Building UIs with Ruby and Sinatra on NW BPM

Applies to:

SAP NetWeaver BPM 7.3 EhP1. For more information, visit the official NW BPM Web site on SDN: http://www.sdn.sap.com/irj/sdn/nw-bpm-info.

Summary

SAP NetWeaver BPM (NW BPM) has always had a strong focus on providing seamless integration with UI technologies provided by SAP, specifically Web Dynpro Java / ABAP and Visual Composer. With the advent of the BPM Public API, NW BPM now opens the door for customers to use arbitrary UI technologies. This way, customers can leverage know-how they might already have in other UI technologies and frameworks and fully customize the look and feel of their UIs according to their needs.

This document exemplifies how to build a custom UI on NW BPM using the Ruby scripting language and the Ruby-based Web framework Sinatra for the Web backend and standard HTML, CSS, and JavaScript for client side rendering.

Note this article is not meant for beginners. Solid NW BPM knowledge is assumed as a prerequisite.

Author: Harald Schubert

Company: SAP AG

Created on: 30 May 2012

Updated on: 15 June 2012

Author Bio

Harald Schubert joined SAP in 2005. He was one of the early contributors to SAP NW BPM and now works as a lead architect for the SAP NetWeaver Composition Environment with a focus on BPM and BRM.

Table of Contents

1 Introduction

The BPM Public API offers many new possibilities to consume NW BPM, the most prominent one being the access via custom UIs, for example those tailored to mobile devices such as iOS or Android handsets.

This article guides you through the steps necessary to do just that. I chose Ruby and Sinatra as the UI-tier programming language and Web framework, respectively, as I find both ideal for the task at hand:

- Ruby allows you to quickly pull some scripts together and is a great way to deal with the boiler plate code necessary to connect to the BPM backend through its public APIs. More specifically, JRuby allows you to easily blend Ruby and Java code, making it possible to access the BPM APIs without having to resort to intermediary layers such as REST or SOAP Web services.
- Sinatra is about the most lightweight Web framework that exists today. It is not as feature-rich as competitors such as Ruby on Rails or Python-based Django but what it provides nicely matches the capabilities needed when building on top of a JEE server and process engine in particular. For example, it doesn't come with its own persistence layer – but this isn't necessary either as the BPM runtime and your running process instances constitute your persistence, if you will.

To better illustrate the topics covered in this document, a simple leave request scenario is used. The employee can ask his manager for allowance to take some time off by instantiating a leave request approval process. The manager has the possibility to approve or reject the request. In the latter case, the employee can either cancel or modify and then re-submit the request (Figure 5: Leave Request Scenario).

