSecurityCheck()` method to implement their own security model. The example class below provides a base Secure page class which other pages can extend to ensure the user is logged in. In this example the login page creates a session when a user successfully authenticates. This Secure page then checks to make sure the user has a session, otherwise the request is redirected to the login page.

```java
public class Secure extends Page {

    /**
     * @see Page#onSecurityCheck()
     */
    public boolean onSecurityCheck() {

        if (getContext().hasSession()) {
            return true;

        } else {
            setRedirect(LoginPage.class);
            return false;
        }
    }
}
```

2.4.2. Container Authentication

Alternatively you can also use the security services provided by the JEE Servlet Container. For instance to ensure users have been authenticated by the Serlvet Container you could use a Secure page of:

```java
public class Secure extends Page {

    /**
     * @see Page#onSecurityCheck()
     */
    public boolean onSecurityCheck() {

        if (getContext().getRequest().getRemoteUser() != null) {
            return true;

        } else {
            setRedirect(LoginPage.class);
            return false;
```

```
        }
    }
}
```

2.4.3. Container Access Control

The Servlet Container also provides facilities to enforce role based access control (authorization). The example below is a base page to ensure only users in the "admin" role can access the page, otherwise users are redirected to the login page. Application Admin pages would extend this secure page to provide their functionality.

```java
public class AdminPage extends Page {

    /**
     * @see Page#onSecurityCheck()
     */
    public boolean onSecurityCheck() {

        if (getContext().getRequest().isUserInRole("admin")) {
            return true;

        } else {
            setRedirect(LoginPage.class);
            return false;
        }
    }
}
```

2.4.4. Logging Out

To logout using the application or container based security models you would simply invalidate the session.

```java
public class Logout extends Page {

    /**
     * @see Page#onInit()
     */
    public void onInit() {
        getContext().getSession().invalidate();
    }
}
```

2.5. Page Navigation

Navigation between pages is achieved by using forwards, redirects and by setting the page template path.

2.5.1. Forward

To forward to another page using the servlet RequestDispatcher, set the Page's forward property. For example to forward to a page with a path index.htm:

```
/**
```

```
 * @see Page#onPost()
 */
public void onPost() {
    // Process form post
    ..

    setForward("index.htm");
}
```

This will invoke a new Page class instance mapped to the path index.htm.

Please note when a request is forwarded to another Page, the controls on the second page will not be processed. This prevents confusion and bugs, like a form on the second page trying to process a POST request from the first page.

2.5.1.1. Forward Parameter Passing

When you forward to another page the request parameters are maintained. This is a handy way of passing through state information with the request. For example you could add a customer object as a request parameter which is displayed in the template of the forwarded page.

```
public boolean onViewClick() {
    Long id = viewLink.getValueLong();
    Customer customer = CustomerDAO.findByPK(id);

    // Set the customer object as a request parameter
    getContext().setRequestAttribute("customer", customer);
    setForward("view-customer.htm");

    return false;
}
```

The snippet above forwards to the page template view-customer.htm:

```
<html>
 <head>
   <title>Customer Details</title>
 </head>
 <body>
   <h1>Customer Details</h1>
   <pre>
     Full Name: $customer.fullName
     Email:     $customer.email
     Telephone: $customer.telephone
   </pre>
 </body>
</html>
```

Request attributes are automatically added to the Velocity Context object so are available in the page template.

2.5.1.2. Page Forwarding

Page forwarding is another way of passing information between pages. In this case you create the page to be forwarded to using the Context createPage(String) method and then set properties directly on the Page. Finally set this page as the page to forward the request to. For example:

```
public boolean onEditClick() {
    Long id = viewLink.getValueLong();
    Customer customer = CustomerDAO.findByPK(id);

    // Create a new EditPage instance based on the specified path
    EditPage editPage = (EditPage) getContext().createPage("/edit-customer.htm");
    editPage.setCustomer(customer);
    setForward(editPage);

    return false;
}
```

When creating a page with the `createPage()` method, ensure you prefix the page path with the "/" character.

You can also specify the target page using its class as long as the Page has a unique path. Using this technique the above code becomes:

```
public boolean onEditClick() {
    Long id = viewLink.getValueLong();
    Customer customer = CustomerDAO.findByPK(id);

    // Create a new EditPage instance based on its class
    EditPage editPage = (EditPage) getContext().createPage(EditPage.class);
    editPage.setCustomer(customer);
    setForward(editPage);

    return false;
}
```

This Page forwarding technique is best practice as it provides you with compile time safety and alleviates you from having to specify page paths in your code. Please always use the Context `createPage()` methods to allow Click to inject Page dependencies.

Although uncommon it is possible to map more than one path to the same class. In these cases invoking Context `createPage(Class)` will throw an exception, because Click will not be able to determine which path to use for the Page.

2.5.2. Template Path

An alternative method of forwarding to a new page is to simply set the current Page's path to the new page template to render. With this approach the page template being rendered must have everything it needs without having its associated Page object created. Our modified example would be:

```
public boolean onViewClick() {
    Long id = viewLink.getValueLong();
    Customer customer = CustomerDAO.findByPK(id);

    addModel("customer", customer);

    // Set the Page's path to a new value
    setPath("view-customer.htm");

    return false;
```

```
}
```

Note how the customer object is passed through to the template in the Page model. This approach of using the Page model is not available when you forward to another Page, as the first Page object is "destroyed" before the second Page object is created and any model values would be lost.

2.5.3. Redirect

Redirects are another very useful way to navigate between pages. See HttpServletResponse. sendRedirect (location) for details.

The great thing about redirects are that they provide a clean URL in the users browser which matches the page that they are viewing. This is important for when users want to bookmark a page. The downside of redirects are that they involve a communications round trip with the users browser which requests the new page. Not only does this take time, it also means that all the page and request information is lost.

An example of a redirect to a logout.htm page is provided below:

```
public boolean onLogoutClick() {
    setRedirect("/logout.htm");
    return false;
}
```

If the redirect location begins with a "/" character the redirect location will be prefixed with the web applications context path. For example if an application is deployed to the context "mycorp" calling setRedirect("/customer/details.htm") will redirect the request to: "/mycorp/customer/details.htm".

You can also obtain the redirect path via the target Page's class. For example:

```
public boolean onLogoutClick() {
    String path = getContext().getPagePath(Logout.class);
    setRedirect(path);
    return false;
}
```

Note when using this redirect method, the target Page class must have a unique path.

A short hand way of redirecting is to simply specify the target Page class in the redirect method. For example:

```
public boolean onLogoutClick() {
    setRedirect(Logout.class);
    return false;
}
```

2.5.3.1. Redirect Parameter Passing

You can pass information between redirected pages using URL request parameters. The ClickServlet will encode the URL for you using HttpServletResponse.encodeRedirectURL (url).

In the example below a user will click on an OK button to confirm a payment. The onOkClick() button handler processes the payment, gets the payment transaction id, and then redirects to the trans-complete.htm page with the transaction id encoded in the URL.

```
public class Payment extends Page {
    ..

    public boolean onOkClick() {
        if (form.isValid()) {
            // Process payment
            ..

            // Get transaction id
            Long transId = OrderDAO.purchase(order);

            setRedirect("trans-complete.htm?transId=" + transId);

            return false;
        }
        return true;
    }
}
```

The Page class for the trans-complete.htm page can then get the transaction id through the request parameter "transId":

```
public class TransComplete extends Page {
    /**
     * @see Page#onInit()
     */
    public void onInit() {
        String transId = getContext().getRequest().getParameter("transId");

        if (transId != null) {

            // Get order details
            Order order = OrderDAO.findOrderByPK(new Long(transId));
            if (order != null) {
                addModel("order", order);
            }
        }
    }
}
```

2.5.3.2. Post Redirect

The parameter passing example above is also an example of a Post Redirect. The Post Redirect technique is a very useful method of preventing users from submitting a form twice by hitting the refresh button.

2.6. Page Templating

Click supports page templating (a.k.a. *Tiles* in Struts) enabling you to create a standardized look and feel for your web application and greatly reducing the amount of HTML you need to maintain.

To implement templating define a border template base Page which content Pages should extend. The template base Page class overrides the Page getTemplate() method, returning the path of the border template to render. For example:

```
public class BorderedPage extends Page {
```

```
    /**
     * @see Page#getTemplate()
     */
    public String getTemplate() {
        return "/border.htm";
    }
}
```

The BorderedPage template border.htm:

```html
<html>
  <head>
    <title>$title</title>
    <link rel="stylesheet" type="text/css" href="style.css" title="Style"/>
  </head>
  <body>

    <h2 class="title">$title</h2>

    #parse($path)

  </body>
</html>
```

Other pages insert their content into this template using the Velocity #parse directive, passing it their contents pages path. The $path value is automatically added to the VelocityContext by the ClickServlet.

An example bordered Home page is provided below:

```
<page path="home.htm" classname="Home"/>
```

```java
public class Home extends BorderedPage {

    public String title = "Home";

}
```

The Home page's content home.htm:

```
<b>Welcome</b> to Home page your starting point for the application.
```

When a request is made for the Home page (home.htm) Velocity will merge the border.htm page and home.htm page together returning:

```html
<html>
  <head>
    <title>Home</title>
    <link rel="stylesheet" type="text/css" href="style.css" title="Style"/>
  </head>
  <body>

    <h2 class="title">Home</h2>
```

```
    <b>Welcome</b> to Home page your application starting point.

  </body>
</html>
```

Which may be rendered as:
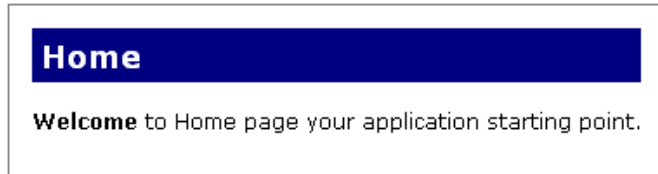
Figure 2.4. Home Page

Note how the Home page class defines a title model value which is referenced in the border.htm template as $title. Each bordered page can define their own title which is rendered in this template.

Templating with JSP pages is also supported using the same pattern. Please see the Click Examples application for a demonstration.

2.7. Direct Rendering

Pages support a direct rendering mode where you can render directly to the servlet response and bypass the page template rendering. This is useful for scenarios where you want to render non HTML content to the response, such as a PDF or Excel document. To do this:

- get the servlet response object

- set the content type on the response

- get the response output stream

- write to the output stream

- close the output stream

- set the page path to null to inform the ClickServlet that rendering has been completed

A direct rendering example is provided below.

```
/**
 * Render the Java source file as "text/plain".
 *
 * @see Page#onGet()
 */
public void onGet() {
    String filename = ..

    HttpServletResponse response = getContext().getResponse();
```

```
        response.setContentType("text/plain");
        response.setHeader("Pragma", "no-cache");

        ServletContext context = getContext().getServletContext();

        InputStream inputStream = null;
        try {
            inputStream = context.getResourceAsStream(filename);

            PrintWriter writer = response.getWriter();

            BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));

            String line = reader.readLine();

            while (line != null) {
                writer.println(line);
                line = reader.readLine();
            }

            setPath(null);

        } catch (IOException ioe) {
            ioe.printStackTrace();

        } finally {
            ClickUtils.close(inputStream);
        }
}
```

2.8. Stateful Pages

Click supports stateful pages where the state of the page is saved between the users requests. Stateful pages are useful in a number of scenarios including:

- Search page and edit page interactions. In this scenario you navigage from a stateful search page which may have filter criteria applied to an object edit page. Once object update has been completed on the edit page, the user is redirected to the search page and the stateful filter criteria still applies.

