UserManual

Updated Today (119 minutes ago) by gessedafe

Labels: Featured, Phase-Implementation

1 Introduction

Crux applications are basically GWT applications. It means you must attempt to all GWT restrictions in terms of source code limitations and project structure directives. If you are not familiar with GWT, please consult its documentation first.

This documentation will show you, in a detailed and gradual way, how to use the ***Crux*** framework, by exploring and exemplifying its features. Here you will also learn the concepts behind the code and understand how does ***Crux*** work internally. In order to easy follow this documentation you will need:

- A Java SE Development Kit (JDK), version 1.5 or higher (we recommend 1.6);
- An Eclipse IDE;

To see an example application using ***Crux***, take a look at the our Showcase.

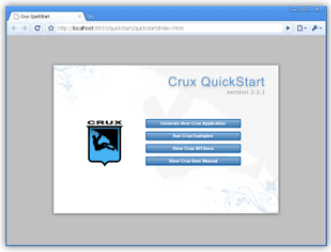
1.1 *Crux* and GWT

Crux is built over GWT and support all of its features. Current ***Crux*** version uses GWT 2.0.

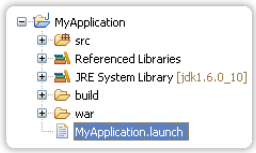
2 Quick Start

2.1 Install

- Download the latest ***Crux*** release here;
- Unzip the archive into a folder you want;
- Be sure that your default JVM is the version 1.6 or higher. It is necessary to execute the ***Crux*** installer (in runtime, Crux projects run fine with Java 1.5);
- Execute the `start.cmd` or `start.sh` file, according to your operating system;
- The ***Crux QuickStart*** tool will start, opened in your default web browser:



- Click the `Generate New Crux Application` button and follow the instructions. At the end of the wizard, an Eclipse project will be generated for you.
- Import the freshly generated project into your Eclipse IDE, and you will get something like this:



- Run the `<your project name>.launch` file, located at the project root folder;
- If everything's gone OK, the GWT DevMode console will appear, and your first application will look like this:

2.2 Environment Configuration

To enable auto-completion on your ***Crux*** pages you must add the project catalog to your Eclipse. This project catalog is located under the folder `xsd` (`crux-catalog.xml`) and includes information about all the ***Crux*** widget libraries that are present in your project classpath.

To add the catalog file to your Eclipse catalogs list, just go to

> `Window->Preferences...->XML Catalog`

Then, select `User Specified Entries` and choose `Add...`. Choose `Next Catalog` and inform the path to the `crux-catalog.xml` file.

2.3 Sample Application

Here we will explore the application generated in the previous section, explaining each single part of it.

- The `war` folder is the root context of your application and contains:
  - a WEB-INF folder, compliant with JEE specifications, where you will find:
    - the `web.xml` file (more details at 4.3.1 Web.xml);
    - the `lib` folder, which stores all ***Crux*** jar files your need at development time (more details at section Setup);
    - a `classes` folder, which is the output for java compilation;
    - a `<the name you gave>.crux.xml` file: the *welcome file* of your application (more about ***Crux*** XML files at 3.1.2 Writing XML Pages;
- the `build` folder, containing:
  - a `lib` folder, containing the files needed by ***Crux*** compilation;
  - a `build.xml` ant file, which defines the following tasks:
    - **dist**: generates the war file for deployment;

        - **compile-scripts**: invokes the ***Crux*** compilation, generating the static files that can be tested in browsers;
        - **generate-schemas**: generates all XSD files you need to auto-complete your XML pages code.
- the `src` folder, containing the source files:
    - `<the name you gave the module>.gwt.xml` - a GWT module which extends ***Crux*** modules (like shown at Coding Client Side);
    - `<the same module package>.client.controller.MyController.java` - a client-side controller for the welcome page (see 3.3.3 Controller);
    - `<the same module package>.client.remote.GreetingService.java` - a client-side business interface;
    - `<the same module package>.client.remote.GreetingServiceAsync.java` - a client-side asynchronous interface for accessing the server;
    - `<the same module package>.server.GreetingServiceImpl.java` - the server-side business class;

  for more about last three items, see 4.1 Writing Server Services.

3 Coding Client Side

Your modules must inherit ***br.com.sysmap.crux.core.Crux***. You don't need to specify an EntryPoint, because ***Crux*** defines a basic one that loads its engine.

The following example shows a typical module which can use all ***Crux*** features:

```
<module rename-to='mymodule'>
        <inherits name='br.com.sysmap.crux.core.Crux'/>
        <inherits name='br.com.sysmap.crux.gwt.CruxGWTWidgets'/>
        <inherits name='br.com.sysmap.crux.widgets.CruxWidgets'/>
</module>
```

The code above creates a module that inherits the ***Crux*** core and the two sets of widgets that compose the default distribution.

- The CruxGWTWidgets set contains all widgets that are distributed directly with GWT. It's packaged in the `crux-gwt-widgets.jar` file.
- The CruxWidgets set contains some complex widgets like MaskedTexBox, multi-frame capable dialogs, etc. It's packaged in the `crux-widgets.jar` file.

3.1 Building User Interface

Any GWT widget can be used in user interface construction. Consult the Widget Developer Manual for information about how to use custom widgets with ***Crux***.

To add widgets to your pages, you can use these methods:

1. Create a page as a XML file and use some XSDs to enable auto completion on your favorite editor.
2. Programmatically instantiate widgets, exactly as you already do using with pure GWT.

3.1.1 Pages as XML Files

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <gwt:textBox id="myBox" />
       <gwt:button id="myButton" text="Hello" onClick="clientHandler.helloWorld" />
   </body>
</html>
```

You must attempt to:

- Your files must have the extension ***.crux.xml***.
- In order to enable auto completion, you will need to configure your IDE to point to all XSD files generated by the Schema Generator:

  If you are using an Eclipse based IDE, it can be done at the menu **Window -> Preferences -> XML -> XML Catalog**

- Configure the DeclarativeUIFilter in your web.xml files as following (if you use the Crux Project Generator, it will already configure the filter for you.):

```
<filter>
   <display-name>DeclarativeUIFilter</display-name>
   <filter-name>DeclarativeUIFilter</filter-name>
   <filter-class>br.com.sysmap.crux.core.declarativeui.filter.DeclarativeUIFilter</filter-class>
   <init-param>
      <param-name>outputCharset</param-name>
      <param-value>ISO-8859-1</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>DeclarativeUIFilter</filter-name>
   <url-pattern>*.html</url-pattern>
</filter-mapping>
```

The **.crux.xml** files is used to turn development easier. When you generate the final application distribution file, the ***Crux*** compiler translate that page into a **.html** page. See Crux Compiler for more information.

It means that you can, for example, create a page called `index.crux.xml`, but the url you must pass to the browser will refer to `index.html`.

3.1.2 Instantiating Widgets Programmatically

You can instantiate widgets exactly as you do using pure GWT.

```
...
Button myButton = new Button();
myButton.addClickHandler(new ClickHandler(){
   public void onClick(ClickEvent event)
   {
       Window.alert("hello");
   }
});
...
```

3.2 Screen

Crux creates an abstraction over the page that is called ***Screen***. Declaratively, you can refer to it using a `<screen>` tag:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <crux:screen onClose="clientHandler.onClose" onLoad="clientHandler.onLoad" useController="clientHandler" />
       <gwt:textBox id="myBox" />
       <gwt:button id="myButton" text="Hello" onClick="clientHandler.helloWorld" />
   </body>
</html>
```

The screen can be retrieved programmatically by a call to the static method `Screen.get()`. Through Screen, you can:

1. Access any widget created declaratively on pages.
2. Add handlers to Window events, like `load`, `close` or `resize`.
3. Communicate with other screens to, for example, exchange data between ***Crux*** pages in different frames or windows of the same application.
4. Control the history without need to add a hidden frame directly on the page.
5. Block and unblock the user interaction with the page.
6. Access a lot of other informations, like client locale.