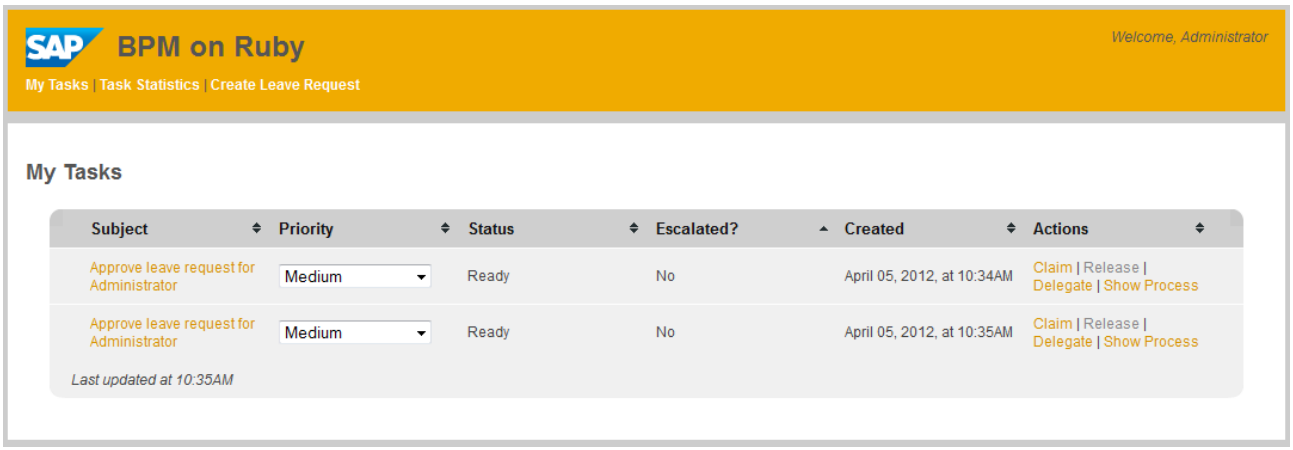

Figure 1: Demo Project

The rest of this document is structured as follows:

- Chapter 2 gives a brief overview of Ruby and Sinatra and how both are used in NW BPM in our demo scenario. For a full coverage of these topics, refer to the references at the end of the document.
- Chapter 3 describes the DC structure used in the leave request scenario and helps you put necessary things in place such as JRuby and Sinatra libraries.
- Chapter 4 outlines the leave request process and its associated tasks.
- Chapter 5 describes how to write a simple Ruby/Sinatra UI and goes into details about a few of the client-side specifics used in the demo scenario.
- Chapter 6 shows how to access the BPM Public API and how to consume the data obtained from it in the Ruby UI.
- Chapter 7 gives an outlook of possible enhancements you could make.
- Chapter 8 is the conclusion.

1.1 About the Demo Project

The full demo project is based on the NW 7.3 EhP1 release (CE usage type) and can be downloaded on Code Exchange at https://cw.sdn.sap.com/cw/groups/bpm-on-ruby. With limitations, it can be run on NW 7.3 as well (refer to the readme.txt in the rubybpm/rbui DC for details).

The UI was tested to work with Chrome 18, Firefox 11, and Internet Explorer 9.

You will need to access to the sources to complete the exercise as some basic configuration files, utility classes and scripts will just be copied to speed up development. If you are among the impatient, you can of course just run the application and read through the code. On Code Exchange (see link above), you will also find a description on how to sync from SVN and integrate it into an SAP NWDI/JDI workspace.

We nevertheless recommend you to take the effort of writing the code and wiring things together to get the most out of this tutorial.

Role Assignment

The project requires authentication and assumes you log in with the Administrator user having the following roles assigned:

- Standard User Role
- BPEM End User role
- Start Process Role

E-Mail Notifications

The project makes use of the e-mail notifications feature. Refer to the link below for details on how to configure your server properly.

http://help.sap.com/saphelp_nw73/helpdata/en/e4/32ff7865fa483dba3686160f5c6dd0/content.htm

Process Visualization

The project also leverages the process visualization which needs to be configured analogously to e-mail notifications. To do so, navigate to NWA → Configuration Management → Infrastructure → Java System Properties and edit the http.baseurl of the tc~bpem~base~ear application to match your host name and port.

2 JRuby and Sinatra in a Nutshell

Ruby is an interpreted scripting language supporting multiple programming styles (specifically functional and procedural/object-oriented programming) that was invented by Yukihiro Matsumoto with the intent to provide a language that is easy to learn and fun to work with. The original version of the interpreter was written in C in the mid-1990s and still prevails as the leading implementation used in practice. It also serves as the reference implementation against which other implementations must qualify. JRuby is a more recent implementation that serves the Java community and, as with NW CE / BPM, is particularly attractive if you have a Java backend you want to work with. With JRuby, you can write beautiful Ruby code while at the same time leverage the vast code base offered by the Java ecosystem simply by calling (pretty much any type of) Java code from within a Ruby program.

Sinatra is a modular Web framework written in Ruby for providing lightweight Web applications. It follows a REST approach in which you register Ruby handlers with specific (patterns of) URLs to process user requests and render back HTML or JSON responses, for example:

```
require 'sinatra'

get '/hi' do
  "Hello World!"
end
```

The actual generation of the response markup can be done with your template language of choice. ERB (Embedded Ruby) is the standard template language shipping with Ruby. It is also the one used for the demo scenario in this project.