- Complex pages with multiple forms and or tables which need to maintain their state between interactions.

To make a page stateful you simply need to set the page stateful property to true, have the page implement the Serializable interface and set the serialVersionUID indicator. For example:

```
package com.mycorp.page;

import java.io.Serializable;

import org.apache.click.Page;

public class SearchPage extends Page implements Serializable {

    private static final long serialVersionUID = 1L;

    public SearchPage() {
```

```
        setStateful(true);
        ..
    }
}
```

Stateful page instances are stored in the user's [HttpSession](#) using the pages class name as the key. In the example above the page would be stored in the users session using the class name: `com.mycorp.page.SearchPage`

2.8.1. Page Creation

With stateful pages they are only created once, after which they are retrieved from the session. However page event handlers are invoked for each request, including the `onInit()` method.

When you are creating stateful pages you typically place all your control creation code in the Pages constructor so it is invoked only once. It is important not to place control creation code in the `onInit()` method which will be invoked with each request.

If you have dynamic control creation code you would typically place this in the `onInit()` method, but you will need to take care that controls and or models are not already present in the page.

2.8.2. Page Execution

The default Click page execution model is thread safe as a new Page instance is created for each request and thread. With stateful pages a user will have a single page instance which is reused in multiple requests and threads. To ensure page execution is thread safe, users page instances are synchronized so only one request thread can execute a page instance at any one time.

2.8.3. Page Destruction

After normal page instances have been executed, they are de-referenced and garbage collected by the JVM. However with stateful pages they are stored in the users `HttpSession` so care needs to be take not to store too many objects in stateful page instances which may cause memory and performance issues.

When pages have completed their execution, all the Page's controls `onDestroy()` methods are invoked, and then the Page's `onDestroy()` method is invoked. This is your opportunity to de-reference any large sets or graphs. For example the Table control by default de-references its rowList in its [onDestory()](#) method.

2.9. Error Handling

If an Exception occurs processing a Page object or rendering a template the error is delegated to the registered handler. The default Click error handler is the [ErrorPage](#), which is automatically configured as:

```
<page path="click/error.htm" classname="org.apache.click.util.ErrorPage"/>
```

To register an alternative error handler you must subclass ErrorPage and define your page using the path ["click/error.htm"](#). For example:

```
<page path="click/error.htm" classname="com.mycorp.page.ErrorPage"/>
```

When the ClickServlet starts up it checks to see whether the error.htm template exists in the click web sub directory. If it cannot find the page the ClickServlet will automatically deploy one. You can tailor the click/error.htm template to suite you own tastes, and the ClickServlet will not overwrite it.

The default error template will display extensive debug information when the application is in `development` or `debug` mode. Example error page displays include:

- **NullPointerException** - in a page method

- **ParseErrorException** - in a page template

When the application is in `production` mode only a simple error message is displayed. See <u>Configuration [58]</u> for details on how to set the application mode.

Please also see the <u>Examples</u> web app Exception Demo for demonstrations of Clicks error handling.

2.10. Page Not Found

If the ClickServlet cannot find a requested page in the `click.xml` config file it will use the registered <u>not-found.htm</u> page.

The Click `not found page` is automatically configured as:

```
<page path="click/not-found.htm" classname="org.apache.click.Page"/>
```

You can override the default configuration and specify your own class, but you cannot change the path.

When the ClickServlet starts up it checks to see whether the not-found.htm template exists in the click web sub directory. If it cannot find the page the ClickServlet will automatically deploy one.

You can tailor the click/not-found.htm template to suite you own needs. This page template has access to the usual Click objects.

2.11. Page Message Properties

The Page class provides a messages property which is a MessagesMap of localized messages for the page. These messages are made available in the VelocityContext when the page is rendered under the key `messages`. So for example if you had a page title message you would access it in your page template as:

```
<h1> $messages.title </h1>
```

This messages map is loaded from the page class property bundle. For example if you had a page class `com.mycorp.page.CustomerList` you could have an associated property file containing the pages localized messages:

```
/com/mycorp/page/CustomerList.properties
```

You can also defined a application global page messages properties file:

/click-page.properties

Messages defined in this file will be available to all pages throughout your application. Note messages defined in your page class properties file will override any messages defined in the application global page properties file.

Page messages can also be used to override Control messages, see the Controls <u>Message Properties [38]</u> topic for more details.

2.12. Page HEAD Elements

The Page class provides the method <u>getHeadElements()</u> for contributing HEAD <u>elements</u> such as <u>JsImport</u>, <u>JsScript</u>, <u>CssImport</u> and <u>CssStyle</u>.

Here is an example of adding HEAD elements to the `MyPage` class:

```java
public class MyPage extends Page {

    public MyPage() {
        // Add the JavaScript import "/mypage.js" to the page
        getHeadElements().add(new JsImport("/mypage.js"));

        // Add some inline JavaScript content to the page
        getHeadElements().add(new JsScript("alert('Welcome to MyPage');"));

        // Add the Css import "/mypage.css" to the page
        getHeadElements().add(new CssImport("/mypage.css"));

        // Add some inline Css content to the page
        getHeadElements().add(new CssStyle("body { font-family: Verdana; }"));
    }

    ...

}
```

In the example above we added the HEAD elements from the Page constructor, however you can add HEAD elements from anywhere in the Page including the event handlers `onInit`, `onGet`, `onPost`, `onRender` etc. Please see <u>getHeadElements()</u> for more details.

Below is the `/my-page.htm` template:

```html
<html>
  <head>
    $headElements
  </head>

  <body>

    ...

    $jsElements

  </body>
```

```
</html>
```

The two variables, $headElements and $jsElements, are automatically made available to the Page template. These variables references the JavaScript and Css HEAD elements specified in `MyPage`.

The following HTML will be rendered (assuming the application context is "/myapp"):

```html
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="/myapp/mypage.css"></link>
    <style rel="stylesheet" type="text/css">
      body { font-family: Verdana; }
    </style>
  </head>

  <body>

    ...

    <script type="text/javascript" src="/myapp/mypage.js"/>
    <script type="text/javascript">
      alert('Welcome to MyPage');
    </script>

  </body>
</html>
```

A live demo showing how to add HEAD elements to a Page can be seen here.

Chapter 3. Controls

Apache Click provides a rich set of Controls which support client side rendering and server side processing. Please see the [Javadoc](#), which provides extensive information and examples of the core Controls.

This chapter covers Control in detail including the Control life cycle, Control event callbacks and localization.

3.1. Control Interface

Controls provide the server side components that process user input, and render their display to the user. Controls are equivalent to Visual Basic Controls or Delphi Components.

Controls handle the processing of user input in the [onProcess](#) method and render their HTML display using the toString() method. The execution sequence for a Control being processed and rendered is illustrated in the figure below.
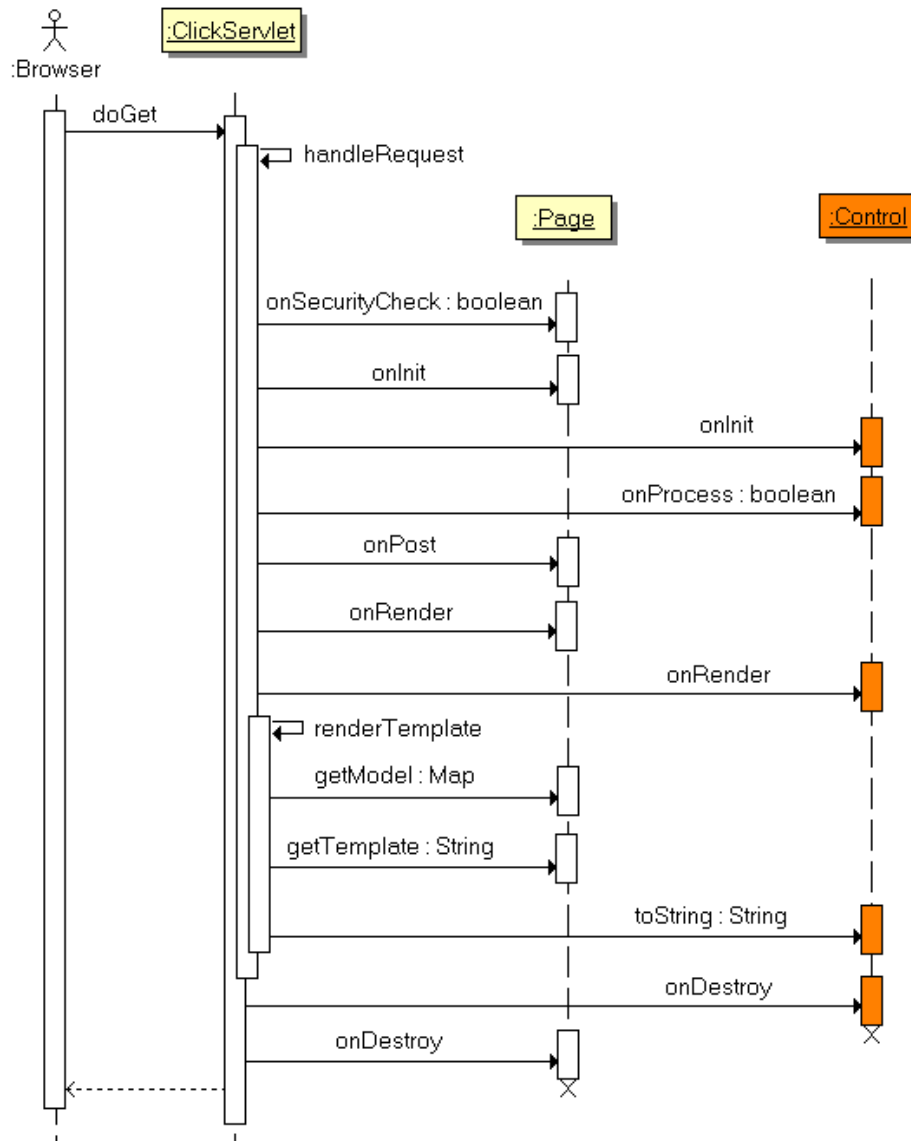
Figure 3.1. Post Sequence Diagram

In Click all control classes must implement the Control interface. The Control interface is depicted in the figure below.
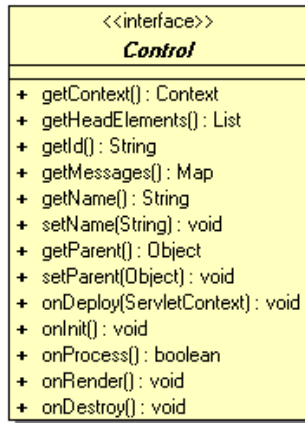
Figure 3.2. Control Interface Diagram

Methods on the Control interface include:

- getHeadElements() - defines the controls HTML header imports.