3.2.1 Access Screen Widgets

To access screen widgets, you can use the static method `Screen.get(screenId)`.

```
	Button myButton = (Button)Screen.get("myButton");
	or
	Button myButton = Screen.get("myButton", Button.class);
```

You can also create an interface to access your widgets on screen. Your interface must extend the interface ScreenWrapper and the methods must follow the pattern: `<widgetType> get<WidgetID>()`. See the example:

```
@Controller("myController")
public class MyClass
{
	public static interface MyScreen extends ScreenWrapper
	{
		Button getMyButton();
		TextBox getMyBox();
	}

	@Create
	protected MyScreen myScreen;

	public void myMethod()
	{
		myScreen.getMyBox().setValue("Test");
	}
}
```

We recommend you create wrappers for screens, that is most elegant and avoid mismatches on typing widgets ids as strings all over the code.

3.2.2 Screen Events

Crux Screen support the following events:

Event	Description
Load	Called when page loads. It is fired after the screen's building process is completed
Close	Called when page is closed
Closing	Called before close the page
Resize	Called on page resize
HistoryChanged	Called when back button is pressed

3.2.3 Communication Between Screens

Using some static methods on Screen class, it is possible to invoke operations on other ***Crux*** screens running in different frames or windows.

The following example shows how this can be done:

```
	Screen.invokeControllerOnTop("topController.method", new MyParameterClass());
```

That code will call the method "method" on controller identified by "topController" on the top page. Consult the section Managing Events for more details about how this call is processed.

The following table shows the methods that allow calls on others screens:

Method	Description
invokeControllerOnParent	call a controller on parent page
invokeControllerOnOpener	call a controller on opener page
invokeControllerOnAbsoluteTop	call a controller on top of the first opener page
invokeControllerOnTop	call a controller on top page
invokeControllerOnSelf	call a controller on self
invokeControllerOnFrame`<FrameName>`	call a controller on an inner frame named `<frameName>`
invokeControllerOnSiblingFrame`<FrameName>`	call a controller on a sibling inner frame named `<frameName>`

You can create an interface to wrap the invocations to a specific controller. Your interface must extends the interface `Invoker` and the methods must follow the pattern: `<returnType> <methodName>On<position>(<paramType>);`. See the example:

```
public interface MyControllerInvoker extends Invoker
{
   void myMethodOnTop(String[] params); // is the same that Screen.invokeControllerOnTop("myController.myMethod", params);
   String mySecondMethodOnSelf();
   String mySecondMethodOnOpener(Integer param0, String param1, String[] param2);
   String mySecondMethodOnFrameTest(Integer param0, String param1, String[] param2);
}
```

```
   ...
   MyControllerInvoker invoker = GWT.create(MyControllerInvoker.class);
   invoker.myMethodOnTop(new String[]{"value1", "value2"});
   ...
```

To inform the name of the controller that will be bound to the invoker interface, you can use the annotation `@ControllerName`. If it is not present, ***Crux*** will use the name of the invoker interface without the suffix 'Invoker'.

The above interface could also be written as:

```
@ControllerName("controller")
public interface MyControllerInvoker extends Invoker
{
   ...
}
```

Another way of information sharing is available for ***Crux*** screens. You can use a common context to read and write variables.

The ***Crux*** Context is a common area where you can put and read values associating them with a key.

To turn easier the access to context information, you can define an interface that extends the interface `Context`. Its methods must follow the pattern: `<valueType> get<valueKey>()` or `void set<valueKey>(valueType)`. See the example:

```
public interface MyContextWrapper extends Context
{
	Double getValueOne();
	String[] getValueTwo();
	void setValueTwo(String[] value);
}
```

```
@Controller("myController")
public class MyClass
{
    @Create
    protected MyContextWrapper context;

	public void myMethod()
	{
```

```
                context.setValueTwo(new String[]{"Value One", "Value Two"});
                Window.alert(context.getValueTwo()[0]);
        }
}
```

If you pass null as argument for a setter method of a Context Wrapper object, it will remove that value from context. Example:

```
        context.setValueTwo(null);
```

Note that you can have more than one Context Wrapper. Different modules can use different wrappers if you want. However, the area where context information is written is unique. It's important to advice that you must initialize the context before using it. This can be done this way:

```
Screen.createContext();
```

If you call `Screen.createContext()` more than once, you will erase the context and create a new one.

Screen class has the following static methods to support Context management:

Method	Description
createContext	Initialize the context. It just needs to be called once in one of the modules that are sharing information.
clearContext	remove all context entries

Behind the scenes, Context is managed by a ContextHandler object. It is provided two implementations for this interface:

Handler	Description
CookieContextHandler	Use cookies to store values
TopContextHandler	Store values in a HashMap located on top of the first opener window

The default implementation is CookieContextHandler. However, each one has pros and cons.

TopContextHandler is faster and does not leave data after the application is closed. It is not possible to have dirty read, because the context area is created only when used for the first time. However, in a multi window application, the context data is lost if top window is closed.

CookieContextHandler never loses data if a window is closed. However, you must ensure that you call the method `Screen.createContext()` in your application to avoid dirty reads (cookies can contain old values from a previous execution).

If you desire, you can add the following lines to your module config file to change the default context behavior:

```
        <replace-with class="br.com.sysmap.crux.core.client.context.TopContextHandler">
                <when-type-assignable class="br.com.sysmap.crux.core.client.context.ContextHandler" />
        </replace-with>
```

3.2.4 Control History

Screen provides a simple mechanism to manage history. Using the static method `Screen.addToHistory(String token)` you can create a history token (exactly as GWT tokens). To handle the changes on history, you can add a handler to HistoryChanged events, as you can see in the next example:

```
        // To put a token on history
        Screen.addToHistory("link1Clicked");
```

```
        // To add a HistoryChanged Handler
        Screen.addHistoryChangedHandler(new addValueChangeHandler<String>(){
                public void onValueChange(ValueChangeEvent<String> event)
                {
                        Window.alert(event.getValue());
                }
        });
```

or Declaratively:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <crux:screen useController="clientHandler" onHistoryChanged='clientHandler.onHistoryChanged' />
   </body>
</html>
```

Note that you don't need here to add any iFrame to your host page, as in pure GWT.

3.2.5 Block and Unblock

You can block and unblock the user interaction with the page using the static methods `Screen.blockToUser()` e `Screen.unblockToUser()`. See the following example:

```
...
@Create
 protected TestServiceAsync service;

 public void helloWorld()
 {
      Screen.blockToUser();
      service.hello(new AsyncCallback<String>()
      {
              public void onSuccess(String s)
              {
                      Screen.unblockToUser();
                      Window.alert(s);
              }
              public void onFailure(Throwable e)
              {
                      Screen.unblockToUser();
                      Window.alert(e.getMessage());
              }
      });
 }
 ...
```

3.3 Managing Events

It's possible to add event handlers:

1. Declaratively.
2. Programmatically.

3.3.1 Add Event Declaratively

To add an event declaratively, you must create a Controller and give it a name.

An event declaration must follow the pattern `on<eventName> = "<controllerName>.<methodName>"`. For example:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
```

```
    </head>
    <body>
        <crux:screen onClose='clientHandler.onClose' useController="clientHandler" >
                <gwt:textBox id="myBox" />
                <gwt:button id="myButton" text="Hello" onClick="clientHandler.helloWorld" />
            </crux:screen>
    </body>
</html>
```

Note that you must also "import" your controller through the attribute `useController` of the screen's tag.

3.3.2 Add Event Programmatically

You can still use the default GWT mechanism to add handlers for events programmatically, like:

```
...
Button myButton = new Button();
myButton.addClickHandler(new ClickHandler(){
   public void onClick(ClickEvent event)
   {
       Window.alert("hello");
   }
});
...
```

The only point to observe here is that you need to put this code in some controller method that is called declaratively, unless you has overwritten the ***Crux*** EntryPoint and call it directly from there.

3.3.3 Controller

Controller classes are called to handle events.

To Create a Controller, you just create a simple java class with the `@Controller` annotation. That annotation has a value property to inform the name of the controller. That is the name used on pages to point to the controller.

See the following example:

```
@Controller("clientHandler")
public class MyController
{
   ... // event handlers here
}
```

Your controller can have a lot of methods to handle events. These methods must follow the conditions:

1. It must have public visibility;
2. It must have zero or one argument. If an argument is present, it must be a GwtEvent and this method only will be able to handle this type of events.
3. It must be marked with the annotation `@Expose`.