Rack is a thin adapter layer that basically bridges between Ruby Web servers and Ruby Web frameworks. JRuby-Rack is the counterpart for the Java world, making it possible for you to run a Ruby Web framework on a Java Servlet container. Any Web framework providing Rack support automatically runs on all Web servers supported by Rack.

With these building blocks in mind, we arrive at the following picture describing the architecture of our leave request scenario implementation:

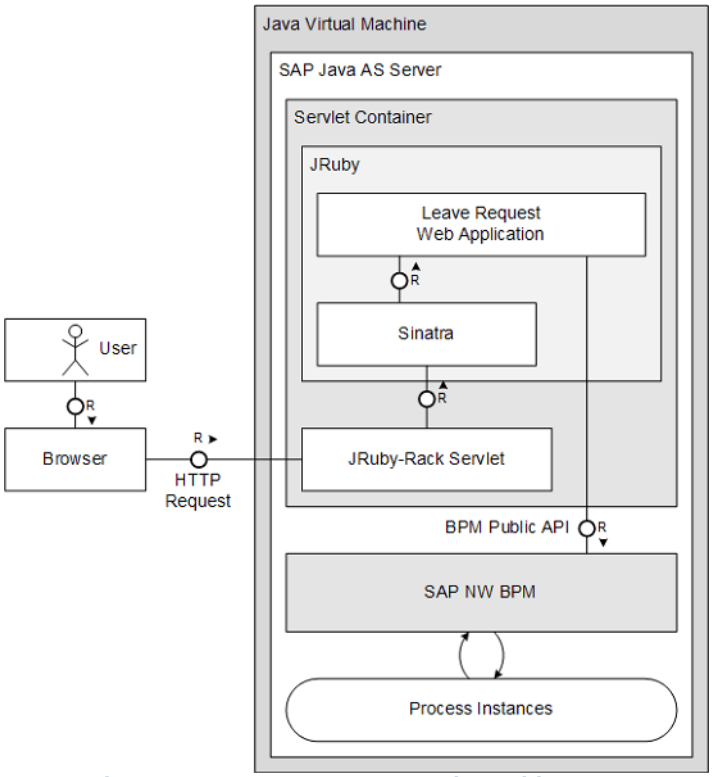

Figure 2: Leave Request Scenario Architecture

3 Project Setup

Before we get started with the actual implementation, we will set up the DC structure that we'll need for storing our development artifacts. For our application, we need five DCs:

- **BPM DC –** Contains your leave request process and tasks.
- **WD UI DC –** Contains the WD UIs you get when generating your tasks via the process context. Your users don't really use these UIs but the BPM engine expects them to be there at runtime. Also, these will be the UIs that get opened when accessing tasks through UWL. Finally, note this DC is best created for you by using the UI generation coming with NW BPM (see task generation in chapter 4).
- **EAR DC –** Assembles the external library dependencies and Web module into a deployable JEE enterprise archive.
- **Web DC –** Contains your actual Ruby Web application and any other Java glue code you might need.
- **External Library DC –** Provides the jars you need for running your Ruby application: JRuby, JRuby-Rack, the Ruby standard library, and Log4J (used for logging in JRuby and your Web app).

This leads to the following picture:

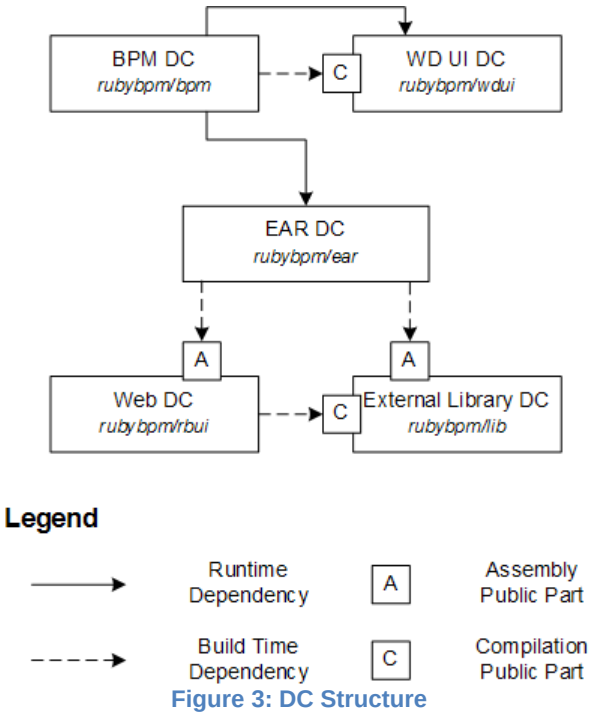

Figure 3: DC Structure

Note that the runtime dependency between BPM and the EAR DC is purely voluntary. I added it to express the fact that the BPM process conceptually depends on the Ruby UIs the same way it depends on the WD UIs. That way, you can ensure that all parts are in place at runtime. If you choose to define a deploy time dependency, too, you can deploy the entire application in a single step by deploying the BPM DC.

In addition to the intra-application dependencies outlined above, you will need to add a couple of extra dependencies to the EAR and Web DC to gain access to the BPM Public API (all with vendor sap.com):

EAR DC:

- SC BPEM-FAÇADE → DC tc/bpem/façade/ear
  - Runtime dependency to DC itself (not the *api* compilation public part)
- SC ENGFACADE → DC tc/je/sdo21/api
  - Runtime dependency to DC itself (not the *api* compilation public part)

Web DC:

- SC ENGFACADE → DC tc/je/usermanagement/api
  - Build time dependency to *api* compilation public part
- SC BPEM-FAÇADE → DC tc/bpem/façade/ear
  - Build time dependency to *api* compilation public part
- SC ENGFACADE → DC tc/je/sdo21/api
  - Build time dependency to *api* compilation public part

4 Modeling the Leave Request Process and Supporting Tasks

4.1 Control Flow

As explained in the introduction, we'll look into a leave request process with two roles involved: an employee and his manager. To that end, create a new process *Leave Request Process* with a pool *Leave Request* and two lanes in it, and name them *Employee* and *Manager* (if you didn't do so already, create a Process Composer DC rubybpm/bpm now):