- getMessages() - defines the controls localized messages map.

- getName() / setName()   -   defines the controls name in the Page model or Form fields.

- getParent() / setParent()   -   defines the controls parent.

- onDeploy() - deploy resources on startup.

- onInit() - on initialize event handler.

- onProcess() - process request event handler.

- onDestroy() - on destroy event handler.

- render() - generate the control's HTML representation.

3.2. Control Callback

Click Controls provide an event callback mechanism similar to a `java.awt.ActionListener` callback.

Click supports two styles of action listeners. The first is using the ActionListener interface and setActionListener(ActionListener) method which provides compile time safety.

The second is to register the action listener via the setListener(Object, String) method where you specify the call back method via its name. This second style uses less lines of code, but has no compile time safety.

Examples of these two action listener styles are provided below:

```
public class ActionDemo extends BorderPage {
```

```
    // Uses listener style 1
    public ActionLink link = new ActionLink();

    // Uses listener style 2
    public ActionButton button = new ActionButton();

    public ActionDemo() {

        // Verbose but provides compile time safety
        link.setActionListener(new ActionListener() {
            public boolean onAction(Control source) {
                return onLinkClick(source);
            }
        });

        // Succinct but typos will cause runtime errors
        button.setListener(this, "onButtonClick");
    }

    // Event Handlers -------------------------------------------------------

    public boolean onLinkClick(Control source) {
        ..
        return true;
    }

    public boolean onButtonClick() {
        ..
        return true;
    }
}
```

All call back listener methods must return a boolean value. If they return true the further processing of other controls and page methods should continue. Otherwise if they return false, then any further processing should be aborted. By returning false you can effectively exit at this point and redirect or forward to another page. This execution logic is illustrated in the Page Execution Activity Diagram [18].

Being able to stop further processing and do something else can be very handy. For example your Pages onRender() method may perform an expensive database operation. By returning false in an event handler you can skip this step and render the template or forward to the next page.

3.3. Control Classes

Core control classes are defined in the package org.apache.click.control. This package includes controls for the essential HTML elements.

Extended control classes are provided in the Click Extras package org.apache.click.extras.control. Click Extras classes can contain dependencies to 3rd party frameworks.

A subset of these control classes are depicted in the figure below.
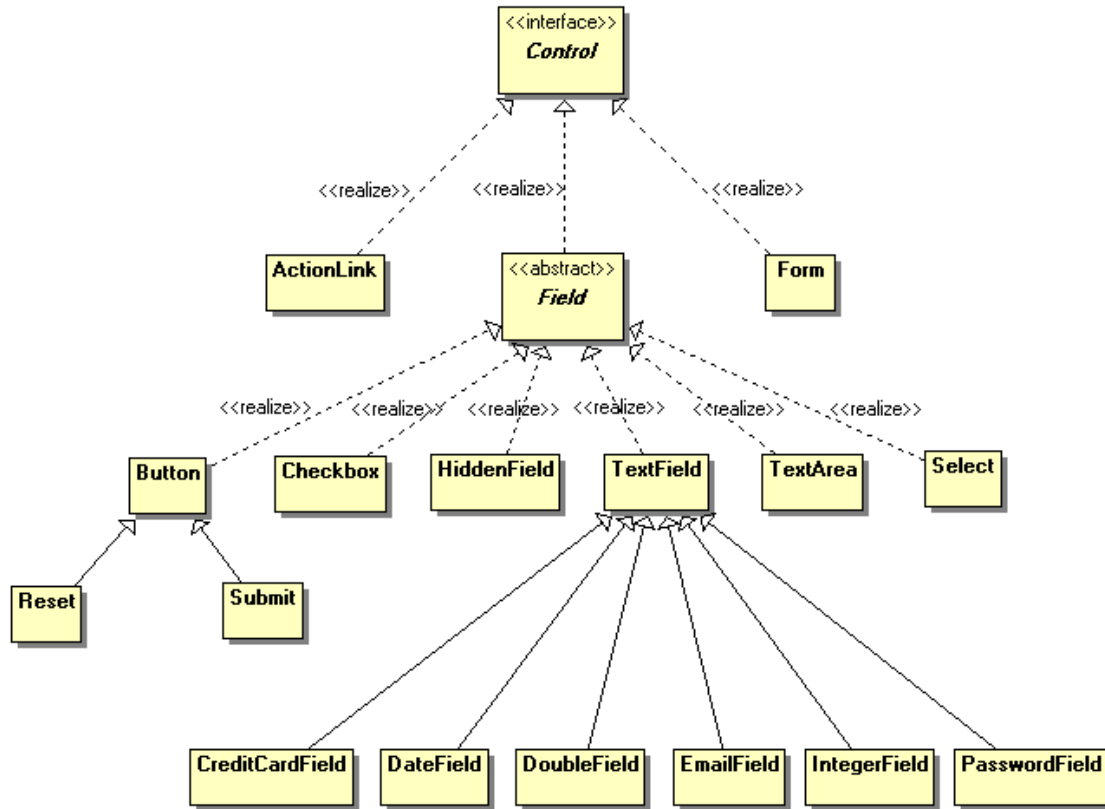
Figure 3.3. Package Class Diagram

The key control classes include:

- [ActionLink](#) - provides an anchor link which can invoke callback listeners.

- [Field](#) - provides the abstract form field control.

- [Form](#) - provides a form control for processing, validation and rendering.

- [Submit](#) - provides an input type submit control which can invoke callback listeners.

- [TextField](#) - provides an input type text control which can invoke callback listeners.

The control classes are designed to support subclassing for customized behaviour. All control fields have protected visibility and have public accessor methods.

You can also aggregate controls to build more complex controls. For example the [CreditCardField](#) uses a [Select](#) control to render the different credit card types.

3.4. Message Properties

Control strings for field validation messages and HTML formatting strings are externalized in the properties file. By using these properties files you can localize a Click application for your particular language and dialect.

3.4.1. Message Resolution

Messages are looked up in a particular order enabling taylor specific messages, for your controls, individual pages or across your entire application. The order in which localized messages are resolved is:

Page scope messages

Message lookups are first resolved to the Page classes message bundle if it exists. For example a `Login` page may define the message properties:

/com/mycorp/page/Login.properties

If you want to tailor messages for a particular page this is where to place them.

Global page scope messages

Next message lookups are resolved to the global pages message bundle if it exists.

/click-page.properties

If you want messages to be used across your entire application this is where to place them.

Control scope messages

Next message lookups are resolved to the Control classes message bundle if it exists. For example a `CustomTextField` control may define the message properties:

/com/mycorp/control/CustomTextField.properties

A custom control's messages can be placed here (or the global control scope covered next) and overridden by one of the above options.

Global control scope messages

Finally message lookups are resolved to the global application control message bundle if the message has not already been found. The global control properties file is:

/click-control.properties

Control messages can be placed here and overridden by one of the above options.

3.4.2. Control Properties

To customize the `click-control.properties` simply add this file to your classpath and tailor the specific values.

Note when customizing the message properties you must include all the properties, not just the ones you want to override.

```
# Click Control messages
field-maxlength-error={0} must be no longer than {1} characters
field-minlength-error={0} must be at least {1} characters
field-required-error=You must enter a value for {0}
```

```
file-required-error=You must enter a filename for {0}

label-required-prefix=
label-required-suffix=<span class="required">*</span>
label-not-required-prefix=
label-not-required-suffix= 

not-checked-error=You must select {0}

number-maxvalue-error={0} must not be larger than {1}
number-minvalue-error={0} must not be smaller than {1}

select-error=You must select a value for {0}

table-first-label=First
table-first-title=Go to first page
table-previous-label=Prev
table-previous-title=Go to previous page
table-next-label=Next
table-next-title=Go to next page
table-last-label=Last
table-last-title=Go to last page
table-goto-title=Go to page
table-page-banner=<span class="pagebanner">{0} items found, displaying {1} to {2}.</span>
table-page-banner-nolinks=
  <span class="pagebanner-nolinks">{0} items found, displaying {1} to {2}.</span>
table-page-links=<span class="pagelinks">[{0}/{1}] {2} [{3}/{4}]</span>
table-page-links-nobanner=<span class="pagelinks-nobanner">[{0}/{1}] {2} [{3}/{4}]</span>
table-no-rows-found=No records found.

table-inline-first-image=/click/paging-first.gif
table-inline-first-disabled-image=/click/paging-first-disabled.gif
table-inline-previous-image=/click/paging-prev.gif
table-inline-previous-disabled-image=/click/paging-prev-disabled.gif
table-inline-next-image=/click/paging-next.gif
table-inline-next-disabled-image=/click/paging-next-disabled.gif
table-inline-last-image=/click/paging-last.gif
table-inline-last-disabled-image=/click/paging-last-disabled.gif
table-inline-page-links=Page   {0} {1} {2} {3} {4}

# Message displayed when a error occurs when the application is in "production" mode
production-error-message=
  <div id='errorReport' class='errorReport'>The application encountered an unexpected error.
  </div>
```

3.4.3. Accessing Messages

Controls support a hierarchy of resource bundles for displaying validation error messages and display messages. These localized messages can be accessed through the AbstractControl methods:

- getMessage(String)

- getMessage(String, Object...)

- getMessages()

- [setErrorMessage(String)](#) - this method is defined on the Field class

- [setErrorMessage(String, Object)](#) - this method is defined on the Field class

These methods use the `Locale` of the request to lookup the string resource bundle, and use `MessageFormat` for any string formatting.

3.5. Control HEAD Elements

The Control interface provides the method [getHeadElements()](#) which allows the Control to add Page HEAD [elements](#) such as [JsImport](#), [JsScript](#), [CssImport](#) and [CssStyle](#).

Here is an example of adding HEAD elements to a custom `Control`:

```java
public class MyControl extends AbstractControl {

    public MyControl() {

        /**
         * Override the default getHeadElements implementation to return
         * MyControl's list of HEAD elements.
         *
         * Note that the variable headElements is defined in AbstractControl.
         *
         * @return list the list of HEAD elements
         */
        public List getHeadElements() {

            // Use lazy loading to only add the HEAD elements once and when needed.
            if (headElements == null) {

                // Get the head elements from the super implementation
                headElements = super.getHeadElements();

                // Add the JavaScript import "/mycontrol.js" to the control
                headElements.add(new JsImport("/mycontrol.js"));

                // Add the Css import "/mycontrol.css" to the control
                headElements.add(new CssImport("/mycontrol.css"));
            }
            return headElements;
        }
    }

    ...

}
```

In the example above we added the HEAD elements by overriding the Control's `getHeadElements` method, however you can add HEAD elements from anywhere in the Control including the event handlers `onInit`, `onGet`, `onPost`, `onRender` etc. Please see [getHeadElements()](#) for more details.

`MyControl` will add the following HEAD elements to the Page HEAD section, together with HEAD elements added by the Page and other controls (assume the application context is "/myapp"):

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="/myapp/mycontrol.css"></link>
  </head>

  <body>

    ...