See the example:

```
@Controller("clientHandler")
public class MyController
{
   @Expose
   public void onClose(CloseEvent<Window> event)
   {
       // code here
   }

   @Expose
   public void helloWorld()
   {
       // code here
   }

   @Expose
   public void onClick(ClickEvent event)
   {
       // code here
   }

   protected void myMethod(String string)
   {
       // code here
   }
}
```

Note that the above controller contains a method that does not follow the conditions to be an event handler (`myMethod`). It can not be called declaratively.

3.3.3.1 The @Controller Annotation

The Controller annotation has the following properties:

Property	Required	Default Value	Description
value	yes	none	defines the name of the controller. Used inside pages to point to the controller
statefull	no	true	If true, one controller object is created and the same instance is used to handle all events. If false, a new instance is used for each new event
autoBind	no	true	If true, ValueObjects are automatically bound from screen widgets before the event occurs and bound back to screen widgets when the event handling terminates
lazy	no	true	If true, the controller object is built only when first called.
fragment	no	`<empty>`	You can inform a fragment identifier. The compiler will split your code grouping controllers by these identifiers.

3.3.3.2 The @Create Annotation

This annotation can be used to simplify the code for Controllers. It automatically creates an object (according with field type) and initializes the field with this value.

This creation is done by a call to GWT.create method, assuring that any generator eventually associated with the requested type will be called correctly.

See the following example:

```
@Controller("myNewController")
public class MyClass
{
    public static interface MyScreen extends ScreenWrapper
    {
        Button getMyButton();
        TextBox getMyBox();
    }

    @Create
    protected MyScreen myScreen;

    @Create
    protected MyContextWrapper context;

    @Create
```

```
    protected TestServiceAsync service;

    @Create
    protected MyControllerInvoker invoker;

    @Expose
    public void myMethod()
    {
        myScreen.getMyBox().setValue("Test");
        context.setValueTwo(new String[]{"Value One", "Value Two"});
    }
}
```

Note that `@Create` can handle service creation too, despite the fact that the variable type is not the same passed to `GWT.create` (in above example would be `TestService`).

The `@Create` annotation does more than simply creates an object. It also makes some initializations for the created object, depending on the field type (eg. parameter fields are loaded, etc.)

You must note, however, that to a field can be created through `@Create` annotation, that field must has public or protected visibility or has a public getter and setter methods.

3.3.3.3 The @Expose Annotation

The Expose annotation has the following properties:

Property	Required	Default Value	Description
allowMultipleCalls	no	false	If true, allow user to dispatch more than one event at time. If false, when an event handler is called, the screen is blocked until the method finish.

3.3.3.4 Value Binding

Crux provides a mechanism to help you to automatically bind values between screen widgets and data objects.

You can create a value object and annotate it with `@ValueObject` annotation. Doing it, you allows ***Crux*** to populate an object of this type with values present on screen widgets before run the event handler methods. After method execution, the screen is also updated with any change in these objects.

See the following example:

```
@ValueObject
public class Person
{
        private String name;
        private String phone;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getPhone() {
                return phone;
        }
        public void setPhone(String phone) {
                this.phone = phone;
        }
}
```

```
@Controller("myController")
public class MyClass
{
    @Create
    protected Person person;

    @Expose
    public void myMethod()
    {
        Window.alert(person.getName());
        person.setPhone("1234-5678");
    }
}
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt">
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <crux:screen useController="myController" >
               <gwt:textBox id="name" />
               <gwt:textBox id="phone" />
               <gwt:button id="myButton" text="Hello" onClick="myController.myMethod" />
           </crux:screen>
   </body>
</html>
```

In the above example, the value of the "name" textBox on page will be bound to field "name" of the Person object created by controller (the same is true to "phone"). After the handler execution, the changes made in the value object will be reflected on page.

If want, you can use the `@ScreenBind` annotation on value object field to inform ***Crux*** which widget will be bound to this field. The above example can be changed to:

```
@ValueObject
public class Person
{
        @ScreenBind("person.name")
        private String name;

        @ScreenBind("person.phone")
        private String phone;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getPhone() {
                return phone;
        }
        public void setPhone(String phone) {
                this.phone = phone;
        }
}
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
```

```
    <body>
        <crux:screen useController="myController" >
                <gwt:textBox id="person.name" />
                <gwt:textBox id="person.phone" />
                <gwt:button id="myButton" text="Hello" onClick="myController.myMethod" />
            </crux:screen>
    </body>
</html>
```

You can also control which fields of a value object must be bound to some widget screen. `@ValueObject` annotation has a boolean property called `bindWidgetByFieldName` (default to true). Setting this value to false make ***Crux*** to does not bind all value object fields to widgets automatically. If you set this, you must specify for each field, the name of the widget that it will be bound (through `@ScreenBind` annotation).

See the following example:

```
@ValueObject(bindWidgetByFieldName=false)
public class Person
{
        @ScreenBind("person.name")
        private String name;

        @ScreenBind
        private String phone; //will be bound to "phone" widget

        private String address; // will not be bound.

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getPhone() {
                return phone;
        }
        public void setPhone(String phone) {
                this.phone = phone;
        }
        public String getAddress() {
                return address;
        }
        public void setAddress(String address) {
                this.address = address;
        }
}
```

Any field in a value object can be bound to a widget if it:

- is a primitive type (or a primitive wrapper);
- is a CharSequence type (String, StringBuilder, StringBuffer, etc);
- is a Date type (java.util.Date, java.sql.Date, java.sql.Timestamp, etc);
- is an Enum type;
- is any type annotated with `@ValueObject` annotation;
- has public or protected visibility or has public getter and setter methods.

The following code shows more examples:

```
@ValueObject
public class Person
{
        private String name;
        private String phone;
        private Date birth;
        private Address address;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getPhone() {
                return phone;
        }
        public void setPhone(String phone) {
                this.phone = phone;
        }
        public Date getBirth() {
                return birth;
        }
        public void setBirth(String birth) {
                this.birth = birth;
        }
        public Address getAddress() {
                return address;
        }
        public void setAddress(Address address) {
                this.address = address;
        }
}
```

```
@ValueObject
public class Address
{
        private String street;

        public String getStreet() {
                return street;
        }
        public void setStreet(String street) {
                this.street = street;
        }
}
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <crux:screen useController="myController" >
                <gwt:textBox id="name" />
                <gwt:dateBox id="birth" />
                <gwt:textBox id="street" />
                <gwt:button id="myButton" text="Hello" onClick="myController.myMethod" />
            </crux:screen>
   </body>
</html>
```

In the above example, the field `phone` will not be bound to any widget, once there is not any widget with id `phone`. Another important point is that each field only can be bound to widgets that are able to return values of the same type of the field.

The field `birth` only can be bound to widgets that implements `HasValue<Date>` (like `DateBox`) or `HasFormatter` and is associated with a formatter that returns `Date` objects. See the Formatters section to more info about formatters).

If you want to disable the automatic value binding mechanism to a specific controller, you can set the `@Controller` property `autoBind` to false. You can, later, control the value object and screen updates through methods `Screen.updateScreen(controller)` and

`Screen.updateController(controller)`. See the following example:

```
@Controller(value="myController", autoBind=false)
public class MyClass
{
    @Create
    protected Person person;

    @Expose
    public void myMethod()
    {
        Screen.updateController(this);
        Window.alert(person.getName());
        person.setPhone("1234-5678");
        Screen.updateScreen(this);
    }
}
```

3.3.3.5 Using Controllers on Screen

To inform that a controller will be used on a screen, you must explicitly "import" it using the `useController` attribute.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <crux:screen useController="myController, myOtherController" >
           </crux:screen>
   </body>
</html>
```

It is necessary for performance reasons.

However, if you want that a controller be imported in every screen you have, you can put an annotation `@Global` in the Controller class.

```
@Global
@Controller("myController")
public class MyClass
{
       // code here
}
```

That will allow these controller methods to be called even if no `useController` declaration is present on your screen.

3.3.3.6 Comunication Between Controllers

In section Communication Between Screens we show how you can, from a controller, to invoke methods on another controller. Now we will see the details associated with the methods of those controllers.

The first point is that you must explicitly inform ***Crux*** that an event handler can be called from out of the current page (by another document). To do this, you must put an annotation `@ExposeOutOfModule` on the method you want to expose.