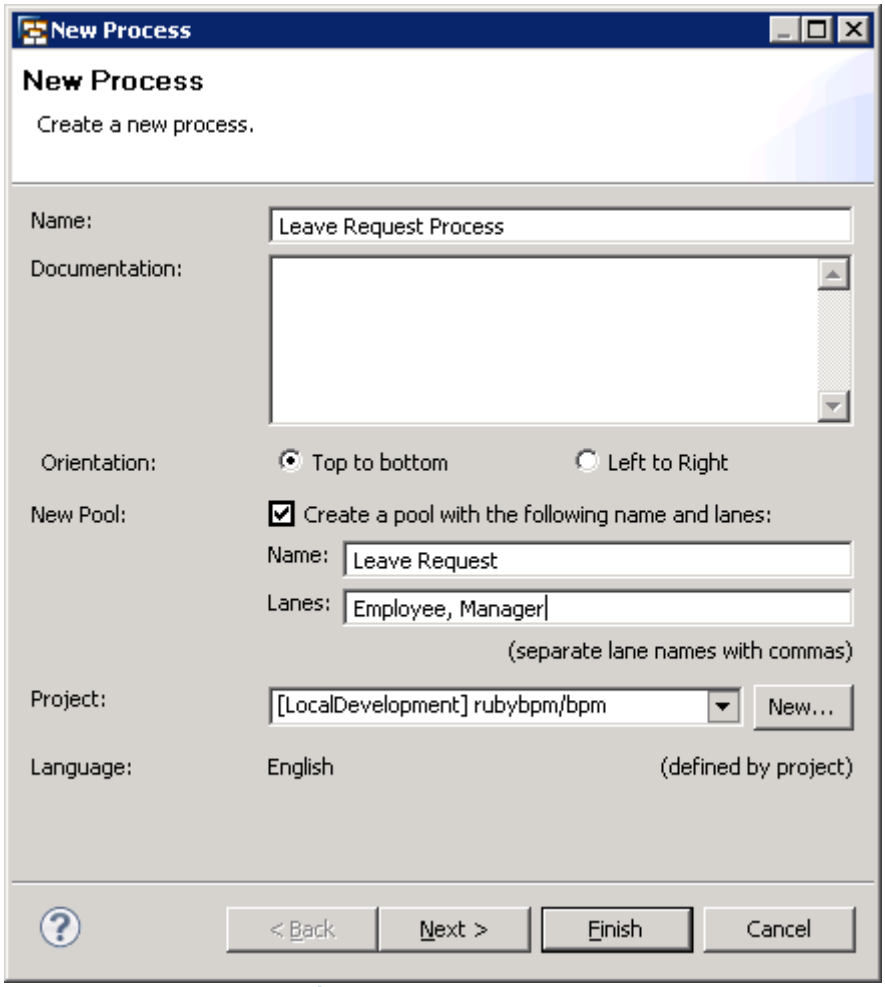

Figure 4: Create Process

Then, draw the sequence of steps shown in Figure 5: Leave Request Scenario:

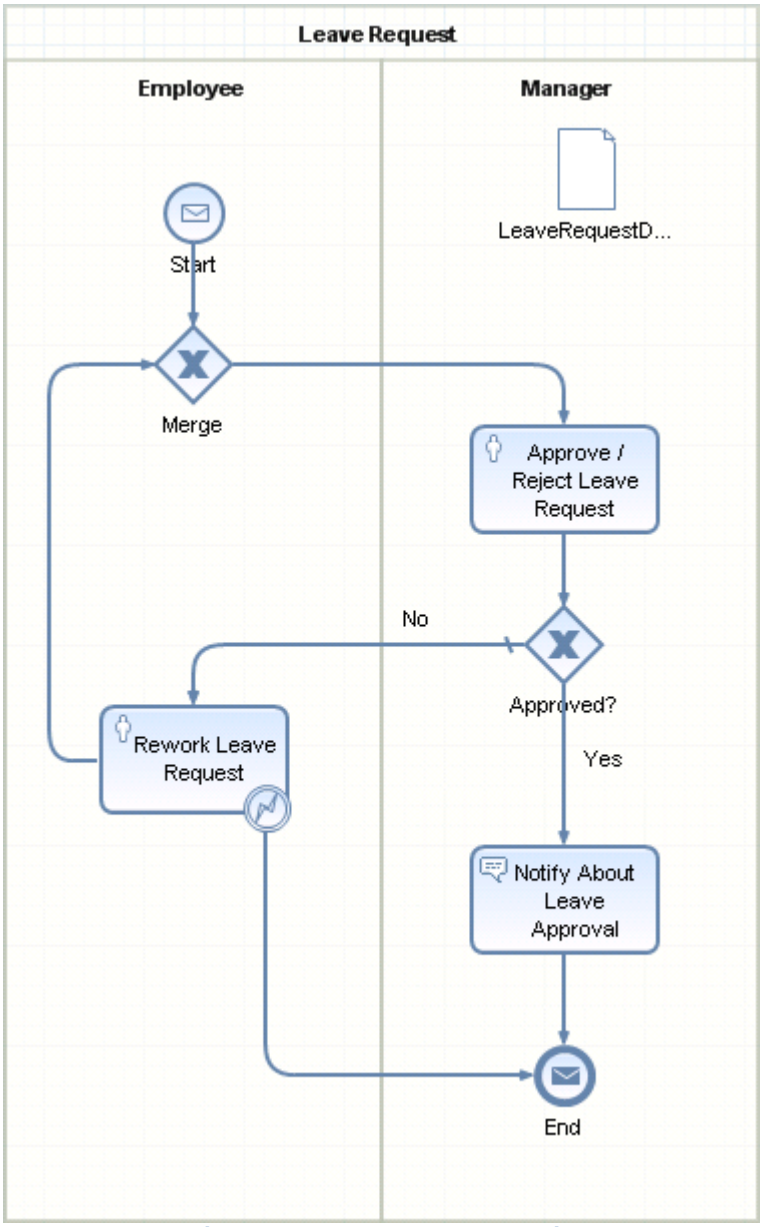

Figure 5: Leave Request Scenario

Below you find a brief explanation of all steps:

- **A start event –** This event gets triggered through the yet-to-be-built leave request start UI. Its payload contains all necessary information such as the requester, vacation start and end date, and some room for notes. The event payload gets stored in a dedicated data object.
- **A merge gateway –** a gateway used to merge the flow of execution from a rework leave request activity back into beginning of a process to re-initiate the manager approval.
- **A manager approval human activity –** the manager is asked to either approve or reject the leave request at this point. His decision gets stored in a boolean field in the same data object in which the start event payload was stored.
- **An exclusive choice gateway –** this is the point at which based on the manager's decision, the process either finishes by sending out a notification to the employee or the employee gets the opportunity to change his request or cancel it altogether.
- **A rework leave request human activity –** the leave request rework step for the employee. Note you can only add the boundary event once you assigned a task to the activity (chapter 4.3).
- **A notification activity –** the step in which a leave request confirmation mail is sent to the employee.
- **An end event –** terminates the leave request process.

4.2 Data Flow, Part 1

To begin with, create these two files inside the src/wsdl folder of your BPM DC:

LeaveRequestData.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns=http://www.w3.org/2001/XMLSchema
        targetNamespace=http://com.sap.demo/LeaveRequestData
        xmlns:tns="http://com.sap.demo/LeaveRequestData"
        elementFormDefault="qualified">

        <complexType name="LeaveRequestData">
                <sequence>
                        <element name="requester" type="string"></element>
                        <element name="fromDate" type="date"></element>
                        <element name="toDate" type="date"></element>
                        <element name="notes" type="string"></element>
                        <element name="approved" type="boolean"></element>
                </sequence>
        </complexType>

</schema>
```

LeaveRequestProcess.wsdl:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:leaveapp="http://com.sap.demo/LeaveRequestProcess/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="LeaveRequestProcess"
        targetNamespace="http://com.sap.demo/LeaveRequestProcess/">
        <wsdl:types>
                <xsd:schema targetNamespace="http://com.sap.demo/LeaveRequestProcess/">

                        <xsd:complexType name="LeaveRequest">
                                <xsd:sequence>
                                        <xsd:element name="requester" type="xsd:string"></xsd:element>
                                        <xsd:element name="fromDate" type="xsd:date"></xsd:element>
                                        <xsd:element name="toDate" type="xsd:date"></xsd:element>
                                        <xsd:element name="notes" type="xsd:string"></xsd:element>
                                </xsd:sequence>
                        </xsd:complexType>

                        <xsd:element name="requestLeave"
                                type="leaveapp:LeaveRequest">
                        </xsd:element>

                        <xsd:element name="requestLeaveResponse">
                                <xsd:complexType>
                                        <xsd:sequence>
                                        </xsd:sequence>
                                </xsd:complexType>
                        </xsd:element>

                </xsd:schema>
        </wsdl:types>

        <wsdl:message name="requestLeaveRequest">
                <wsdl:part element="leaveapp:requestLeave" name="parameters" />
        </wsdl:message>

        <wsdl:message name="requestLeaveResponse">
                <wsdl:part element="leaveapp:requestLeaveResponse" name="parameters" />
        </wsdl:message>

        <wsdl:portType name="LeaveRequestProcess">
                <wsdl:operation name="requestLeave">
                        <wsdl:input message="leaveapp:requestLeaveRequest" />
                        <wsdl:output message="leaveapp:requestLeaveResponse" />
                </wsdl:operation>
        </wsdl:portType>
```

```
        <wsdl:binding name="LeaveRequestProcessSOAP" type="leaveapp:LeaveRequestProcess">
                <soap:binding style="document"
                        transport="http://schemas.xmlsoap.org/soap/http" />
                <wsdl:operation name="requestLeave">
                        <soap:operation
                                soapAction="http://com.sap.demo/LeaveRequestProcess/requestLeave" />
                        <wsdl:input>
                                <soap:body use="literal" />
                        </wsdl:input>
                        <wsdl:output>
                                <soap:body use="literal" />
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>

        <wsdl:service name="LeaveRequestProcess">
                <wsdl:port binding="leaveapp:LeaveRequestProcessSOAP"
                        name="LeaveRequestProcessSOAP">
                        <soap:address location="http://com.sap.demo/" />
                </wsdl:port>
        </wsdl:service>

</wsdl:definitions>
```

