

RichFaces CDK Developer Guide

1. Introduction	1
2. Roadmap document	3
3. Setting up the environment	5
4. inputDate component development	9
4.1. Creating project for component	9
4.2. Creating and building component skeleton	12
4.3. Creating a UI prototype	13
4.4. Creating a Renderer	14
4.4.1. Templating	14
4.4.2. Creating a Renderer Base class	16
4.4.3. Skinnability	19
4.5. Component resources registration	19
4.5.1. resources-config.xml file format	19
4.6. Configuring component	19
4.7. Creating tag classes and descriptor for JSP and Facelets	19
5. Generating unit tests	21
6. Component usage overview	23
6.1. Developer sample creation	23
7. Button component development	25
8. Creating projects in different IDEs	27
9. Naming conventions	29
10. CDK Tag Reference	31
11. Template tags overview	33

Introduction

The major benefit of the JSF framework is a component based architecture. The component in JSF is not just a set of HTML code rendered and interpreted by a browser. The JSF component is a composition of a client-side widget coupled with the server-side object that represents component behavior including data validation, events handling, business layers bean binding, etc.

In contrast to a page-oriented development approach, JSF allows to use a component-oriented paradigm to build a well-designed, highly customizable UI interface based on reusable components.

However, there is not yet enough sets of rich components on the market that might enable the rapid application developments. One of important problems is a long and very complicated process of the component creation. Even the very primitive JSF component requires to write the `UIComponent` class, `Renderer` class, `Tag` class and a faces configuration file (`faces-config.xml`).

In order to use the component library in a Facelets environment, you should add the `*.taglib.xml` file to this checklist.

Creation of the rich component takes even more time. You have to provide the `ListenerTagHandler` class, a class for creating a listener interface, an event processing method name in a listener interface, an event class, render specific classes for each possible render kit used with the component.

Therefore, the process of JSF component creation is pretty complicated but repeatable. Jonas Jacobi and John R. Fallows in their "Pro JSF and Ajax Building Rich Internet Components" book describe the process in details. This description and used approaches are very similar to our own experience and have been used as a methodology for Component Development Kit (CDK) - a sub-project of RichFaces that allows you to easily create rich components with built-in Ajax support.

The significant features of the Component Development Kit (CDK) are:

- Quick development start. A new component development starts from a pre-generated component project template. It contains the whole required infrastructure and necessary files generated. It's necessary only to have a [Maven](http://maven.apache.org) [http://maven.apache.org] installed. All other required stuff will be loaded and configured automatically.
- Declarative approach for a component development. It's necessary only to specify meta-data and a code specific for your component. All other required artifacts are generated for you.
- Independent development life-cycle. Component Development Kit (CDK) presumes development of each component isolated from each other with further assembling them into the component library. Hence, this allows to organize a continuous process when one component is already in production condition, but another is just started.

- Possibility to create a first-class rich components with built-in Ajax functionality and add Ajax capability to the existing JSF applications.
- Facility for automatic testing. At the moment of generating the initial project structure, the Unit Test classes are also generated. The RichFaces also provides the Mock-test facility that allows to emulate a run-time environment and automatically test the components before their are gathered into the result library.
- Optimization for different JSF implementations. As soon as the most part of a result code is generated, the Component Development Kit (CDK) becomes able to generate an implementation specific code along with a universal code. It makes sense if it's necessary to optimize a result code using features specific for the particular implementation. For example, for using with JSF 1.2 only.
- Create a modern rich user interface look-and-feel with JSP-like templates and skins-based technology. RichFaces comes with a number of predefined skins to get you started, but you can also easily create your own custom skins.

Roadmap document

This document is aimed to describe components development with the Component Development Kit (CDK) and its features.

In order to be successful in Component Development Kit (CDK) usage and components development, it's necessary to be acquainted with Java Server Faces and RichFaces framework. To read more on these topics, please, follow the links:

- *JavaServer Faces Technology* [<http://java.sun.com/javaee/javaxserverfaces>]
- *Facelets Official Resource* [<https://facelets.dev.java.net>]
- *JSF Official Resource of MyFaces implementation* [<http://java.sun.com/javaee/javaxserverfaces/reference/api/index.html>]
- *RichFaces Official Site* [<http://www.jboss.org/jbossrichfaces>]

Setting up the environment

In order to start working with the Component Development Kit (CDK) and to create your rich component, it's necessary to have the following installed:

- [The Java SE 5 Development Kit \(JDK\)](http://java.sun.com/javase/downloads/index_jdk5.jsp) [http://java.sun.com/javase/downloads/index_jdk5.jsp]
- [Apache Maven 2.0.9](http://maven.apache.org/download.html) [http://maven.apache.org/download.html]
- [Apache Tomcat 6.0](http://tomcat.apache.org) [http://tomcat.apache.org]
- Browser (on client side)