```
@Controller("myController")
public class MyClass
{
    @ExposeOutOfModule
    public void myMethod()
    {
       // code here
    }
}
```

Now, other pages can call this method through `Screen.invokeControllerOn<?>()` methods.

For calls to `Screen.invokeControllerOnSelf()` the handler does not need to be exposed to out of module. You can call a handler exposed only with `@Expose` annotation.

So we can talk about two kinds of controller methods invocation. One remote and one local. Remote calls are those made to another pages (top, parent, opener, etc) and local are made to the same page (self).

An event handler for calls made through screen invoker methods, can receive an `InvokeControllerEvent` parameter. This class provides a method to access values passed as argument in method invocation.

See the example:

```
...
Window.alert(Screen.invokeControllerOnTop("myController.hello","Thiago"));
Window.alert(Screen.invokeControllerOnSelf("myController.helloLocal","Thiago"));
Window.alert(Screen.invokeControllerOnParent("myController.parametersExample",new Object[]{new Integer(123), "Thiago", new Boolean(true)}));
...
```

or

```
...
Window.alert(myControllerInvoker.helloOnTop("Thiago"));
Window.alert(myControllerInvoker.helloLocalOnSelf("Thiago"));
Window.alert(myControllerInvoker.parametersExampleOnParent(1234, "Thiago", true));
...
```

```
@Controller("myController")
public class MyClass
{
    @ExposeOutOfModule
    public String hello(InvokeControllerEvent event)
    {
       String param = event.getParameter(String.class);
       return "Hello "+param;
    }

    @Expose
    public String helloLocal(InvokeControllerEvent event)
    {
       String param = event.getParameter(String.class);
       return "Hello "+param;
    }

    @ExposeOutOfModule
    public String parametersExample(InvokeControllerEvent event)
    {
       if (event.getParameterCount() == 3)
       {
          int param0 = event.getParameter(0, Integer.class);
          String param1 = event.getParameter(1, String.class);
          boolean param2 = event.getParameter(2, Boolean.class);
          return "Hello "+param0;
       }
       return null;
    }
}
```

3.3.3.6.1 Parameter types

The following types are allowed as arguments to a remote call to a controller method:

- Primitive types (or a primitive wrapper);
- CharSequence types (String, StringBuilder, StringBuffer, etc);
- Date types(java.util.Date, java.sql.Date, java.sql.Timestamp, etc);
- Enum types;
- Any type that implements the interface `CruxSerializable`.
- Arrays of any of those types above;

So, if you need to pass a custom object to another page, your object class must implements the interface CruxSerializable. This interface forces your type to implement a serialize and deserialize methods.

For local calls, any type is allowed, once no serialization is needed. If you define a custom serializable type, you must inform ***Crux*** explicitly that you want to use such type in your module. You can do it declaratively using `useSerializable` attribute on screen tag, or through the method `ModuleComunicationSerializer.registerCruxSerializable`.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <crux:screen useController="myController" useSerializable="mySerializableType">
           </crux:screen>
   </body>
</html>
```

```
@SerializableName("mySerializableType")
public class MySerializableType implements CruxSerializable
{
  private String field1;
  private Integer field2;

  ... // getter and setters

   public String serialize() {
      try  {
         Object[] values =  new Object[]{field1, field2};
         return Screen.getCruxSerializer().serialize(values);
      } catch (ModuleComunicationException e) {
         ...
      }
      return null;
  }

  public Object deserialize(String serializedData) {
    try {
       Object[] des = (Object[]) Screen.getCruxSerializer().deserialize(serializedData);
       MySerializableType dto =  new MySerializableType();
       dto.setField1((String) des[0]);
       dto.setField2((Integer) des[1]);
        return dto;
    }catch (ModuleComunicationException e) {
                          e.printStackTrace();
    }
    return null;
  }
}
```

or

```
   ModuleComunicationSerializer.registerCruxSerializable(MySerializableType.class.getName(), new MySerializableType());
```

The annotation `@SerializableName` could be used to simplify the name you write on screen tag.

3.3.3.7 Validation

Crux supports declaration of validators for a controller handler method. A Validator method is called before the handler method itself. If it runs without problem, the handler is called. If it throws any exception, the handler execution is aborted and a message is reported to the user through the ***Crux*** Error Handlers.

To declare a validator method to a given handler method, you just need to use the `@Validate` annotation :

```
@Controller("clientHandler")
public class MyController
{

   @Validate("myValidationMethod")
   @Expose
   public void onClose(CloseEvent<Window> event)
   {
       // code here
   }

   protected void myValidationMethod(CloseEvent<Window> event) throws ValidateException
   {
       // code here
   }
}
```

If no value is passed to the `@Validate` annotation, ***Crux*** tries to find a method called `validate<methodName>`.

```
@Controller("clientHandler")
public class MyController
{

   @Validate
   @Expose
   public void onClose(CloseEvent<Window> event)
   {
       // code here
   }

   protected void validateOnClose() throws ValidateException
   {
       // code here
   }
}
```

Note that the validate method can receive a parameter of the same type as the main method parameter type (as in the first example) or no parameter (as in the second example).

3.3.3.8 Parameters Binding

Crux provides a mechanism to help you to automatically bind values between window parameters and data objects. This can be done using the annotations `@Parameter` and `@ParameterObject` on controller fields and DTO classes.

You can annotate a controller field with the annotation `@Parameter` as in the following example:

For the following URL

```
        http://myhost.com/myapp/mymodule/mypage.html?person=Thiago&parameterName=123
```

And Controller

```
@Controller("myController")
public class MyClass
{
```

```
    @Parameter
    protected String person;

    @Parameter(value="parameterName", required=true)
    protected int field;

    @Expose
    public void myMethod()
    {
        Window.alert(person);
        Window.alert(Integer.toString(field));
    }
}
```

In the above example, the value of the "person" parameter on window URL will be bound to field "person" of the controller (the same is true to "field").

The `@Parameter` annotation has two fields:

Property	Default Value	Description
value	`empty`	defines the name of the parameter. If not present, the field name is used
required	false	If true, a validation is done to ensure that the parameter is present in the URL.

If a validation error occur while binding the parameter, a message is reported to the user through the Crux Error Handlers. A validation error can occur caused by a type conversion error or by a missing required parameter.

Another way to bind parameters is to create an object and annotate it with `@ParameterObject` annotation, exactly as you do with Value Objects. Doing it, you allows ***Crux*** to populate an object of this type with values present on window url parameters.

See the following example:

```
@ParameterObject
public class Parameters
{
        private String person;
        private int field;

        public String getPerson() {
                return person;
        }
        public void setPerson(String person) {
                this.person = person;
        }
        public int getField() {
                return field;
        }
        public void setField(int field) {
                this.field = field;
        }
}
```

```
@Controller("myController")
public class MyClass
{
    @Create
    protected Parameters parameters;

    @Expose
    public void myMethod()
    {
        Window.alert(parameters.getPerson());
        Window.alert(Integer.toString(parameters.getField()));
    }
}
```

If want, you can use the `@Parameter` annotation on parameter object field to inform ***Crux*** which parameter will be bound to this field. The above example can be changed to:

```
@ParameterObject
public class Parameters
{
        @Parameter("personName")
        private String person;

        @Parameter("fieldParameter", required=true)
        private int field;

        public String getPerson() {
                return person;
        }
        public void setPerson(String person) {
                this.person = person;
        }
        public int getField() {
                return field;
        }
        public void setField(int field) {
                this.field = field;
        }
}
```

You can also control which fields of a parameter object must be bound to some window parameter. `@ParameterObject` annotation has a boolean property called `bindParameterByFieldName` (default to true). Setting this value to false make ***Crux*** to does not bind all parameter object fields to window parameter automatically. If you set this, you must specify for each field, the name of the parameter that it will be bound (through `@Parameter` annotation).