    <script type="text/javascript" src="/myapp/mycontrol.js"/>

  </body>
</html>
```

A live demo showing how to add HEAD elements from a custom Control can be seen here.

3.6. Container

Container is a Control that can contain other Controls, thus forming a hierarchy of components. Container enables components to add, remove and retrieve other controls. Listed below are example Containers:

- Form - an HTML form which provides default layout of fields and error feedback.

- Panel - similar to Page, this Container provides its own template and model.

- FieldSet - draws a legend (border) around its child Controls.

These Containers are depicted in the figure below.
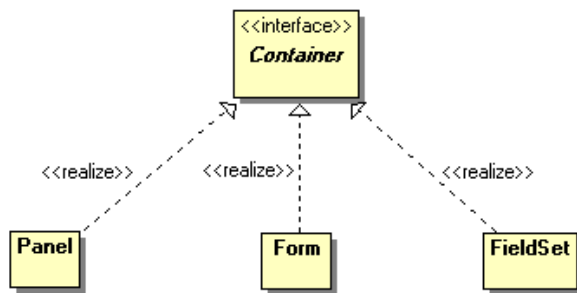
Figure 3.4. Container Class Diagram

The following classes provides convenient extension points for creating custom Containers:

- AbstractContainer

- AbstractContainerField

Lets cover each of them here.

3.6.1. AbstractContainer

Enables easy creation of custom Containers, for example an html *div* or *span* element:

```java
public class Div extends AbstractContainer {

    public Div(String name) {
        super(name);
    }

    public String getTag() {
        // Return the control's HTML tag.
        return "div";
    }
}
```

Lets try out the newly created `Container` above: (note the `MockContext` used in this test is described in the [Mock Test Support](#) documentation)

```java
public class Test {
    public static void main (String args[]) {
        // Create mock context in which to test the container.
        MockContext.initContext();

        // Create a div instance called "mydiv"
        String containerName = "mydiv";
        Div mydiv = new Div(containerName);

        // Add a control to the container
        mydiv.add(new TextField("myfield"));

        System.out.println(mydiv);
    }
}
```

Executing the above example results in the following output:

```html
<div name="mydiv" id="mydiv">
    <input type="text" name="myfield" id="myfield" value="" size="20" />
</div>
```

3.6.2. AbstractContainerField

AbstractContainerField extends Field and implements the Container interface. This provides a convenient base class in case you run into a situation where you need both a Field and Container.

Below is an example of how AbstractContainerField might be used:

```java
public class FieldAndContainer extends AbstractContainerField {

    public FieldAndContainer(String name) {
        super(name);
    }

    // Return the html tag to render
    public String getTag() {
        return "div";
    }
```

```
    }
```

To test the new class we use the following snippet:

```
public class Test {
    public static void main (String args[]) {
        // Create mock context in which to test the container.
        MockContext.initContext();

        // Create a FieldContainer instance called "field_container"
        String containerName = "field_container";
        FieldAndContainer fieldAndContainer = new FieldAndContainer(containerName);

        // Add a couple of fields to the container
        fieldAndContainer.add(new TextField("myfield"));
        fieldAndContainer.add(new TextArea("myarea"));

        System.out.println(fieldAndContainer);
    }
}
```

Executing the snippet produces the output:

```
<div name="field_container" id="field_container">
    <input type="text" name="myfield" id="myfield" value="" size="20"/>
    <textarea name="myarea" id="myarea" rows="3" cols="20"></textarea>
</div>
```

3.7. Layouts

Controls such as [Form](#) takes care of layout and error reporting automatically, and for many use cases the auto-layout approach is good enough. It is certainly very productive.

However for custom or complex layouts, auto-layout is not always the best choice. There are two approaches for creating custom layouts.

- Template approach - use a template engine such as Velocity, Freemarker or JSP to declare the layout as HTML markup.

- Programmatic approach - build custom layout components using Java. This option is very similar to building components using Swing.

3.7.1. Template layout

The [Template](#) approach separates the Page and layout logic. The Page is used to implement the presentation logic such as creating controls, registering listeners and copying data to domain objects, while the template is used to layout the Page controls.

Lets walk through an example using the template approach. Below we create an EmployeePage which contains a Form and a bunch of fields and submit button.

```java
// EmployeePage.java
public EmployeePage extends Page {

    private Form form;

    public void onInit() {
        // Create form
        Form form = new Form("form");

        // Add a couple of fields to the form
        form.add(new TextField("firstname"));
        form.add(new TextField("lastname"));
        form.add(new IntegerField("age"));
        form.add(new DoubleField("salary"));

        // Add a submit button to form
        form.add(new Submit("submit", "Add Employee"));

        // Add form the page
        addControl(form);
    }
}
```

Lets imagine we want to create a layout using the HTML tags, <div> and <ol>.

We would then provide the markup for the employee.htm template as shown below, using a template engine such as Velocity:

```html
<!-- employee.htm -->
${form.startTag()}
    <div style="margin: 1em;">
        <ol>
            <li>
                <label for="firstname">Firstname:</label>
                ${form.fields.firstname}
            </li>
            <li>
                <label for="lastname">Lastname:</label>
                ${form.fields.lastname}
            </li>
            <li>
                <label for="age">Age:</label>
                ${form.fields.age}
            </li>
            <li>
                <label for="salary">Salary:</label>
                ${form.fields.salary}
            </li>
        </ol>
    </div>
    ${form.fields.submit}
${form.endTag()}
```

Using CSS the markup above can further be styled and transformed into a nice looking form.

There are pros and cons to using the template approach.

One of the advantages is that the layout is explicit and one can easily tweak it if needed. For example instead of using divs and ordered lists, one can change the template to leverage a table layout.

A disadvantage is added redundancy. In the example above we created the fields in Java, and laid them out using markup in the template. If the requirements should change to add a new field for example, one will have to add the field in the Page as well as the template.

However it is possible to "generify" the layout using template engines such as Velocity, Freemarker and JSP. Macro.vm is an example of a generic form layout using Velocity.

3.7.2. Programmatic layout

To combat the redundancy introduced by the Template approach, you can take a programmatic approach and use normal Java and some Click classes to build custom layouts.

Click extras provides two useful classes in this situation namely, HtmlForm and HtmlFieldSet.

Unlike Form and FieldSet which renders its controls using a Table layout, HtmlForm and HtmlFieldSet renders its controls in the order they were added and does not add any extra markup. HtmlForm will be used in the examples below.

To make it easy to compare the two layout approaches we will recreate the example from the template layout section, but using the programmatic approach.

When creating custom layouts, the HTML construct List <ul> is pretty useful. Since Click does not provide this component, we will create it as shown below. First we create the HTML list element <ol>, to which list item elements <li> can be added:

```java
// HtmlList.java
public class HtmlList extends AbstractContainer {

    public String getTag() {
        return "ol";
    }

    // Can only add ListItems: <li> tags
    public Control add(Control control) {
        if (!(control instanceof ListItem)) {
            throw new IllegalArgumentException("Only list items can be added.");
        }
        return super.add(control);
    }
}
```

Next we create the HTML list item element <li>:

```java
// ListItem.java
public class ListItem extends AbstractContainer {

    public String getTag() {
        return "li";
    }
}
```

Another component that will be used in the example is a FieldLabel which renders an HTML label element for a target Field.

```java
// FieldLabel.java
public class FieldLabel extends AbstractControl {

    private Field target;

    private String label;

    public FieldLabel(Field target, String label) {
        this.target = target;
        this.label = label;
    }

    public String getTag() {
        return "label";
    }

    // Override render to produce an html label for the specified field.
    public void render(HtmlStringBuffer buffer) {
        // Open tag: <label
        buffer.elementStart(getTag());

        // Set attribute to target field's id
        setAttribute("for", target.getId());

        // Render the labels attributes
        appendAttributes(buffer);

        // Close tag: <label for="firstname">
        buffer.closeTag();

        // Add label text: <label for="firstname">Firstname:
        buffer.append(label);

        // Close tag: <label for="firstname">Firstname:</label>
        buffer.elementEnd(getTag());
    }

}
```

Now the form can be assembled. Continuing with the employee example from the template approach [44], we again create an EmployeePage, but this time an HtmlForm and HtmlList is used to create the custom layout:

```java
// EmployeePage.java
public class EmployeePage extends Page {
    // A form instance variable
    private HtmlForm form;

    // Build the form when the page is initialized
    public void onInit() {
        // Create an HtmlForm which is ideal for composing manual layouts
        form = new HtmlForm("form");

        // Create a list and add it to the form.
```

```
        HtmlList list = new HtmlList();
        form.add(list);

        // Add firstname field and pass in its name, label and the list to add the field to
        addTextField("firstname", "Firstname:", list);
        addTextField("lastname", "Lastname:", list);
        addTextField("age", "Age:", list);
        addTextField("salary", "Salary:", list);

        // Add a submit button to form
        form.add(new Submit("submit", "Add Employee"));

        // Add the form to the page
        addControl(form);
    }

    // Provide a helper method to add fields to the form
    private void addTextField(String nameStr, String labelStr, List list) {
        // Create a new ListItem <li> and add it to the List
        ListItem item = new ListItem();
        list.add(item);

        // Create a textfield with the specified name
        Field field = new TextField(nameStr);

        // Create a field label, which associates the label with the field id.
        // label.toString would output: <label for="firstname">Firstname:</name>
        FieldLabel label = new FieldLabel(field, labelStr);

        // Next add the label and field to the list item.
        // item.toString would then produce:
        // <li>
        //    <label for="firstname">Firstname:</name>
        //    <input type="text" name="firstname" id="form_firstname" value="" size="20"/>
        // </li>
        //
        item.add(label);
        item.add(field);
    }
}
```

And lastly the `employee.htm` template would only need to specify the name of the top level component, in this case form.

```
<!--employee.htm-->
${form}
```

which produces the following markup:

```
<form method="post" id="form" action="/myapp/employee.htm">
<input type="hidden" name="form_name" id="form_form_name" value="form"/>
    <ol>
        <li>
            <label for="firstname">Firstname:</label>
            <input type="text" name="firstname" id="form_firstname" value="" size="20"/>
        </li>
```

```
        <li>
            <label for="lastname">Lastname:</label>
            <input type="text" name="lastname" id="form_lastname" value="" size="20"/>
        </li>
        <li>
            <label for="age">Age:</label>
            <input type="text" name="age" id="form_age" value="" size="20"/>
        </li>
        <li>
            <label for="salary">Salary:</label>
            <input type="text" name="salary" id="form_salary" value="" size="20"/>
        </li>
    </ol>
    <input type="submit" name="submit" id="form_submit" value="Add Employee"/>
</form>
```

Again using a CSS stylesheet, the markup above can be styled and transformed into a fancy looking form.

There is a live demo showing the programmatic approach.

The advantage of the programmatic approach is that there is no redundancy. Each Field is created and added using normal Java. There is no need to specify where the Field must reside in the markup.

If new requirements arrive and more fields added, only the Page needs to change. There is no need to change the template as the layout is taken care of by CSS and the markup produced by the components.

Disadvantages are that more upfront work is needed to write the components and it is more difficult to *visualize* what output would be rendered by the components.

However once your custom layout components are in place, it can be reused across your project and boost productivity.

Whether you use the template [44] or programmatic [46] layout approach, is up to you. Both work well and have advantages and disadvantages over the other.

Chapter 4. Configuration

This section discusses how to setup and configure an Apache Click web application.

The Click configuration files include:

- WEB-INF/ <u>click.xml [51]</u>   -   Application Configuration (**required**)

- WEB-INF/ <u>web.xml [50]</u>   -   Servlet Configuration (**required**)

4.1. Servlet Configuration

For a Click web application to function the [ClickServlet](#) must be configured in the web application's `/WEB-INF/` `web.xml` file. A basic web application which maps all `*.htm` requests to a ClickServlet is provided below.