After the Maven is installed you should configure it. In this case, please, go to the directory where you've just installed Maven, open a conf/settings.xml file for editing and add to the profiles section this code:

```
...
<profile>
  <id>cdk</id>
  <repositories>
    <repository>
      <id>maven2-repository.dev.java.net</id>
      <name>Java.net Repository for Maven</name>
      <url>http://download.java.net/maven/1</url>
      <layout>legacy</layout>
    </repository>
    <repository>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
      </snapshots>
      <id>repository.jboss.com</id>
      <name>Jboss Repository for Maven</name>
      <url>http://repository.jboss.com/maven2/</url>
      <layout>default</layout>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>maven.jboss.org</id>
      <name>JBoss Repository for Maven Snapshots</name>
```

```
<url>http://snapshots.jboss.org/maven2/</url>
<releases>
  <enabled>false</enabled>
</releases>
<snapshots>
  <enabled>true</enabled>
  <updatePolicy>always</updatePolicy>
</snapshots>
</pluginRepository>
<pluginRepository>
  <releases>
    <enabled>true</enabled>
  </releases>
  <snapshots>
    <enabled>false</enabled>
    <updatePolicy>never</updatePolicy>
  </snapshots>
  <id>repository.jboss.com</id>
  <name>Jboss Repository for Maven</name>
  <url>http://repository.jboss.com/maven2/ </url>
  <layout>default</layout>
</pluginRepository>
</pluginRepositories>
</profile>
...
```

In order to activate new profile, please, add the following after the profiles section:

```
...
<activeProfiles>
  <activeProfile>cdk</activeProfile>
</activeProfiles>
...
```



Note:

In order to work with Maven from Eclipse, it's possible to download and install the Maven plugin. Please, follow the instruction at [Eclipse plugins for Maven page](http://maven.apache.org/eclipse-plugin.html) [http://maven.apache.org/eclipse-plugin.html]

The environment is set up now to use the Component Development Kit (CDK).

We are going to create two components throughout the RichFaces CDK Developer Guide, but at first you need take the following steps in order to set up the Project and create your library:

- Create a new directory where all the components will be stored (for example Sandbox).
- Create a file named pom.xml in the directory with the following content:

```
...
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.mycompany</groupId>
  <artifactId>sandbox</artifactId>
  <url>http://mycompany.org</url>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.1</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.faces</groupId>
      <artifactId>jsf-api</artifactId>
      <version>1.2_07</version>
    </dependency>
```

```
<dependency>
  <groupId>javax.faces</groupId>
  <artifactId>jsf-impl</artifactId>
  <version>1.2_07</version>
</dependency>
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>el-api</artifactId>
  <version>1.0</version>
</dependency>
<dependency>
  <groupId>el-impl</groupId>
  <artifactId>el-impl</artifactId>
  <version>1.0</version>
</dependency>
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>jsr250-api</artifactId>
  <version>1.0</version>
</dependency>
  <dependency>
    <groupId>org.richfaces.ui</groupId>
    <artifactId>richfaces-ui</artifactId>
    <version>3.2.1.GA</version>
  </dependency>
</dependencies>
</project>
...
```

- Close the file

Here are some of these elements with descriptions:

Table 3.1. The POM elements

Element	Description
groupId	Prefix for the Java package structure of your library
url	Namespace for your library to be used in the TLD file
version	Version of your library

inputDate component development

We are going to create the **<inputDate>** component that can take a value, process that value, and then push it back to the underlying model as a strongly typed Date object.

The **<inputDate>** component allows to attach a converter in order to set the desired date format such as mm/dd/yyyy. So the component could convert and validate the date entered by user.

4.1. Creating project for component

At first we need to create a project for the component itself. In the library directory Sandbox you just created, launch the following command (all in one line):

```
mvn archetype:create -DarchetypeGroupId=org.richfaces.cdk -DarchetypeArtifactId=maven-archetype-jsf-component -DarchetypeVersion=3.2.1.GA -DartifactId=inputDate
```

As is easy to see a new directory with the name inputDate will be created. It does not have any components in it yet, but it has this predefined structure:

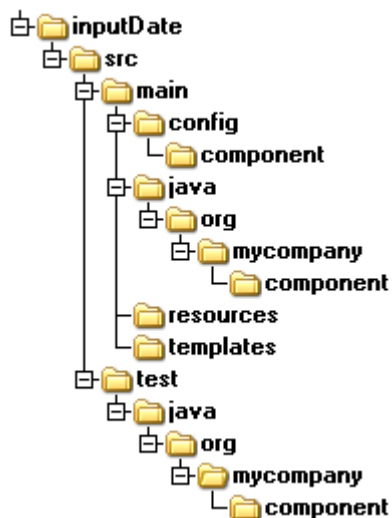


Figure 4.1. The project structure

Here are main directories with descriptions:

Table 4.1. The project structure

Directory	Description
src/main/config	Contains the metadata for the components
src/main/java	Contains Java code (both pre-generated and created by you)

Directory	Description
src/main/resources	Used to store resource files, such as pictures, JavaScript and CSS files
src/main/templates	Used to contain the JSP-like templates that define the component layout

It is necessary to extend predefined structure with the following directories:

Table 4.2. The project structure

Directory	Description
src/main/config/resources	Contains the resource-config.xml file for the resources registration
src/main/java/org/mycompany/renderkit	Contains Renderer Base class
src/main/resources/org/mycompany/renderkit/html/css	Used to store CSS files
src/main/resources/org/mycompany/renderkit/html/images	Used to store images
src/main/resources/org/mycompany/renderkit/html/scripts	Used to store JavaScript files
src/main/templates/org/mycompany	Used to contain the JSP-like template

Now you should add maven-compiler-plugin to the plugins section in the inputDate/pom.xml file:

```
...
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <inherited>true</inherited>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
...
```

Finally your inputDate/pom.xml should look like this one:

```
<?xml version="1.0"?>
<project>
  <parent>
```

```
<artifactId>sandbox</artifactId>
<groupId>org.mycompany</groupId>
<version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>
<groupId>org.mycompany</groupId>
<artifactId>inputDate</artifactId>
<name>inputDate</name>
<version>1.0-SNAPSHOT</version>
<build>
  <plugins>
    <plugin>
      <groupId>org.richfaces.cdk</groupId>
      <artifactId>maven-cdk-plugin</artifactId>
      <version>3.2.1.GA</version>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <goals>
            <goal>generate</goal>
          </goals>
        </execution>
      </executions>
      <configuration>
        <library>
          <prefix>org.mycompany</prefix>
          <taglib>
            <shortName>inputDate</shortName>
          </taglib>
        </library>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <inherited>true</inherited>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
<dependencies>
  <dependency>
```

```
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
  <groupId>org.richfaces.framework</groupId>
  <artifactId>richfaces-impl</artifactId>
  <version>3.2.1.GA</version>
</dependency>
</dependencies>
</project>
```

4.2. Creating and building component skeleton

Let's create a skeleton for the **<inputDate>** component.

You need to go to the inputDate directory and then launch the following command:

```
mvn cdk:create -Dname=inputDate
```

As a result three artifacts will be created:

- An XML configuration file for the metadata
- A UI class
- A JSP-like template

In order to build the component you should stay in the inputDate directory and launch the following command:

```
mvn install
```

This command generates and compiles the library and then creates a result JAR file. A directory named target will be created along with a src directory. If you get a file named target/inputDate-1.0-SNAPSHOT.jar, everything is set up successfully.

If you want to rebuild the component you could use the following command:

```
mvn clean install
```


4.3. Creating a UI prototype

It is a good idea to create at first a prototype of the intended markup. You will find out which markup elements the component has to generate and also which renderer-specific attributes are needed in order to parameterize the generated markup.

The **<inputDate>** component consists of an HTML form **<input>** element, an **** element, and **<div>** element:

```
...
<div title="Date Field Component">
  <input name="dateField" value="01 January 2008" />
  
</div>
...
```

As it is shown in the listing above there are three HTML attributes - *"title"*, *"name"*, and *"value"* - are needed to be parameterize the generated markup.

You map the HTML attributes to the corresponding `UIComponent` attributes:

```
...
<div title="[title]">
  <input name="[clientID]" value="[converted value]" />
  
</div>
...
```

All information about styles applied to the **<inputDate>** component is considered in the following chapter.

This is the result of your prototype which shows a simple page with an input field and an icon indicating that this is a date field:

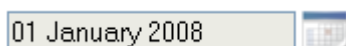


Figure 4.2. The date field component prototype implemented in HTML with an icon

4.4. Creating a Renderer

Component functionality typically centers around two actions: decoding and encoding data. Decoding is the process of converting incoming request parameters to the values of the component. Encoding is converting the current values of the component into the corresponding markup. In the following figure you can see decoding and encoding occur in the JSF lifecycle:

Thus, JSF components consist of two parts: the component class and the renderer. The component class is responsible for the state and behavior of a UI component and will be discussed later in the "Configuring component" section.