See the following example:

```
@ParameterObject(bindParameterByFieldName=false)
public class Parameters
{
        @Parameter("personName")
        private String person;

        @Parameter("fieldParameter", required=true)
        private int field;

        private int field2; // not bound

        public String getPerson() {
                return person;
        }
        public void setPerson(String person) {
                this.person = person;
        }
        public int getField() {
                return field;
        }
        public void setField(int field) {
                this.field = field;
        }
        public int getField2() {
                return field2;
        }
        public void setField2(int field2) {
                this.field2 = field2;
```

```
        }
}
```

Any field in a parameter object can be bound if it:

- is a primitive type (or a primitive wrapper);
- is a CharSequence type (String, StringBuilder, StringBuffer, etc);
- is a Date type (java.util.Date, java.sql.Date, java.sql.Timestamp, etc);
- is an Enum type;
- is any type annotated with `@ParameterObject` annotation;
- has public or protected visibility or has public getter and setter methods.

3.4 I18N

Crux supports i18n for widgets created declaratively. The native GWT mechanism is still valid.

You can use the following pattern to tell ***Crux*** that you want to use a GWT message or a constant value in a widget tag declaration:

```
   "${<messageResource>.<messageEntry>}"
```

For example, suppose the following messages interface:

```
public interface MyMessages extends Messages
{
        @DefaultMessage("my message")
        String myMessage();
}
```

And the crux page that uses it:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
       <gwt:label id="label" text="${myMessages.myMessage}" />
   </body>
</html>
```

In the above example, the message resource name is derived from interface name. If you want to change this value, you can use the annotation `@Name` in your messages interface.

See the example:

```
@Name("msg")
public interface MyConstants extends Constants
{
        @DefaultStringValue("my message")
        String myMessage();
}
```

```
   ...
       <bas:label id="label" text="${msg.myMessage}" />
   ...
```

3.5 Client-Server Communication

You can use GWT RPC and JSON support to communicate with server. ***Crux***, however, adds some few features to pure GWT RPC mechanisms to turn it easier.

With ***Crux***, you can use a *Front Controller*. The use of this *Front Controller* allows you to make just one mapping in your web.xml file.

In other words, you don't need to add a new servlet declaration on your GWT module definition nether on web.xml for each new service declared. Other improvement is the use of the @Create annotation. It will create the service object and set its entry point name to point to the *Front Controller* automatically. The following example shows all this features together:

```
package crux.examples.client.remote;

import com.google.gwt.user.client.rpc.RemoteService;

public interface GreetingService extends RemoteService
{
        public String getHelloMessage(String name);
}
```

```
public class MyController {
        @Create
        protected GreetingServiceAsync service;

        @Expose
        public void sayHello() {
                service.getHelloMessage("Thiago", new AsyncCallbackAdapter<String>(this){
                                @Override
                                public void onComplete(String result){
                                   Window.alert(result);
                                }
                        }
                );
        }
}
```

Note that the service interface does not use the annotation RemoteServiceRelativePath. It will assume the mapping to the Front Controller. If that annotation was present, ***Crux*** would use it to set the entry point name.

Another point to observe in the above example is the use of the abstract class `AsyncCallbackAdapter` in the place of GWT `AsyncCallback` interface.

That class does the following:

1. Implements a default error handler that will delegate to Crux Error Handler any error received.
2. If controller autoBind for ValueObjects is enabled, it will automatically update screen with value object properties after process the `onComplete()` method.

So, the two following approach is equivalents:

```
@Controller(value="myController", autoBind=true)
public class MyController {
        @Create
        protected GreetingServiceAsync service;

        @Create
        protected Person aValueObject;

        @Expose
        public void sayHello() {
                service.getHelloMessage("Thiago", new AsyncCallbackAdapter<String>(this){
                                @Override
                                public void onComplete(String result){
                                   aValueObject.setName("Thiago");
                                }
                        }
                );
        }
}
```

```
@Controller(value="myController", autoBind=false)
public class MyController {
        @Create
        protected GreetingServiceAsync service;

        @Create
        protected Person aValueObject;

        @Expose
        public void sayHello() {
                Screen.updateController(this);
                service.getHelloMessage("Thiago", new AsyncCallback<String>(this){
                                public void onSuccess(String result){
                                   aValueObject.setName("Thiago");
                                   Screen.updateScreen(this);
                                }
                                public void onFailure(Throwable e){
                                     Crux.getErrorHandler().handleError(e.getLocalizedMessage(), e);
                                     Screen.updateScreen(this);
                                }
                        }
                );
        }
}
```

3.5.1 Server Sensitive Methods

Crux supports the Synchronizer Token pattern for sensitive methods protection. This pattern helps to avoid the duplicated request problem and CSRF attacks.

To inform ***Crux*** that a server method is sensitive, you just need to put the annotation `@UseSynchronizerToken` on the service interface method.

See the following example:

```
package crux.examples.client.remote;

public interface GreetingService extends RemoteService
{
   @UseSynchronizerToken
   public String getHelloMessage(String name);
}
```

This annotation accepts the following attributes:

Attribute	Type	Default	Description
notifyCallsWhenProcessing	boolean	true	If this property is true, when the user tries to send a duplicated request, an informative message is showed. To change the message, use the property `methodIsAlreadyBeingProcessed` on ClientMessages.properties file.
blocksUserInteraction	boolean	true	If this property is true, when a request to a sensitive method is fired, the screen became blocked to user.

3.6 Handling Errors

Crux provides two basic interfaces for client errors reporting.

- `ErrorHandler`
- `ValidationErrorHandler`

Those interfaces are used always occurs an error that needs to be reported to user (or to developer). The interface `ErrorHandler` is called to report errors in application code (bad use of the framework, or an uncaught exception) and `ValidationErrorHandler` is called to report errors caused by client miss using the application (validations on the screen before perform an action).

Crux provides a default `Errorhandler` that implements both interfaces and, for the both types of errors:

- Logs in GWT console all exceptions received.
- Shows the messages using the `Window.alert()` method.

If you want to change the default error handling class, you can specify in your module file:

```
        <!-- Specify the implementation to ErrorHandler.-->
        <replace-with class="YourErrorHandlerClass">
                <when-type-assignable class="br.com.sysmap.crux.core.client.errors.ErrorHandler" />
        </replace-with>

        <!-- Specify the implementation to ValidationErrorHandler.-->
        <replace-with class="YourValidationErrorHandlerClass">
                <when-type-assignable class="br.com.sysmap.crux.core.client.errors.ValidationErrorHandler" />
        </replace-with>
```

3.7 Formatters

Formatters can be used to provide widgets, the capabilities:

- Input/Output formatting/unformatting;
- Input masking (masks can be defined with regular expressions);
- Data conversions to/from string.

A widget must implements the interface `HasFormatter` to be associated with a formatter and formatters can be defined implementing the interface `Formatter`.

Here you can see how a formatter could be used:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:c="http://www.sysmap.com.br/crux"
      xmlns:crux="http://www.sysmap.com.br/crux/widgets" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
      <c:screen useFormatter="phone, date" />
      <crux:maskedTextBox id="maskedTextBox" formatter="phone" width="90"/>
      <crux:maskedTextBox id="dateMaskedTextBox" formatter="date" width="90"/>
   </body>
</html>
```

The widget `MaskedTextBox` is delivered with ***Crux*** default distribution. It is like a GWT TextBox, but provides formatting support.