```
<web-app>

  <servlet>
    <servlet-name>ClickServlet</servlet-name>
    <servlet-class>org.apache.click.ClickServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>ClickServlet</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>

</web-app>
```

4.1.1. Servlet Mapping

By convention all Click page templates should have an .htm extension, and the ClickServlet should be mapped to process all *.htm URL requests. With this convention you have all the static HTML pages use an .html extension and they will not be processed as Click pages.

4.1.2. Load On Startup

Note you should always set `load-on-startup` element to be 0 so the servlet is initialized when the server is started. This will prevent any delay for the first client which uses the application.

The `ClickServlet` performs as much work as possible at startup to improve performance later on. The Click start up and caching strategy is configured with the Click application mode element in the `"click.xml"` config file, covered next.

4.1.3. Type Converter Class

The ClickServlet uses the OGNL library for type coercion when binding request parameters to bindable fields. The default type converter class used is [RequestTypeConverter](). To specify your own type converter configure a `type-converter-class` init parameter with the ClickServlet. For example:

```xml
<servlet>
  <servlet-name>ClickServlet</servlet-name>
  <servlet-class>org.apache.click.ClickServlet</servlet-class>
  <load-on-startup>0</load-on-startup>
  <init-param>
   <param-name>type-converter-class</param-name>
   <param-value>com.mycorp.util.CustomTypeConverter</param-value>
  </init-param>
</servlet>
```

4.1.4. Config Service Class

Click uses a single application configuration service which is instantiated by the ClickServlet at startup. This service defines the application configuration and is used by the ClickServlet to map requests to pages amongst other things.

Once the ConfigService has been initialized it is stored in the ServletContext using the key [ConfigService]() The default ConfigService is [XmlConfigService](), which configuration is discussed in detail in the next section.

To use an alternative configuration service specify a `config-service-class` context parameter. For example:

```xml
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
   version="2.4">

 ...

    <context-param>
        <param-name>config-service-class</param-name>
         <param-value>com.mycorp.service.CustomConfigSerivce</param-value>
     </context-param>

  ...

  </web-app>
```

4.2. Application Configuration

The heart of a Click application is the `click.xml` configuration file. This file specifies the application pages, headers, the format object and the applications mode.

By default the ClickServlet will attempt to load the application configuration file using the path: `/WEB-INF/click.xml`

If this file is not found under the `WEB-INF` directory, then ClickServlet will attempt to load it from the classpath as `/click.xml`.

See Click DTD for the click-app XML definition.

A complete Click configuration example is available here which can be used as a quick reference when configuring Click.

A basic Click app config file is provided below:

```xml
<click-app>

  <!-- Specify the Java package where Page classes can be found -->
  <pages package="com.mycorp.page"/>

  <mode value="profile"/>

</click-app>
```

An advanced config file would look like this:

```xml
<click-app charset="UTF-8" locale="de">

  <!-- Specify the Java package where Page classes can be found -->
  <pages package="com.mycorp.banking.page">
    <page path="index.htm" classname="Home"/>
  </pages>

  <!-- Specify a second Java package where Page classes can be found -->
  <pages package="com.mycorp.common.page"/>

  <format classname="com.mycorp.util.Format"/>

  <mode value="profile"/>

  <log-service classname="org.apache.click.extras.service.Log4JLogService"/>

</click-app>
```

The take away point is that there is not much to configure, even for advanced uses.

4.2.1. Click App

The root click-app element defines two application localization attributes charset and locale.

```
<!ELEMENT click-app (pages*, headers?, format?, mode?, controls?,
            file-upload-service?, log-service?, template-service?)>
 <!ATTLIST click-app charset CDATA #IMPLIED>
 <!ATTLIST click-app locale CDATA #IMPLIED>
```

The charset attribute defines the character encoding set for:

• Velocity templates

• HttpServletRequest character encoding

- Page Content-Type charset, see Page getContentType()

The locale attribute defines the default application Locale. If this value is defined it will override Locale returned by the request. Please see the Context getLocale() for details. For example the following configuration sets the application character set to UTF-8 and the default Locale as German (de):

```
<click-app charset=" UTF-8" locale="de">
  ..
</click-app>
```

4.2.2. Pages

The first child element of the click-app is the mandatory `pages` element which defines the list of Click pages.

```
<!ELEMENT pages (page*)>
  <!ATTLIST pages package CDATA #IMPLIED>
  <!ATTLIST pages automapping (true|false) "true">
  <!ATTLIST pages autobinding (default|annotation|none) "default">
```

The pages element can specify a default package name which is prepended to the classname of any pages defined.

The pages element also defines the automapping and automapping attributes which is discussed in the Page Automapping [54] and Page Autobinding [55] sections respectively.

4.2.2.1. Multiple Pages Packages

Click can support multiple pages elements to enable the automapping of multiple packages.

```
<click-app>

  <pages package="com.mycorp.banking.page"/>

  <pages package="com.mycorp.common.page"/>

</click-app>
```

With multiple pages elements, pages are loaded in the order of the page elements, with manual page elements being loaded before automapped pages. Once a page template has been mapped to a Page class it will not be replaced by a subsequent potential match. So pages elements at the top take priority over lower pages elements.

4.2.3. Page

The page element defines the Click application pages.

```
<!ELEMENT page(header*)>
  <!ATTLIST page path CDATA #REQUIRED>
  <!ATTLIST page classname CDATA #REQUIRED>
```

Each page path must be unique, as the Click application maps HTTP requests to the page paths.

The Click application will create a new Page instance for the given request using the configured page classname. All pages must subclass Page and provide a public no arguments constructor, so they can be instantiated.

Pages can also define header values which are discussed in the next topic.

When the Click application starts up it will check all the page definitions. If there is a critical configuration error the ClickServlet will log an ERROR message and throw an UnavailableException. If this occurs the click application will be permanently unavailable until the error is fixed and the web app is restarted.

4.2.3.1. Page Automapping

Page automapping will automatically configure application pages using a simple set of rules. This enables you to greatly streamline your configuration file as you only need to define pages which don't fit the automapping rules.

Automapping will attempt to associate each page template (*.htm) and JSP file in the web application (excluding those under the WEB-INF and click directories) to a Page class. Automapped pages are loaded after the manually defined pages are loaded, and manually defined pages takes preference. When automapping is enabled the Click application will log the page mappings when in debug or trace mode.

For example, given the following page path to class mapping:

```
index.htm                   => com.mycorp.page.Home
search.htm                  => com.mycorp.page.Search
contacts/contacts.htm        => com.mycorp.page.contacts.Contacts
security/login.htm           => com.mycorp.page.security.Login
security/logout.htm          => com.mycorp.page.security.Logout
security/change-password.htm => com.mycorp.page.security.ChangePassword
```

The above mapping could be configured manually by setting the automapping attribute to "false" and using the package prefix, for example:

```
<click-app>
  <pages package="com.mycorp.page" automapping="false">
    <page path="index.htm"                  classname="Home"/>
    <page path="search.htm"                 classname="Search"/>
    <page path="contacts/contacts.htm"       classname="contacts.Contacts"/>
    <page path="security/login.htm"          classname="security.Login"/>
    <page path="security/logout.htm"         classname="security.Logout"/>
    <page path="security/change-password.htm" classname="security.ChangePassword"/>
  </pages>
</click-app>
```

By using automapping the page paths will automatically map to page classes: (except for Home page which doesn't automatically map to index.html)

```
<click-app>
  <pages package="com.mycorp.page" automapping="true">
    <page path="index.htm" classname="Home"/>
  </pages>
</click-app>
```

Note automapping is true by default, so it could be omitted from the configuration file.

The page template name to classname convention is:

```
change-password.htm => ChangePassword
```

```
change_password.htm  =>  ChangePassword
changePassword.htm  =>  ChangePassword
ChangePassword.htm  =>  ChangePassword
```

When automapping pages, if a class cannot be found Click will attempt to add the 'Page' suffix to the classname if not already present and map this. For example:

```
customer.htm        =>  CustomerPage
change-password.htm  =>  ChangePasswordPage
```

4.2.3.2. Automapping Excludes

With Page automapping there can be resources where you don't want automapping applied. For example when using a JavaScript library with lots of `.htm` files, you don't want automapping to try and find Page class for each of these files. In these situations you can use the pages excludes element.

```
<!ELEMENT excludes (#PCDATA)>
  <!ATTLIST excludes pattern CDATA #REQUIRED>
```

For example if our application uses the TinyMCE JavaScript library we could configure our pages automapping to exclude all `.htm` files under the `/tiny_mce` directory.

```xml
<click-app>
  <pages package="com.mycorp.page">
    <excludes pattern="/tiny_mce/*"/>
  </pages>
</click-app>
```

The excludes pattern can specify multiple directories or files using a comma separated notation. For example:

```xml
<click-app>
  <pages package="com.mycorp.page">
    <excludes pattern="/dhtml/*, /tiny_mce/*, banner.htm, about.htm"/>
  </pages>
</click-app>
```

HTM files excluded from Page automapping are handled by an internal Page class with caching headers enabled.

4.2.3.3. Page Autobinding

Autobinding is a feature that allows certain page fields to be handled in a special way by the ClickServlet. The autobinding attribute can be configured with one of the following values:

- default: `bindable` fields include both public page fields and fields annotated with the @Bindable annotation

- annotation: `bindable` fields are fields annotated with the @Bindable annotation

- none: disables the autobinding feature

By default all pages have autobinding enabled in default mode. With autobinding the ClickServlet will automatically:

- add any `bindable` controls to the page, after the page constructor has been invoked

- if the `bindable` control name is not defined, its name will be set to the value of its field name

- bind any request parameters to `bindable` page fields, after page constructor has been invoked. See ClickServlet.processPageRequestParams(Page) for more details

- add all `bindable` page fields to the page model (this step occurs right before the page is rendered)

For example:

```java
public class EmployeePage extends Page {

    public Form employeeForm = new Form();

    public Table myTable = new Table();

}
```

Note in the above example that the employeeForm and myTable controls were not added to the page. Also note that Form and Table do not have their names defined.

When autobinding is enabled, ClickServlet will create a new Page and add the public controls to the page. Following the example above the employeeForm and myTable will be added to the page, similar to calling `addControl(employeeForm)` and `addControl(myTable)`.

Furthermore, the control names were not defined so ClickServlet will set their names to the value of their field/variable name. In this case the Form name will be set to employeeForm while the Table name will set to myTable.

The above example is thus a shorthand way of writing the following:

```java
public class EmployeePage extends Page {

    private Form employeeForm = new Form();

    private Table myTable = new Table();

    public void onInit() {
        employeeForm.setName("employeeForm");
        addControl(employeeForm);

        myTable.setName("myTable");
        addControl(myTable);
    }
}
```

You can turn this behaviour off by setting the autobinding attribute to none for example:

```xml
<click-app>
  <pages package="com.mycorp.page" autobinding="none"/>
</click-app>
```

4.2.3.4. Page Autobinding - Using Annotations

Click provides the Bindable annotation which enables autobinding of Page fields. The Bindable annotation can bind private, protected and public Page fields.

By default, Click's autobinding feature operates on both public and @Bindable fields. To instruct Click to operate only on @Bindable annotated fields, you can set the autobinding attribute to annotation, for example:

```
<click-app>
  <pages package="com.mycorp.page" autobinding="annotation"/>
</click-app>
```

Click won't autobind `public` fields anymore.

Below is an example using the @Bindable annotation:

```
public class EmployeePage extends Page {

    @Bindable protected Form employeeForm = new Form();

    @Bindable protected Table myTable = new Table();

}
```

4.2.4. Headers

The optional `headers` element defines a list of `header` elements which are applied to all pages.