The Renderer is responsible for the JSF component representation. It generates the appropriate client-side markup, such as HTML, WML, XUL, etc. Renderer is also responsible for the converting information coming from the client to the proper type for the component (for example, a string value from the request is converted to a strongly type `Date` object).

It is necessary to create following renderer classes for the `<inputDate>` component:

- `InputDateRenderer` class where you should override `encode()` method
- `InputDateRendererBase` class where you should override `decode()` method. You could also define associated converter in this class.

You could actually implement the renderer-specific component subclass that exposes client-side attributes such as `"style"`, `"class"`, etc. It is common practice to implement the client-specific component subclass to make some aspects of application development easier, but in our case we do not need to do it. The `<inputDate>` is a simple `UIInput` component, therefore `InputDateRenderer` class generates all the markup itself.

It is a time to start creating the `InputDateRenderer` class. The smartest way to create the `InputDateRenderer` class is a Templating mechanism, one of the most convenient features of the Component Development Kit (CDK).

4.4.1. Templating

The Component Development Kit (CDK) allows to use templates for generation Renderer class.

Templates are JSP-like markup pages with special tags that are converted into Renderer by a build script.

It's possible to use evaluated expressions in components templates with the help of scriptlets. It's also possible to create the base class for a template to implement additional functions in it, so as the functions could be called from the template. Hence, in the generated Renderer class there are corresponding function calls on the place of these elements.

Let's create the template for HTML 4.0 markup. At first you should proceed to the `inputDate/src/main/templates/org/mycompany` directory where `htmlInputDate.jspx` template file is stored. This file contains a Template Skeleton like this one:

```

<?xml version="1.0" encoding="UTF-8"?>
<f:root
  xmlns:f="http://ajax4jsf.org/cdk/template"
  xmlns:c=" http://java.sun.com/jsf/core"
  xmlns:ui=" http://ajax4jsf.org/cdk/ui"
  xmlns:u=" http://ajax4jsf.org/cdk/u"
  xmlns:x=" http://ajax4jsf.org/cdk/x"
  class="org.mycompany.renderkit.html.InputDateRenderer"
  baseclass="org.ajax4jsf.renderkit.AjaxComponentRendererBase"
  component="org.mycompany.component.UInputDate"
>
  <f:clientId var="clientId"/>
  <div id="#{clientId}"
    x:passThruWithExclusions="value,name,type,id"
  >
  </div>
</f:root>

```

According to the created UI prototype you need to extend Template Skeleton with proper elements:

- **<div>**-wrapper element with *"title"*, *"value"*, *"name"*, *"type"*, *"id"* attributes and with the *"caption"* facet
- **<input>** element with *"id"*, *"value"*, *"name"*, *"type"*, *"class"*, *"style"* attributes and with the *"icon"* facet used to define an icon of the component.
- **** element with *"src"*, *"class"*, *"style"* attributes.

The **<inputDate>** component uses styles and scripts that should be defined in the template with the help of **<h:styles>** and **<h:scripts>** tags. How to register all resources is explained in the "Component resources registration" chapter.



Important:

Don't forget to add namespace `xmlns:h=" http://ajax4jsf.org/cdk/h "` to the **<f:root>** element:

```

<?xml version="1.0" encoding="UTF-8"?>
<f:root
  ...
  xmlns:h=" http://ajax4jsf.org/cdk/h"

```

```
...
>
...
</f:root>
```

Here is a full example of the template for the **<inputDate>** component: [htmlInputDate.jspx](#) [examples/htmlInputDate.jspx].



Note:

In the Template Skeleton Renderer Baseclass is `org.ajax4jsf.renderkit.AjaxComponentRendererBase`, but you need to define Renderer Baseclass special for **<inputDate>** component. In the next section "[Creating a Renderer Base class](#)" we will define Renderer Baseclass `org.mycompany.renderkit.InputDateRendererBase`.

All the Template tags you could find in the "Template tags overview" chapter.

4.4.2. Creating a Renderer Base class

After the component tree is restored on the Restore View Phase , each component in the tree extracts its new value from the request parameters by using its `decode()` method. Then the value is stored locally on the component.