The following example shows how a custom formatter could be defined:

```
@FormatterName("phone")
public class PhoneFormatter implements Formatter
{
        public String format(Object input)
        {
                if (input == null || !(input instanceof String) || ((String)input).length() != 10)
                {
                        return "";
                }

                String strInput = (String) input;

                return "("+strInput.substring(0,2)+")"+strInput.substring(2,6)+"-"+strInput.substring(6);
        }

        public Object unformat(String input) throws InvalidFormatException
```

```
        {
                if (input == null || !(input instanceof String) || ((String)input).length() != 13)
                {
                        return "";
                }
                String inputStr = (String)input;
                inputStr = inputStr.substring(1,3)+inputStr.substring(4,8)+inputStr.substring(9,13);
                return inputStr;
        }
}
```

The above code, specifies a formatter that will present its associated widget content (a phone number) in the format: `(99)9999-9999`. That formatter does not provide input masking feature. To create a version of the formatter with this feature, you must implements the interface `MaskedFormatter`

To create masked formatters for `MaskedTextBox` widgets, an abstract class that already implements `MaskedFormatter` can be used. The previous formatter example could be re-written as:

```
@FormatterName("phone")
public class PhoneFormatter extends MaskedTextBoxBaseFormatter{
        @Override
        protected String getMask()      {
                return "(99)9999-9999";
        }

        public String format(Object input){
                if (input == null || !(input instanceof String) || ((String)input).length() != 10)      {
                        return "";
                }

                String strInput = (String) input;

                return "("+strInput.substring(0,2)+")"+strInput.substring(2,6)+"-"+strInput.substring(6);
        }

        public Object unformat(String input) throws InvalidFormatException{
                if (input == null || !(input instanceof String) || ((String)input).length() != 13)      {
                        return "";
                }
                String inputStr = (String)input;
                inputStr = inputStr.substring(1,3)+inputStr.substring(4,8)+inputStr.substring(9,13);
                return inputStr;
        }
}
```

Another example:

```
@FormatterName("date")
public static class DateFormatter extends MaskedTextBoxBaseFormatter {

    DateTimeFormat format = DateTimeFormat.getFormat("MM/dd/yyyy");

    protected String getMask(){
        return "99/99/9999";
    }

    public Object unformat(String input){
        if (input == null || input.length() != 10){
            return null;
        }

        return format.parse(input);
    }

    public String format(Object input) throws InvalidFormatException {
        if(input == null){
            return "";
        }
        if (!(input instanceof Date)){
            throw new InvalidFormatException();
        }

        return format.format((Date) input);
    }
}
```

Note that the `MaskedFormatter` methods (`applyMask` and `removeMask`) are already implemented by the abstract class. Only the `getMask` method must be implemented to specify the pattern used to build the mask.

TODO: *A basic set of common formatters is being created and will be available as crux-formatters.jar that will be included in the distribution soon*.

3.8 Data Sources

DataSources are objects capable of providing a set of data to widgets that implement `HasDataSource` interface. DataSources support features like pagination, data sorting and editing.

Crux provides a wide range of different DataSources that can be classified by the following criteria:

1. How them **present the data**
    1. **Paged** - PagedDataSources can divide the data into pages
    2. **Scrollable** - This kind od DataSource handles all data in a single page
2. How they **fetch the data**
    1. **Local** - This kind of DataSource can load data once and keep it locally on user's brownser, so it can be paged, sorted or edited locally.
    2. **Remote** - RemoteDatasources load data on demand, as widgets request them.
3. How them **restrict access** to data
    1. **Read Only** - Data can not be modified or selected
    2. **Editable** - Data can be modified and selected
4. How data is **structured** inside the DataSource
    1. **Records** - Data is organized as columns of records
    2. **Value Objects** - Data is organized as a list of objects (Value Objects) (see BindableDataSource)

To create a DataSource, you can extend one of the abstract DataSource classes provided by ***Crux***. The class you should choose depends on which categories (between the exposed above) your DataSource will belong.

For a complete guide about DataSources (including the complete list of basic DataSource classes), consult the following tutorial.

3.9 Templates

Templates in ***Crux*** are parameterizable XML files that can be used for:

1. Create simple components in a declarative way;
2. Create smart fragments that can be used to compose greater pages;
3. Define reusable layout pages.

A template must be defined in a file with the extension `.template.xml` and can be placed anywhere under your classpath (even inside a jar file).Template files must follow the schema `http://www.sysmap.com.br/templates`.

Sections Creating a Simple Component, Creating a Smart Fragment and Defining a Reusable Layout shows examples of templates usage.

After creating a template file, you must run the Schema Generator again. It will generate a XSD file for your template, enabling auto-completion when using it in a page.

A template can be defined to receive parameters and include sub-sections, as showed in the following examples:

3.9.1 Examples

3.9.1.1 Creating a Simple Component

The file `labeledBox.template.xml` defines a template for a simple labeled box component:

```
<t:template xmlns="http://www.w3.org/1999/xhtml"
          xmlns:t="http://www.sysmap.com.br/templates"
          xmlns:gwt="http://www.sysmap.com.br/crux/gwt"
          library="custom">

   <gwt:horizontalPanel id="#{id}.hPanel" >
      <gwt:label id="#{id}.label" text="#{label}:" />
      <gwt:textBox id="#{id}" value="#{value}" />
   </gwt:horizontalPanel>
</t:template>
```

Then, you can use it on any crux page:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:c="http://www.sysmap.com.br/templates/custom" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
      <c:labeledBox id="personName" label="Name" value="Type your name here..." />
   </body>
</html>
```

3.9.1.2 Creating a Smart Fragment

The File `userInfo.template.xml` defines a simple header that can be used on different pages:

```
<t:template xmlns="http://www.w3.org/1999/xhtml"
      xmlns:t="http://www.sysmap.com.br/templates"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt"
      library="custom" useController="userController">
   <b:HTMLPanel id="userPanel" onLoadWidget="userController.loadUserInfo" >
      <table>
             <tr>
                <td>Login:</td>
                <td><gwt:label id="login" /></td>
             </tr>
             <tr>
                <td>Name:</td>
                <td><gwt:label id="name" /></td>
             </tr>
      </table>
   </b:HTMLPanel>
</t:template>
```

Then, you can use it on any crux page:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt"
      xmlns:c="http://www.sysmap.com.br/templates/custom" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
      <c:userInfo />
      ...
   </body>
</html>
```

3.9.1.3 Defining a Reusable Layout

The file `pageLayout.template.xml` defines a common layout that has a menu located on the left and a place to insert the page body:

```
<t:template xmlns="http://www.w3.org/1999/xhtml"
      xmlns:t="http://www.sysmap.com.br/templates"
      xmlns:crux="http://www.sysmap.com.br/crux/widgets"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt"
      xmlns:c="http://www.sysmap.com.br/templates/custom"
          library="custom">

   <gwt:dockPanel id="centeringPanel" width="100%" height="100%">
      <gwt:cell direction="north" height="70" verticalAlignment="top">
          <c:userInfo />
      </gwt:cell>
      <gwt:cell direction="south">
         <gwt:dockPanel id="menuTabsDividerPannel">
            <gwt:cell direction="west">
               <crux:stackMenu id="menu" onLoadWidget="#{onLoadMenu}" ></crux:stackMenu>
            </gwt:cell>
            <gwt:cell direction="east">
               <gwt:HTMLPanel id="hPanel">
                  <t:section name="body" />
               </gwt:HTMLPanel>
            </gwt:cell>
         </gwt:dockPanel>
      </gwt:cell>
   </gwt:dockPanel>
</t:template>
```

Then, you can use it on any crux page:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:crux="http://www.sysmap.com.br/crux"
      xmlns:gwt="http://www.sysmap.com.br/crux/gwt"
      xmlns:c="http://www.sysmap.com.br/templates/custom" >
   <head>
       <script language="javascript" src="cruxtest/cruxtest.nocache.js"></script>
   </head>
   <body>
      <crux:screen useController="myController" />
      <c:pageLayout onLoadMenu="myController.loadMenuItems">
         <c:body>
            <!-- Body comes here -->
            <gwt:label id="test" text="Hello World!!" />
         </c:body>
      </c:pageLayout>
   </body>
</html>
```

Note that the above template reuses another template (userInfo) defined in previous section.

3.9.2 Templates attributes and children

According with template.xsd file, the tag `<template>` declare the following attributes:

Attribute	Description
library	Required attribute that inform the library into which this template will be included. This will define the name of the xsd file where the template definition will be put and the namespace associated with this file. (`http://www.sysmap.com.br/templates/<library>`)
useController	Adds controllers to screen controller list

useSerializable	Adds serializables to screen serializable list
useFormatter	Adds formatters to screen formatter list
useDataSource	Adds datasources to screen datasource list

As showed in the previous examples, you can create a template that receive attributes and children.

- To define an attribute for your template, just write the attribute in the form `#{attributeName}` wherever you want to apply the attribute value.
- To add a child to your template, use the tag `section` where you want to create a placeholder in your template. That tag will be replaced by the content of the child declared on the page that uses the template. (See the example showed on section Defining a Reusable Layout).

3.10 Running With a Diferent Server

To run your ***Crux*** application under DevMode with a different server, you must follow all the steps described at the GWT documentation.

In adition to these steps, you must add the follow JVM argument to your application server: `-DCrux.dev=true`.

Note that this parameter just need to be inserted when runnig the server with development purposes. In production, you don't need any aditional configuration.

4 Coding Server Side

4.1 Writing Server Services

As wed said in section Client-Server Communication, ***Crux*** supports the GWT RPC mechanism with some few features to turn it easier.

At server side, the main difference for GWT is that your service implementation class does not need to extend `RemoteServiceServlet`. It just needs to implement the service interface.