```
<!ELEMENT headers (header*)>
```

The header element defines header name and value pairs which are applied to the HttpServletResponse.

```
<!ELEMENT header (#PCDATA)>
  <!ATTLIST header name CDATA #REQUIRED>
  <!ATTLIST header value CDATA #REQUIRED>
  <!ATTLIST header type (String|Integer|Date) "String">
```

Page headers are set after the Page has been constructed and before `onInit()` is called. Pages can then modify their headers property using the setHeader() method.

4.2.4.1. Browser Caching

Headers are typically used to switch off browser caching. By default Click will use the following no caching header values if you don't define a `headers` element in your application:

```
<click-app>
  <pages>
    ..
  </pages>
  <headers>
    <header name="Pragma" value="no-cache"/>
    <header name="Cache-Control"
            value="no-store, no-cache, must-revalidate, post-check=0, pre-check=0"/>
    <header name="Expires" value="1" type="Date"/>
  </headers>
</click-app>
```

Alternatively you can define your headers individually in pages or for all application pages by setting header values. For example to switch off caching in the login page, note the value for a Date type should be a long number value:

```
<pages package="com.mycorp.page">
  <page path="login.htm" classname="Login">
    <header name="Pragma" value="no-cache"/>
    <header name="Expires" value="1" type="Date"/>
  </page>
</pages>
```

If you wanted to enable caching for a particular page you could set the following page cache control header. This will mark the page as cachable for a period of 1 hour after which it should be reloaded.

```
<pages package="com.mycorp.page">
  <page path="home.htm" classname="Home">
    <header name="Cache-Control" value="max-age=3600, public, must-revalidate"/>
  </page>
</pages>
```

To apply header values globally define header values in the headers element. For example:

```
<click-app>
  <pages>
     ..
  </pages>
  <headers>
    <header name="Pragma" value="no-cache"/>
    <header name="Cache-Control"
              value="no-store, no-cache, must-revalidate, post-check=0, pre-check=0"/>
    <header name="Expires" value="1" type="Date"/>
  </headers>
</click-app>
```

4.2.5. Format

The optional `format` element defines the Format object classname which is applied to all pages.

```
<!ELEMENT format (#PCDATA)>
  <ATTLIST format classname CDATA #FIXED "org.apache.click.util.Format">
```

By default all Click pages are configured with a [org.apache.click.util.Format](#) object. The format object is made available in the Velocity page templates using the name [$format](#).

To specify a custom format class configure a `format` element in the click-app descriptor. For example:

```
<click-app>
  ..
  <format classname="com.mycorp.util.CustomFormat"/>
</click-app>
```

4.2.6. Mode

The optional `mode` element defines the application logging and caching mode.

```
<!ELEMENT mode (#PCDATA)>
  <ATTLIST mode value (production|profile|development|debug|trace) "development">
```

By default Click applications run in `development` mode, which switches off page template caching, and the logging level is set to `INFO`.

To change the default application mode configure a mode element in the click-app descriptor. For example to specify `production` mode you would add the following mode element:

```
<click-app>
  ..
  <mode value="production">
</click-app>
```

The application mode configuration can be overridden by setting the system property `"click.mode"`. This can be use in the scenario of debugging a problem on a production system, where you change the mode to `trace` by setting the following system property and restarting the application.

```
-Dclick.mode=trace
```

The Click Application modes and their settings for Page auto loading, template caching and logging levels are:

Application mode	Page auto loading	Template caching	Click log level	Velocity log level
production	No	Yes	WARN	ERROR
profile	No	Yes	INFO	ERROR
development	Yes	No	INFO	ERROR
debug	Yes	No	DEBUG	ERROR
trace	Yes	No	TRACE	WARN

4.2.6.1. Page Auto Loading

When Page Auto Loading is enabled any new page templates and classes will be automatically loaded at runtime. These pages are loaded using the Page Automapping [54] rules.

Page auto loading is a very handy feature for rapid development as you do not have to restart you application server to pick up new pages.

4.2.6.2. Click and Velocity Logging

The Click and Velocity runtimes use LogService for logging messages. The default LogService implementation is ConsoleLogService which will send messages to the console [System.out]. For example the following logging output is for a HomePage request when the application mode is `trace`:

```
[Click] [debug] GET http://localhost:8080/quickstart/home.htm
[Click] [trace]   invoked: HomePage.<<init>>
[Click] [trace]   invoked: HomePage.onSecurityCheck() : true
[Click] [trace]   invoked: HomePage.onInit()
```

```
[Click] [trace]    invoked: HomePage.onGet()
[Click] [trace]    invoked: HomePage.onRender()
[Click] [info ]    renderTemplate: /home.htm - 6 ms
[Click] [trace]    invoked: HomePage.onDestroy()
[Click] [info ] handleRequest:  /home.htm - 24 ms
```

Any unhandled `Throwable` errors are logged by the ClickServlet.

Note that Click Extras also provide log adaptors for [Log4J](#) and the [JDK Logging API](#).

When an application is not in `production` mode the error page displays detailed debugging information. When the application mode is `production` no debug information is displayed to prevent sensitive information being revealed. This behaviour can be changed by modifying the deployed `click/error.htm` page template.

4.2.7. Controls

The optional `controls` element defines a list of `control` elements which will be deployed on application startup.

```
<!ELEMENT controls (control*)>
```

The control registers [Control](#) classes which will have their [onDeploy()](#) method invoked when the click application starts.

```
<!ELEMENT control (#PCDATA)>
  <!ATTLIST control classname CDATA #REQUIRED>
```

For example to have a `CustomField` control deploy its resources on application startup, you would add the following elements to your `click.xml` file:

```
<click-app>
   ..

   <controls>
     <control classname="com.mycorp.control.CustomField"/>
   </controls>
</click-app>
```

4.3. Auto Deployed Files

To make pre-configured resources (templates, stylesheets, etc.) available to web applications, Click automatically deploys configured classpath resources to the [/click](#) directory at startup (if not already present).

You can modify these support files and Click will **not** overwrite them. These files include:

• click/error.htm   -   the Page Error Handling [30] template

• click/control.css   -   the Controls cascading stylesheet

• click/control.js   -   the Controls JavaScript library

• click/not-found.htm   -   the Page Not Found [31] template

For example to customize the control styles you can place a customized copy (or even a brand new version) of `control.css` under the /click folder in your web project:

```
/webapp/click/control.css
```

When Click starts up it will **not** override your copy of `control.css` with its own default version.

Different controls might deploy different stylesheet, javascript or image files, however the above principle still applies. By placing a customized copy of the stylesheet, javascript or image under the /click folder, you will override the default resource.

Be aware that some of the more complex controls (checklist, colorpicker, tree), deploys resources to subfolders under /click, for example `/click/checklist/*`.

A control's Javadoc will normally indicate what resources are deployed for that control.

4.3.1. Deploying resources in a restricted environment

Some environments place restrictions on the file system and Click won't be able to deploy its resources. WebLogic and Google App Engine are examples of such environments. (Note that WebLogic has a property to allow access to the file system. From the Admin Console go to the *Server node > Web Applications* tab and check the *Archived Real Path Enabled* parameter.)

If Click cannot deploy its resources because of limited file system access or permissions, warning messages will be logged.

Note: if your application is running on a *Servlet 3.0* compliant server, there is no need to deploy resources. Servlet 3.0 specifies that if the server cannot find a resource in the root directory of the webapp, it will look for the resource under *'META-INF/resources'*, and if found, serve it up. Click is Servlet 3.0 compliant and packages its resources under *'META-INF/resources'*.

Click provides a number of options to make resources available in restricted environments which is covered below:

* The first option (which will work in all environments) is to deploy the resources at build time. Click ships with an Ant Task called DeployTask that deploys Click static resources to a web application. With this option Click's static resources can be copied to the root directory of your webapp, where you can customize the resources further if needed. The DeployTask can easily be incorporated into your build script.

  Currently the DeployTask is part of the `click-dev-tools-xxx.jar` that can be found in your Click distribution under the *lib* folder.

  Here is a basic example:

```
<target name="deploy" description="Deploy static resources">
    <taskdef name="deploy"
             classname="org.apache.click.tools.deploy.DeployTask"
             classpath="<click-distribution>/lib/click-dev-tasks-1.1.jar"/> ❶

    <deploy dir="<webapp-root>/WEB-INF"
            todir="<webapp-root>"/> ❷
</target>
```

❶ `<click-distribution>` is the location where Click is installed on your machine, for example: `C:\software\click-2.1.0\`.

❷ `<webapp-root>` is the root directory of your webapp, for example: `C:\dev\my-webapp\`.

We use the *<deploy>* Ant Task and specify the attributes dir and todir.

dir specifies the *source* directory to scan for JARs and folders containing static resources, while todir specifies the *target* directory where the resources should be copied to.

dir should point to your web application's *WEB-INF* folder, since that is where Click's JARs will be located. todir should point to your web application's root directory, since that is where Click's resources will be served from.

The DeployTask also supports nested FileSets if you need to deploy resources from multiple source locations. For example:

```
<target name="deploy" description="Deploy static resources">
    <taskdef name="deploy"
             classname="org.apache.click.tools.deploy.DeployTask"
             classpath="<click-distribution>/lib/click-dev-tasks-1.1.jar"/>

    <deploy todir="${dir.webapp}">
        <fileset dir="<webapp-root>/WEB-INF">
            <include name="**/classes"/>
            <include name="**/*.jar"/>
        </fileset>
        <fileset dir="/some/folder/with/jars">
            <include name="lib-with-resources.jar"/>
            <include name="another-lib-with-resources.jar"/>
        </fileset>
    </deploy>
</target>
```

The DeployTask also generates an HTML report in the same folder where the build script is executed from. The report will indicate which resources was deployed successfully and which resources in your webapp root directory is outdated. (An outdated resource means that the resource in the *click-xxx.jar* differs from the resource currently present in your webapp root directory. This can happen when upgrading to a new version of Click)

• Another option is to add a mapping in *web.xml* to inform ClickServlet to serve static resources. This feature is made available through the ResourceService interface and its default implementation, ClickResourceService. Below is an example:

```
<servlet>
  <servlet-name>ClickServlet</servlet-name>
  <servlet-class>org.apache.click.ClickServlet</servlet-class>
  <load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>ClickServlet</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>

  <!-- Inform ClickServlet to serve static resources contained under the /click/*
```

```
       directory directly from Click's JAR files. -->
<servlet-mapping>
  <servlet-name>ClickServlet</servlet-name>
  <url-pattern>/click/*</url-pattern>
</servlet-mapping>
```

With this setup, ClickServlet will serve all static /click/* resources directly from Click's JAR files.

One restriction of ClickResourceService is it only serves resources from the */click/*￼* folder. So if you use third-party Click libraries that serve their resources from a different folder e.g. */clickclick/*￼*, this option won't work out-of-the-box.

Also note that with this option Click's resources are served directly from the JAR files, you won't be able to customize the resources, if for example you want change the default styling through CSS.

• Another option is to manually deploy the resources. Click resources are packaged in JARs under the directory *META-INF/resources*. You can use your IDE to navigate into the JARs and copy all the resources from *META-INF/resources* to your webapp root directory.

For example, to deploy the resources from *click-core.jar*, copy the */click* folder and its contents to your web application root folder.

• And finally you can access Click's resources by deploying your application on a development machine where there are no file system restrictions and the WAR/EAR can be unpacked. You can then copy the deployed resources to your webapp root directory.

4.3.2. Deploying Custom Resources

Click supports two ways of deploying pre-configured resources (templates, stylesheets, JavaScript etc.) from a Jar to a web application. (This assumes that the environment Click is running in supports having write access to the file system and that the WAR is unpacked.)

1. Through a Control's onDeploy() event handler. See the Controls [60] section above.

2. By packaging the resources (stylesheets, JavaScript, Images etc.) into a special folder called *'META-INF/resources'*.

As option #1 was already discussed above in section Controls [60], lets look at option #2.

When Click starts up, it scans each Jar and folder on the classpath for specially marked entries starting with 'META-INF/resources/'. (Please note that even though Click will scan the entire classpath it is strongly recommended to host your Jar files under your WAR lib folder e.g. WEB-INF/lib. Sharing Jars on the classpath can lead to class loading issues.)

Click will then copy all files found under 'META-INF/resources/' to the root directory of the webapp.

For example, given a Jar file with the following entries:

• META-INF/resources/mycorp/edit_customer.js

• META-INF/resources/mycorp/edit_customer.css

• mycorp/pages/EditCustomerPage.class

Click will copy the files *'/mycorp/edit_customer.js'* and *'/mycorp/edit_customer.css'* to the web application folder.

Thus if the web application is called 'webapp', the files will be deployed as *'webapp/mycorp/edit_customer.js'* and *'webapp/mycorp/edit_customer.css'*.

Option #2 is the recommended approach for deploying your own resources since it makes the managing and maintenance of resources much easier.

Chapter 5. Best Practices

This chapter discusses Best Practices for designing and building Apache Click applications.

5.1. Security

For application security it is highly recommended that you use the declarative JEE Servlet path role based security model. While Click pages provide an `onSecurityCheck()` method for rolling your own programmatic security model, the declarative JEE model provides numerous advantages.

These advantages include:

* Its an industry standard pattern making development and maintenance easier.

* Application servers generally provide numerous ways of integration with an organisations security infrastructure, including LDAP directories and relational databases.

* Servlet security model support users bookmarking pages. When users go to access these pages later, the container will automatically authenticate them before allowing them to access the resource.

* Using this security model you can keep your Page code free of security concerns. This makes you code more reusable, or at least easier to write.

If your application has very fine grained or complex security requirements you may need to combine both the JEE declarative security model and a programmatic security model to meet your needs. In these cases its recommended you use declarative security for course grained access and programmatic security for finner grained access control.

5.1.1. Declarative Security

The declarative JEE Servlet security model requires users to be authenticated and in the right roles before they can access secure resources. Relative to many of the JEE specifications the Servlet security model is surprisingly simple.

For example to secure admin pages, you add a security constraint in your `web.xml` file. This requires users to be in the admin role before they can access to any resources under the admin directory:

```xml
<security-constraint>
    <web-resource-collection>
        <web-resource-name>admin</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

The application user roles are defined in the `web.xml` file as `security-role` elements:

```xml
<security-role>
    <role-name>admin</role-name>
```

```
</security-role>
```

The Servlet security model supports three different authentication method:

- BASIC - only recommended for internal applications where security is not important. This is the easiest authentication method, which simply displays a dialog box to users requiring them to authenticate before accessing secure resources. The BASIC method is relatively insecure as the username and password are posted to the server as a Base64 encoded string.

- DIGEST - recommended for internal applications with a moderate level of security. As with BASIC authentication, this method simply displays a dialog box to users requiring them to authenticate before accessing secure resources. Not all application servers support DIGEST authentication, with only more recent versions of Apache Tomcat supporting this method.

- FORM - recommended applications for where you need a customised login page. For applications requiring a high level of security it is recommended that you use the FORM method over HTTPS.

The authentication method is specified in the <login-method> element. For example to use the BASIC authentication method you would specify:

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Admin Realm</realm-name>
</login-config>
```

To use the FORM method you also need to specify the path to the login page and the login error page:

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Secure Realm</realm-name>
    <form-login-config>
        <form-login-page>/login.htm</form-login-page>
        <form-error-page>/login.htm?auth-error=true</form-error-page>
    </form-login-config>
</login-config>
```

In your Click login.htm page you need to include a special j_security_check form which includes the input fields j_username and j_password. For example:

```
#if ($request.getParameter("auth-error"))
<div style="margin-bottom:1em;margin-top:1em;color:red;">
  Invalid User Name or Password, please try again.<br/>
  Please ensure Caps Lock is off.
</div>
#end

<form method="POST" action="j_security_check" name="form">
<table border="0" style="margin-left:0.25em;">
 <tr>
   <td><label>User Name</label><font color="red">*</font></td>
   <td><input type="text" name="j_username" maxlength="20" style="width:150px;"/></td>
   <td> </td>
  </tr>
```

```
  <tr>
   <td><label>User Password</label><font color="red">*</font></td>
   <td><input type="password" name="j_password" maxlength="20" style="width:150px;"/></td>
   <td><input type="image" src="$context/images/login.png" title="Click to Login"/></td>
  </tr>
</table>
</form>

<script type="text/javascript">
  document.form.j_username.focus();
</script>
```

When using FORM based authentication do **NOT** put application logic in a Click Login Page class, as the role of this page is to simply render the login form. If you attempt to put navigation logic in your Login Page class, the JEE Container may simply ignore it or throw errors.

Putting this all together below is a `web.xml` snippet which features security constraints for pages under the admin path and the user path. This configuration uses the FORM method for authentication, and will also redirect unauthorized (403) requests to the `/not-authorized.htm` page.

```
<web-app>

    ..

    <error-page>
        <error-code>403</error-code>
        <location>/not-authorized.htm</location>
    </error-page>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>admin</web-resource-name>
            <url-pattern>/admin/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
    </security-constraint>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>user</web-resource-name>
            <url-pattern>/user/*</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>FORM</auth-method>
        <realm-name>Secure Zone</realm-name>
        <form-login-config>
            <form-login-page>/login.htm</form-login-page>
            <form-error-page>/login.htm?auth-error=true</form-error-page>
```

```
            </form-login-config>
    </login-config>

    <security-role>
        <role-name>admin</role-name>
    </security-role>

    <security-role>
        <role-name>user</role-name>
    </security-role>

</web-app>
```

5.1.2. Alternative Security solutions

There are also alternative security solutions that provide extra features not available in JEE, such as RememberMe functionality, better resource mapping and `Post Logon Page` support. (`Post Logon Page` support allows one to specify a default URL where the user will be forwarded after successful login. This feature allows one to embed a login form in all non-secure pages and after successful authentication the user will be forwarded to their home page.)

Below are some of the alternative security solutions available:

*   [Spring Security](#)

*   [SecurityFilter](#)

*   [JSecurity](#)

5.1.3. Resources

For more information on using security see the resources below:

*   [Form Based Authentication](#) by Louis E. Mauget

*   [Servlet Specification](#) by Sun Microsystems

*   [Basic authentication scheme](#)

*   [Digest authentication scheme](#)

*   [Https URI scheme](#)

5.2. Packages and Classes

An excellent way to design your project package structure is to classify packages initially by technology. So in a Click application all of our pages would be contained under a page package. This also works very well with the Page automapping feature.

All the projects domain entity classes would be contained under a entity package, and service classes would be contained under a service directory. Note alternative names for the entity package include domain or model. We also typically have a util package for any stray classes which don't quite fit into the other packages.

In Java, package names are singular by convention, so we have a util package rather than a utils package.

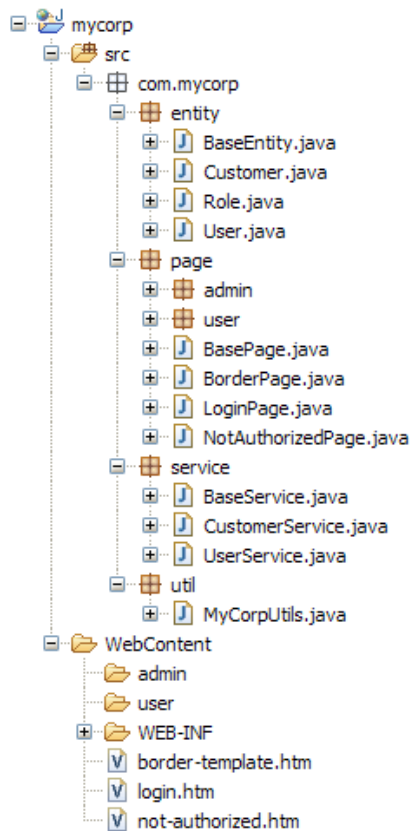An example project structure for a MyCorp web application is illustrated below:

Figure 5.1. Example project structure

In this example application we use declarative role and path based security. All the pages in the admin package and directory require the `"admin"` role to be access, while all the pages in the user package and directory require the `"user"` role to be accessed.

5.2.1. Page Classes

A best practice when developing application Page classes is to place common methods in a base page class. This is typically used for providing access methods to application services and logger objects.