The `InputDateRendererBase` class extends a `HeaderResourcesRendererBase` class. In the `HeaderResourcesRendererBase` class all the `encode()` methods for the right resources encoding are already realized, so in the `InputDateRendererBase` class you need to override the `decode()` method only:

```
public void decode(FacesContext context, UIComponent component){
    ExternalContext external = context.getExternalContext();
    Map requestParams = external.getRequestParameterMap();
    UIInputDate inputDate = (UIInputDate)component;
    String clientId = inputDate.getClientId(context);
    String submittedValue = (String)requestParams.get(clientId);
    if (submittedValue != null) {
        inputDate.setSubmittedValue(submittedValue);
    }
}
```

As you see in the example above the `decode()` method reads values from request parameters, grabs the `clientId` from the component to identify the request parameter to be looked up. The

`clientId` is calculated as the fully qualified name of the component given its container path: `nameOfForm:nameOfComponent` (for example `myForm:inputDate`). The last step of the `decode()` method is to store the submitted value locally on the component.



Note:

By default, the base `Renderer` implementation returns the `submittedValue` directly without any conversion! If you want to convert submitted value to a strongly typed object you should implement `Converter` and the `getConvertedValue()` method in your `Renderer` class (in our case in the `InputDateRendererBase` class).

4.4.2.1. Creating a Converter

As it was mentioned before the `<inputDate>` component at the Apply Request Values phase takes a value and pushes it to the model as a strongly typed `Date` object. Therefore you need to implement a `Converter` in the `Renderer Base` class and also check whether a `Converter` has been already attached by the application developer. If the conversion of the value fails, an error message associated with the component is generated and queued on `FacesContext`.

The `getConverter()` method of the `InputDateRendererBase` class controls the conversion of entered values, as shown in the following example:

```
private Converter getConverter(FacesContext context, UIInputDate inputDate){
    Converter converter = inputDate.getConverter();
    if (converter == null){
        // default the converter
        DateTimeConverter datetime = new DateTimeConverter();
        datetime.setLocale(context.getViewRoot().getLocale());
        datetime.setTimeZone(TimeZone.getDefault());
        datetime.setType("date");
        datetime.setDateStyle("medium");
        datetime.setPattern("d/m/y");
        converter = datetime;
    }
    return converter;
}
```

At first you should check whether the application developer has attached a `Converter` to the `<inputDate>` component (for example, `<f:convertDateTime>`). If not you should follow the next steps:

- create a new `DateTimeConverter`

- get the locale for the client from the context with the help of the `getLocale()` method and set it on the new Converter by means of the `setLocale()` method
- set the time zone, date type, date style, and date pattern on the new converter with the help of `setTimeZone`, `setType`, `setDateStyle`, and `setPattern` methods respectively
- return the Converter

After the Apply Request Values phase the application enters the Process Validation phase during which the `validate()` method calls the `getConvertedValue()` method on every submitted value and passes the newly submitted value from the decode process.

The `getConvertedValue()` method converts the submitted value to a strongly typed object (in our case Date) using the `getAsObject()` method. Then the new strongly typed object is validated and if there are no errors Process Validation phase ends. Otherwise the `getConvertedValue()` method throws a `ConverterException`.

Here is the snippet:

```
public Object getConvertedValue(FacesContext context, UIComponent component, Object
submittedValue) throws ConverterException{
    UIInputDate inputDate = (UIInputDate)component;
    Converter converter = getConverter(context, inputDate);
    String valueString = (String)submittedValue;
    return converter.getAsObject(context, component, valueString);
}
```

Finally on the the Renderer Response phase the value of the component is rendered back to the view. The converter is responsible for transforming the object data back in to a string representation, so you need to implement `getValueAsString`. JSF calls the `getAsString` method of your Converter which converts an Object into a String.

Here is the example:

```
protected String getValueAsString(FacesContext context, UIComponent component) throws
IOException {
    UIInputDate inputDate = (UIInputDate) component;
    String valueString = (String) inputDate.getSubmittedValue();
    if (valueString == null) {
        Object value = inputDate.getValue();
        if (value != null) {
            Converter converter = getConverter(context, inputDate);
            valueString = converter.getAsString(context, component, value);
        }
    }
}
```

```
return valueString;  
}
```

You could find the whole example of the `InputDateRendererBase` class [here](#) [examples/`InputDateRendererBase.java`].

4.4.3. Skinnability

creation

4.4.3.1. XCSS Format

XCSS

4.4.3.2. Dynamic images generation

XCSS

4.5. Component resources registration

Registering component resources

4.5.1. resources-config.xml file format

resources-config.xml file format

4.6. Configuring component

UI Component generation

4.7. Creating tag classes and descriptor for JSP and Facelets

Creating a JSP tag handler and TLD

Generating unit tests

Unit Tests

Component usage overview

overview

6.1. Developer sample creation

Sample creation

Button component development

Command Ajax components TBD with RichFaces CDK.

Creating projects in different IDEs

Eclipse, IDEA.

Naming conventions

Naming

CDK Tag Reference

Tag Reference

Template tags overview

tags