```
public interface GreetingService extends RemoteService{
        public String getHelloMessage(String name);
}
```

```
public class GreetingServiceImpl implements GreetingService{
   public String getHelloMessage(String name){
      return "Server says: Hello, " + name + "!'";
   }
}
```

To find out which implementation will be used for a given service interface, ***Crux*** will search (using javassist) for classes that implements that interface and use the first one found.

This behavior can be changed, as showed in section serviceFactory.

If your service class needs to access the request, response or session, it can implements the interfaces RequestAware, ResponseAware or SessionAware, as in the following example:

```
public class GreetingServiceImpl implements GreetingService, RequestAware, ResponseAware{
   private HttpServletRequest request;
   private HttpServletResponse response;

   public void setRequest(HttpServletRequest request){
      this.request = request;
   }
   public void setResponse(HttpServletResponse response){
      this.response = response;
   }
   ...
}
```

4.2 I18N

Crux provides to server classes a support very similar to the GWT I18N support.

You can create interfaces and uses the annotation `@br.com.sysmap.crux.core.i18n.DefaultServerMessage` exactly as you do at client side.

The main difference is that your interface does not need to extend any other interface and you use the factory method `MessagesFactory.getMessages(<interfaceClass>)` in the place of `GWT.create()`.

See the following example:

```
public interface ServerMessages
{
        @DefaultServerMessage("My server message: {0}.")
        String myServerMessage(String message);
}
```

```
public MyServerClass
{
   private static ServerMessages messages = MessagesFactory.getMessages(ServerMessages.class);

   public void method()
   {
      System.out.println(messages.myServerMessage("test"));
   }
}
```

You can create a resource file called `ServerMessages` and put it under your application classpath to change messages for a specific locale. Example:

```
(file: ServerMessages_pt_BR.properties)
myServerMessage=Minha mensagem no servidor: {0}.
```

That mechanism exposed will work properly for all your classes that are called by a service. ***Crux*** will resolve locale problems in its FrontController, before delegate the application control to your service implementation.

However, if you plan to access i18n messages in classes called by a filter (that executes before the ***Crux*** *FrontController* Servlet), you need to configure a filter in your web.xml file. Read the section Web.xml to see how to do this.

4.3 Setup

To use ***Crux*** in your application, first of all you will need following files:

- In production time, inside the `WEB-INF/lib` folder:
  - commons-logging.jar
  - crux-core.jar
  - gwt-servlet.jar
- In development time:
  - addition to the previous files, inside the `WEB-INF/lib` folder:
    - crux-widgets.jar
    - crux-gwt-widgets.jar
    - crux-scannotation.jar
    - javassist.jar
    - saxon9.jar
    - saxon9-dom.jar
  - in any folder, since it is visible for the GWT Hosted Mode Console and for ANT tasks:
    - crux-compiler.jar
    - gwt-dev.jar (platform dependent)
    - gwt-ll.dll (platform dependent)
    - gwt-module.dtd

- gwt-user.jar

Using the Project Generator you will get a ready to use project structure.

4.3.1 Web.xml

To setup the ***Crux*** *Front Controller*, showed in previous section, you must add the following lines to your web.xml file:

```
<servlet>
        <servlet-name>remoteServiceServlet</servlet-name>
        <servlet-class>
                br.com.sysmap.crux.core.server.dispatch.RemoteServiceServlet
        </servlet-class>
</servlet>

<servlet-mapping>
        <servlet-name>remoteServiceServlet</servlet-name>
        <url-pattern>*.rpc</url-pattern>
</servlet-mapping>
```

There are some other configurations you will need to do to run your application in development environment:

```
<filter>
        <display-name>CruxFilter</display-name>
        <filter-name>CruxFilter</filter-name>
        <filter-class>br.com.sysmap.crux.core.server.CruxFilter</filter-class>
</filter>
<filter-mapping>
        <filter-name>CruxFilter</filter-name>
        <url-pattern>*.html</url-pattern>
</filter-mapping>
```

The above lines is needed by ***Crux*** Generators to find out which module is being compiled. This is used for better performance (see this for more information). Because that information is used only for compilation, it just need to be present in development environment.

You will need too:

```
<filter>
        <display-name>DeclarativeUIFilter</display-name>
        <filter-name>DeclarativeUIFilter</filter-name>
        <filter-class>br.com.sysmap.crux.core.declarativeui.filter.DeclarativeUIFilter</filter-class>
        <init-param>
                <param-name>outputCharset</param-name>
                <param-value>ISO-8859-1</param-value>
        </init-param>
</filter>
<filter-mapping>
        <filter-name>DeclarativeUIFilter</filter-name>
        <url-pattern>*.html</url-pattern>
</filter-mapping>
```

This filter is used to transform your *.crux.xml* files in pure *html* files. This process is only done in development. In a production environment, your application can access the generated version of the page directly.

The two above filters, which is used only for development, does nothing if used in a production environment, generating no overhead. They are removed by the ant task generated for deploy your project (if you are using the Project Generator_).

You can add an optional listener called InitializerListener to initialize some ***Crux*** resources to turn the first call to application faster.

```
<listener>
   <listener-class>br.com.sysmap.crux.core.server.InitializerListener</listener-class>
</listener>
```

If you need I18N before the ***Crux*** *FrontController* Servlet, you have to put these lines too:

```
<filter>
        <display-name>I18NFilter</display-name>
        <filter-name>I18NFilter</filter-name>
        <filter-class>br.com.sysmap.crux.core.i18n.I18NFilter</filter-class>
</filter>
<filter-mapping>
        <filter-name>I18NFilter</filter-name>
        <url-pattern>*.rpc</url-pattern>
</filter-mapping>
```

4.3.2 Crux.properties

Crux provides some configuration options that can modify behaviors on the framework.

Those options can be informed through:

1. A command line argument to JVM, adding a System property like: `-DCrux.<propertyName>=<propertyValue>`. Eg: `-DCrux.wrapSiblingWidgets=false`.
2. A file called `Crux.properties` that can be put (optionally) in any place under your classpath.

If you use both ways showed, the command line has preference.

The following properties can be set:

Property	Description	Default Value
wrapSiblingWidgets	If false, widgets created declaratively under the same parent has no guarantee of order	true
localeResolver	Class used by ***Crux*** to resolve locale for user at the server side	br.com.sysmap.crux.core.i18n.LocaleResolverImpl
screenResourceResolver	Class used by ***Crux*** to retrieve the screen page files	br.com.sysmap.crux.core.rebind.screen.ScreenResourceResolverImpl
classPathResolver	Class used by ***Crux*** to resolve classpath files	br.com.sysmap.crux.core.server.classpath.ClassPathResolverImpl
serviceFactory	Class used by ***Crux*** to instantiate controller classes	br.com.sysmap.crux.core.server.dispatch.ServiceFactoryImpl
allowAutoBindWithNonDeclarativeWidgets	Allow AutoBind feature to be used with widgets that were not created by the declarative engine	true
enableChildrenWindowsDebug	If true, propagates the GWT debug parameters to other windows opened while application runs under the DevMode	true
enableWebRootScannerCache	If true, uses a cache for the resources scanner	true
enableHotDeploymentForWebDirs	If true, ***Crux*** enables hot deployment for all resources on web dir, including templates	true
enableHotDeploymentForWidgetFactories	If true, **Crux** *supports hot deployment when new widgets types are used on screens.*	true

4.3.2.1 wrapSiblingWidgets

To enable ***Crux*** to do very significant improvements in screen creation performance, ***Crux*** will need that all of your widgets are orphan child of their parents.

The property `wrapSiblingWidgets` will automatically create an empty `<span>` tag around each widget that does not follow this restriction.

See the following example:

```
...
<body>
    <crux:textBox id="box1" />
    <crux:textBox id="box2" />
    <div>
            <crux:textBox id="box3" />
    </div>
</body>
...
```

That will be transformed to the following DOM elements:

```
...
<body>
    <span><input type="text" id="box1" ... /></span>
    <span><input type="text" id="box2" ... /></span>
    <div>
            <input type="text" id="box3" ... />
    </div>
</body>
...
```

If you want, you can disable this mechanism setting this property to false. In such a case, you will need to care to put your widgets in panels, or in other html tags alone, or else, ***Crux*** will not guarantee the order of the sibling widgets.

If this property is disabled, the previous example can build the following sequence of elements into the DOM:

```
...
<body>
    <input type="text" id="box2" ... />
    <div>
            <input type="text" id="box3" ... />
    </div>
    <input type="text" id="box1" ... />
</body>
...
```

In that case, the best would be refactory this to:

```
...
<body>
    <crux:flowPanel id="panel1">
       <crux:textBox id="box1" />
       <crux:textBox id="box2" />
       <div>
              <crux:textBox id="box3" />
       </div>
    </crux:flowPanel>
</body>
...
```

4.3.2.2 localeResolver

By default, ***Crux*** will use the same mechanism used by GWT at client side to resolve the user locale.

It means that you can, for example, pass the locale through an url parameter, like:

```
http://www.example.org/myapp.html?locale=pt_BR
```

However, if you need to change this behavior , you can specify your own LocaleResolver class through the property `localeResolver`. That implementation can adopt a custom rule to identify the user locale.

Your class just need to implement the following interface:

```
public interface LocaleResolver
{
        void initializeUserLocale(HttpServletRequest request);
        Locale getUserLocale() throws LocaleResolverException;
}
```

4.3.2.3 screenResourceResolver

The property `screenResourceResolver` tells ***Crux*** which class will be used to retrieve a stream to the application pages. It is useful to plugins, that need to do some processing with pages before they are consumed by the framework.

Your class just need to implement the following interface:

```
public interface ScreenResourceResolver
{
        InputStream getScreenResource(String screenId) throws InterfaceConfigException;
        Set<String> getAllScreenIDs(String module) throws ScreenConfigException;
}
```

4.3.2.4 classPathResolver

Crux needs to know some paths to can retrieve the HTML pages, scan for controllers and other operations. The problem is that if we are running under some application servers - let's say directly: weblogic :) - the strategy to retrieve the web root, the WEB-INF/classes and WEB-INF/lib paths changes completely.

Because of this, we created the `classPathResolver` configuration parameter. You can use it to change the resolver to make your application work in any weblogic.

Your class just need to implement the following interface:

```
public interface ClassPathResolver
{
        URL findWebInfClassesPath();
        URL findWebInfLibPath();
        URL[] findWebInfLibJars();
        URL findWebBaseDir();
}
```

The following example shows how you could build a ClasspathResolver to work in weblogic (tested with weblogic 10.0):

```
public class WebLogicClassPathResolver extends ClassPathResolverImpl
{
        private static final String A_RESOUCE_FROM_CLASSPATH_ROOT = "/"+WebLogicClassPathResolver.class.getName().replaceAll("\\.", "/")+".class";

        @Override
        public URL findWebBaseDir()
        {
                try
                {
                        URL url = findWebInfClassesPath();
                        File webInfClassesFile = new File(url.toURI());
                        File webRoot = webInfClassesFile.getParentFile().getParentFile();
                        return webRoot.toURI().toURL();
                }
                catch (Exception e)
                {
                        throw new RuntimeException(e.getMessage(), e);
                }
        }

        @Override
```

```
        public URL findWebInfClassesPath()
        {
                try
                {
                        URL url = getClass().getResource(A_RESOUCE_FROM_CLASSPATH_ROOT);
                        String path = StringUtils.removeEnd(url.toString(), A_RESOUCE_FROM_CLASSPATH_ROOT);

                        if(path.toUpperCase().startsWith("ZIP:"))
                        {
                                int firstSlash = path.indexOf("/");
                                path = path.substring(firstSlash + 1);
                                path = StringUtils.removeEnd(path, "!");
                                path = "file:///" + path;
                        }

                        return new URL(path + "/");
                }
                catch (MalformedURLException e)
                {
                        throw new RuntimeException(e.getMessage(), e);
                }
        }
}
```

4.3.2.5 serviceFactory

The section Writing Server Services shows the default mechanism used to discovery your service implementation classes.

However, you can need to change this to, for example, integrate ***Crux*** with some other server framework like spring, guice or to make your service classes EJBs.

Using the property `serviceFactory` you can specify your own class, that just needs to implement the following interface:

```
public interface ServiceFactory
{
        Object getService(String serviceName);
        void initialize(ServletContext context);
}
```

4.3.2.6 allowAutoBindWithNonDeclarativeWidgets

This property tells to ***Crux*** that it must support the AutoBind feature even if the widget that is bound with a specific field is created and added to Screen programmatically. To prevent undesirable bugs on applications, the default value to this is true.

However, if you don't need this feature, you can disable it to have some performance improvement. If this property is set to false, ***Crux*** can transfer to a generator the responsability to handle the binding code, keeping the final code smaller and simpler.

4.3.2.7 enableChildrenWindowsDebug

GWT 2.0 uses a parameter to inform its browser debug plugin that it must enable the debug. If enableChildrenWindowsDebug parameter is set to true, ***Crux*** will propagate the GWT debug parameter to other windows when, for example, a Popup is opened.

4.3.2.8 enableWebRootScannerCache

Crux uses a resource scanner to find the application pages, templates and other things. Some ***Crux*** plugins requires to disable this scanner.

4.3.2.9 enableHotDeploymentForWebDirs

Enabling this options cost a little bit more in performance terms, but can turn easier the developement, once you will not need to restart your server to see changes on templates.

4.3.2.10 enableHotDeploymentForWidgetFactories

Crux only register on its client engine the factories that are alrealdy used to parse your page(s). If you set this option to true, ***Crux*** will register all possible factories. It will turn possible hot deployments that insert new kinds of widgets on your page.

This behavior will be adopted only if you are under development mode (-DCrux.dev=true). During the distribution compilation, you must run in production mode, what will make ***Crux*** ignore this option and optimize to use only what is referenced on page.

5 Crux Tools

5.1 Schema Generator

The `Schema Generator` Tool searches in the project classpath for ***Crux*** widget libraries and generates a XSD file for each of them. It also generates a XSD file for each template file found.

You can invoke SchemaGenerator in two different ways:

1. calling it with the command line:

   ```
   java br.com.sysmap.crux.tools.schema.SchemaGenerator <projectBaseDir> <outputDir>
   ```

2. calling the ant task `<generate-schemas>`, that is already defined on the project `build.xml` file, generated by the Project Generator

SchemaGenerator also produces an eclipse catalog file containing all generated XSD files.

5.2 Project Generator

ProjectGenerator is a tool used to create new ***Crux*** projects according with our default project layout.

After download the ***Crux*** distribution file, unzip its content on an empty folder and you will found a shell command file named `projectGenerator`.

Call that file, passing no parameter, or passing `-help` for display the usage screen.

All configurations about the project generation is done through a file called `project.properties` (that is located on the same folder that projectGenerator command).

The following table shows all the properties contained on that file:

Property	Default Value	Description
projectName	MyApplication	The name of the project that will be created
hostedModeStartupURL	index.html	The name of the initial page of your project. The Debug lanch file will point to this and a page with that name willl be created to server as example.
hostedModeStartupModule	com.mycompany.MyModule	The name of the GWT module created to your application
hostedModeVMArgs	-Xss32768k -Xms64M -Xmx256M -DCrux.dev=true	The startup parameters passed to DevMode JVM
useCruxModuleExtension	false	If true, creates a project following the ***Crux*** Module Layout
cruxModuleDescription	My Module Description	The description of your module. Only used if `useCruxModuleExtension` is true

5.3 Crux Compiler

The `CruxCompiler` Tool compiles a crux project. It converts all `.crux.xml` files to simple `.html` files and then call the GWT compiler to produce the javascript for all your application modules.

You can invoke CruxCompiler in two different ways:

1. calling it with the command line:

```
java br.com.sysmap.crux.tools.compile.CruxCompiler <outputDir>
```

2. calling the ant task `<cruxcompiler>`, that is already defined on the project `build.xml` file, generated by the Project Generator