For example the BasePage below provides access to Spring configured service objects and a Log4J logger object:

```java
public class BasePage extends Page implements ApplicationContextAware {

    /** The Spring application context. */
    protected ApplicationContext applicationContext;

    /** The page Logger instance. */
    protected Logger logger;

    /**
     * Return the Spring configured Customer service.
```

```
     *
     * @return the Spring configured Customer service
     */
    public CustomerService getCustomerService() {
        return (CustomerService) getBean("customerService");
    }

    /**
     * Return the Spring configured User service.
     *
     * @return the Spring configured User service
     */
    public UserService getUserService() {
        return (UserService) getBean("userService");
    }

    /**
     * Return the page Logger instance.
     *
     * @return the page Logger instance
     */
    public Logger getLogger() {
        if (logger == null) {
            logger = Logger.getLogger(getClass());
        }
        return logger;
    }

    /**
     * @see ApplicationContextAware#setApplicationContext(ApplicationContext)
     */
    public void setApplicationContext(ApplicationContext applicationContext)  {
        this.applicationContext = applicationContext;
    }

    /**
     * Return the configured Spring Bean for the given name.
     *
     * @param beanName the configured name of the Java Bean
     * @return the configured Spring Bean for the given name
     */
    public Object getBean(String beanName) {
        return applicationContext.getBean(beanName);
    }

}
```

Applications typically use a border template and have a `BorderPage` which extends `BasePage` and defines the template. For example:

```
public class BorderPage extends BasePage {

    /** The root Menu item. */
    public Menu rootMenu = new Menu();

    /**
     * @see Page#getTemplate()
```

```
     */
    public String getTemplate() {
        return "/border-template.htm";
    }
}
```

Most application pages subclass `BorderPage`, except AJAX pages which have no need for a HTML border template and typically extend `BasePage`. The `BorderPage` class should not include common logic, other than that required for rendering the border template. Common page logic should be defined in the `BasePage` class.

To prevent these base Page classes being auto mapped, and becoming directly acessible web pages, ensure that there are no page templates which could match their class name. For example the `BorderPage` class above will not be auto mapped to `border-template.htm`.

5.3. Page Auto Mapping

You should use the Click page automapping configuration feature. See the <u>Page Automapping [54]</u> topic for details.

Automapping will save you from having to manually configure URL path to Page class mappings in your `click.xml` file. If you follow this convention it is very easy to maintain and refactor applications.

You can also quickly determine what the corresponding Page class is for a page HTML template and visa versa, and if you use the ClickIDE Eclipse plugin you can switch between a page's class and template by pressing Ctrl+Alt+S.

An example `click.xml` automapping configuration is provided below (automapping is enabled by default):

```
<click-app>
  <pages package="com.mycorp.dashboard.page"/>
</click-app>
```

To see how the page templates are mapped to Page classes set the application <u>mode [58]</u> to `debug` and at startup the mappings will be listed out. An example Click startup listing is provided below:

```
[Click] [debug] automapped pages:
[Click] [debug] /category-tree.htm -> com.mycorp.dashboard.page.CategoryTree
[Click] [debug] /process-list.htm -> com.mycorp.dashboard.page.ProcessList
[Click] [debug] /user-list.htm -> com.mycorp.dashboard.page.UserList
```

5.4. Navigation

When navigating between Pages using forwards and redirects, you should refer to the target page using the Page class rather than using path. This provides you compile time checking and will save you from having to update path strings in Java code if you move pages about.

To forward to another page using the Page class:

```
public class CustomerListPage extends Page {

    public ActionLink customerLink = new ActionLink(this,"onCustomerClick");

    ..
```

```
    public boolean onCustomerClick() {
        Integer id = customerLink.getValueInteger();
        Customer customer = getCustomerService().getCustomer(id);

        CustomerDetailPage customerDetailPage = (CustomerDetailPage)
            getContext().createPage(CustomerDetailPage.class);

        customerDetailPage.setCustomer(customer);
        setForward(customerDetailPage);

        return false;
    }
}
```

To redirect to another page using the Page class you can obtain the pages path from the `Context`. In the example below we are passing through the customer id as a request parameter to the target page.

```
public class CustomerListPage extends Page {

    public ActionLink customerLink = new ActionLink(this, "onCustomerClick");

    ..

    public boolean onCustomerClick() {
        String id = customerLink.getValueInteger();

        String path = getContext().getPagePath(CustomerDetailPage.class);
        setRedirect(path + "?id=" + id);

        return false;
    }
}
```

A quick way of redirecting to another page is to simply refer to the target class. The example below logs a user out, by invalidating their session, and then redirects them to the application home page.

```
public boolean onLogoutClick() {
    getContext().getSession().invalidate();

    setRedirect(HomePage.class);

    return false;
}
```

5.5. Templating

Use Page templating it is highly recommended. Page templates provide numerous advantages including:

- greatly reduce the amount of HTML you need to maintain

- ensure you have a common look and feel across your application

- make global application changes very easy

To see how to use templates see the Page Templating [26] topic. Also see the Click Examples use of page templating.

5.6. Menus

For many applications using the Menu control to centralize application navigation is very useful. Menus are defined in a `WEB-INF/menu.xml` file which is very easy to change.

A menu is typically defined in the a page border template so they are available through out the application. The Menu control does not support HTML rendering, so you need to define a Velocity macro to programmatically render the menu. You would call the macro in your border template with code like this:

```
#writeMenu($rootMenu)
```

An advantage of using a macro to render your menu is that you can reuse the code across different applications, and to modify an applications menu you simply need to edit the `WEB-INF/menu.xml` file. A good place to define your macros is in the webroot `/macro.vm` file as it is automatically included by Click.

Using macros you can create dynamic menu behaviour such as only rendering menu items a user is authorized to access with isUserInRoles().

```
#if ($menu.isUserInRoles())
  ..
#end
```

You can also use JavaScript to add dynamic behaviour such as drop down menus, for example see the Menu page in Click Examples.

5.7. Logging

For application logging you should use one of the well established logging libraries such as Java Util Logging (JUL) or Log4J.

The library you use will largely depend upon the application server you are targeting. For Apache Tomcat or RedHat JBoss the Log4j library is a good choice. While for the IBM WebSphere or Oracle WebLogic application servers Java Util Logging is better choice as this library is better supported.

If you have to target multiple application servers you should consider using the SLF4J library which uses compile time bindings to an underlying logging implementation.

As a general principle you should probably avoid Commons Logging because of the class loading issues associated with it. If you are using Commons Logging please make sure you have the latest version.

It is a best place to define a logger method in a common base page, for example:

```
public class BasePage extends Page {

    protected Logger logger;

    public Logger getLogger() {
        if (logger == null) {
```

```
            logger = Logger.getLogger(getClass());
        }
        return logger;
    }
}
```

Using this pattern all your application bases should extend `BasePage` so they can use the `getLogger()` method.

```
public class CustomerListPage extends BasePage {

    public void onGet() {
        try {
            ..
        } catch (Exception e) {
            getLogger().error(e);
        }
    }
}
```

If you have some very heavy debug statement you should possibly use an `isDebugEnabled` switch so it is not invoked if debug is not required.

```
public class CustomerListPage extends BasePage {

    public void onGet() {
        if (getLogger().isDebugEnabled()) {
            String msg = ..

            getLogger().debug(msg);
        }

        ..
    }
}
```

Please note the Click logging facility is not designed for application use, and is for Click internal use only. When Click is running in `production` mode it will not produce any logging output. By default Click logs to the console using ConsoleLogService.

If you need to configure Click to log to an alternative destination please configure a LogService such as JdkLogService, Log4JLogService or Slf4jLogService.

5.8. Error Handling

In Click unhandled errors are directed to the ErrorPage for display. If applications require additional error handling they can create and register a custom error page in `WEB-INF/click.xml`. For example:

```
<pages package="com.mycorp.page" automapping="annotation"/>
  <page path="click/error.htm" classname="ErrorPage"/>
</pages>
```

Generally applications handle transactional errors using service layer code or via a servlet Filter and would not need to include error handling logic in an error page.

Potential uses for a custom error page include custom logging. For example if an application requires unhandled errors to be logged to an application log (rather than System.out) then a custom [ErrorPage](#) could be configured. An example `ErrorPage` error logging page is provided below:

```
package com.mycorp.page.ErrorPage;
..

public class ErrorPage extends org.apache.click.util.ErrorPage {

    public void onDestroy() {
        Logger.getLogger(getClass()).error(getError());
    }
}
```

5.9. Performance

Yahoo published a list of [best practices](#) for improving web application performance.

Click Framework provides a [PerformanceFilter](#) which caters for some of these rules. However not all rules can be easily automated.

This section outlines how to apply some important rules which are not covered by the PerformanceFilter namely, [Minimize HTTP Requests (by combining files)](#) and [Minify JavaScript and CSS](#).

The Rule, [Minimize HTTP Requests](#), also mentions [CSS Sprites](#), a method for combining multiple images into a single master image. CSS Sprites can boost performance when your application has many images, however it is harder to create and maintain. Note that CSS Sprites is not covered here.

It is worth pointing out that its not necessary to optimize every page in your application. Instead concentrate on popular pages, for example a web site's *Home Page* would be a good candidate.

There are a couple of tools that are useful in applying the rules "Minimize HTTP Requests" and "Minify JavaScript and CSS":

- [YUICompressor](#) - minifies and compresses JavaScript and CSS files so less bytes have to be transferred across the wire.

- [Ant Task for YUICompressor](#) - an Ant task that uses YUICompressor to compress JavaScript and CSS files.

- [JSMin](#) - similar to YUICompressor but only minifies (remove whitespace and newlines) JavaScript files and does no compression at all. An advantage of JSMin over YUICompressor is that its faster and can be used at runtime to minify JavaScript, while YUICompressor is most often used at build time.

Below are some articles outlining how to use YUICompressor and Ant to concatenate and compress JavaScript and CSS files:

- [Article](#) explaining how to use Ant and YUICompressor for compression.

- [Article](#) outlining how to use a special YUICompressor Ant Task for compression.

Using one of the approaches above you can concatenate and compress all JavaScript and CSS for your Pages into two separate files, for example `home-page.css` and `home-page.js`. Note that the two files must include all the

JavaScript and CSS that is generated by the Page and its Controls. Then you can instruct Click to *only* include the two compressed files, home-page.css and home-page.js.

The Click Page class exposes the property includeControlHeadElements that indicates whether Controls have their CSS and JavaScript resources included or not.

To optimize Page loading one can override Page.getHeadElements(), and import the JavaScript and CSS files and then set the property includeControlHeadElements to `false`, indicating that Controls won't contribute their own JavaScript and CSS resources.

Here is an example:

```java
public class HomePage extends Page {

    private Form form = new Form("form");

    public void onInit() {
        // Indicate that Controls should not import their head elements
        setIncludeControlHeadElements(false);

        form.add(new EmailField("email"));
        addControl(form);
    }

    public List getHeadElements() {
        if (headElements == null) {
            headElements = super.getHeadElements();

            headElements.add(new CssImport("/assets/css/home-page.css"));
            headElements.add(new JsImport("/assets/js/home-page.js"));
        }
        return headElements;
    }
}
```

Using the following `border-template.htm`:

```html
<html>
  <head>
    <title>Click Examples</title>
    ${headElements}
  </head>
  <body>

  ...

  ${jsElements}
  </body>
</html>
```

the rendered HTML will include one CSS and one JavaScript import:

```html
<html>
  <head>
    <title>Click Examples</title>
```

```
    <link type="text/css" rel="stylesheet"
          href="/click-examples/assets/css/home-page.css" title="Style"/>
  </head>
  <body>

  ...

  <script type="text/javascript" src="/click-examples/assets/js/home-page.js"></script>
  </body>
</html>
```

A live demo is available here