You should now see the following tree (refresh the project if this isn't the case):

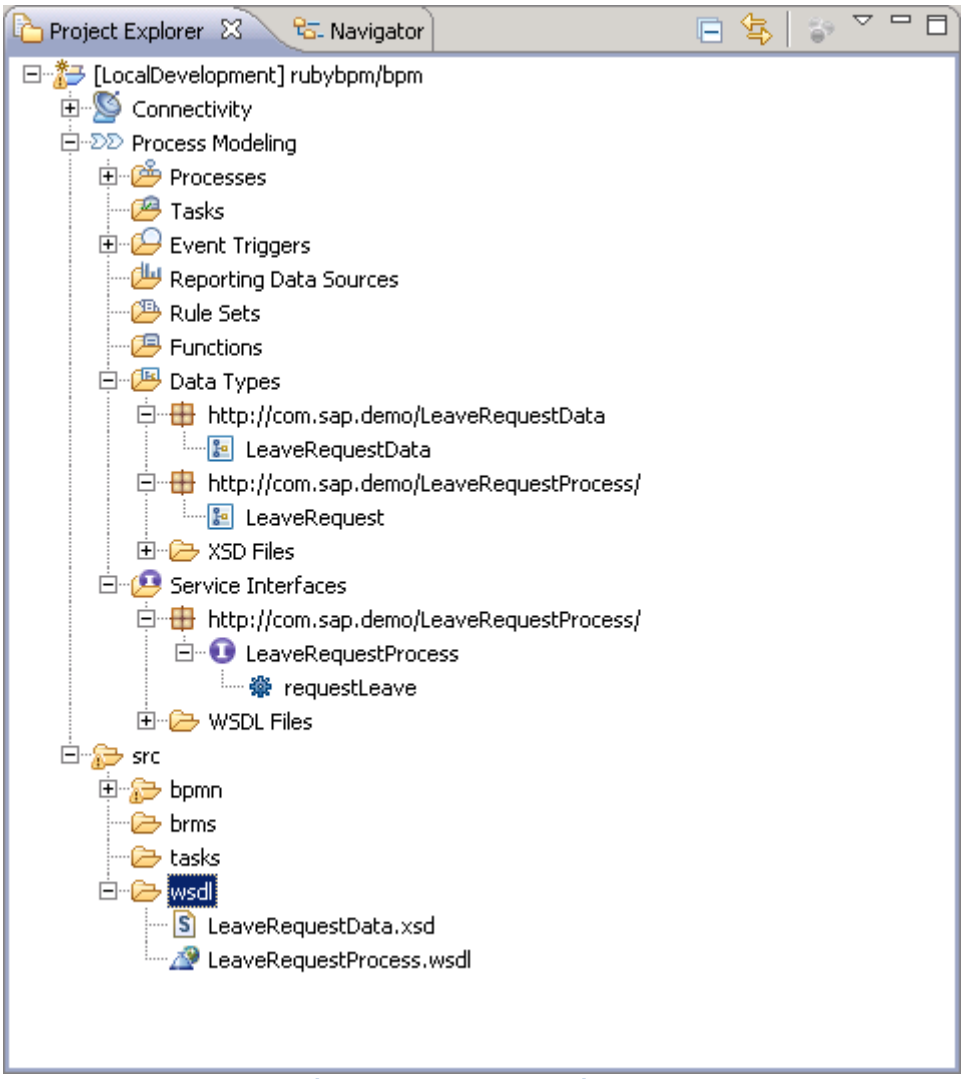

Figure 6: Import XSD and WSDL

Next, drag the service interface operation *requestLeave* shown in Figure 6: Import XSD and WSDL onto the start and end event. This will set the WSDL operation as the Web service interface of the leave request process through which you can create new leave request instances.

Now, drag the *LeaveRequestData* data type onto the pool (not the pool header but either lane). This will create a data object with the data type set to exactly the one you dragged (rename it as you see fit).

The data object will basically capture the operational state of the leave request, including the manager approval status. Use the approved field to implement the condition in the exclusive choice gateway.

You are now also ready to do a data mapping from the start event payload onto the data object. To this end, do a right-drag from the start event payload on the left to the *LeaveRequestData* data object on the right:

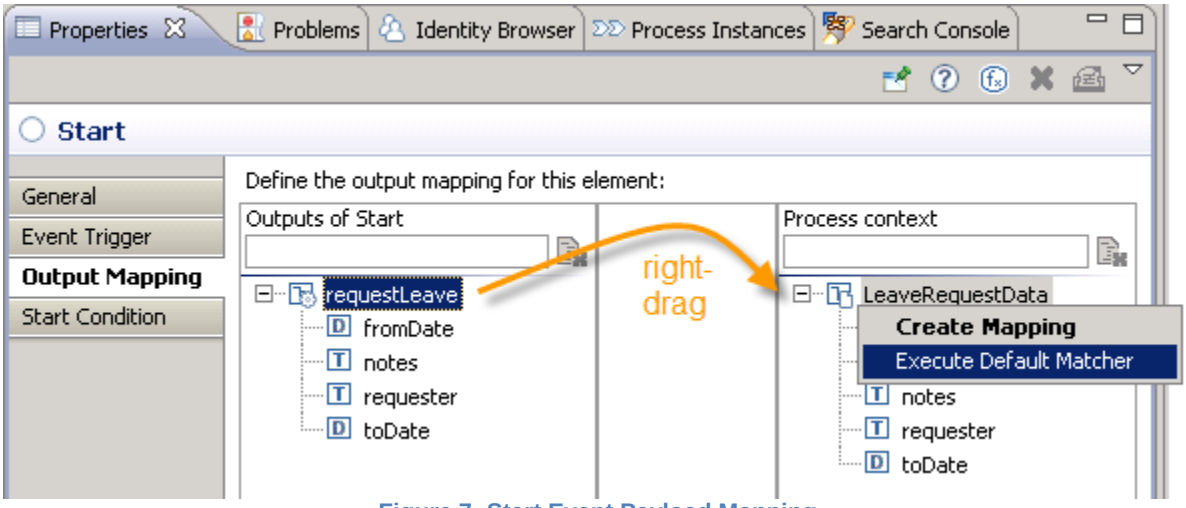

Figure 7: Start Event Payload Mapping

This will do a name-based matching of source data structure onto the target data structure.

4.3 Task Creation

Before we can continue mapping process context data into the approval and rework steps, we need to create tasks for them. The nice side effect of defining the process context first is that tasks and UIs can be nicely generated from there. Go to the *Task* tab of the approval step and select *New…* from the Task drop-down. This will launch the task creation wizard. Enter a name for the approval task, e.g. *ApproveLeaveRequestTask*, ensure the Generate UI Component checkbox is checked, then hit *Next*, confirm the pre-selected *Leave Request Process*, and you end up in a wizard step where you need to select UI technology and DC. Stick to WebDynpro, create a new development component *rubybpm/wdui*, confirm the WD UI component details on the next wizard screen, and you will end up in a step where you need to choose the portion of the process context you want to take over into your task UI. Take everything:

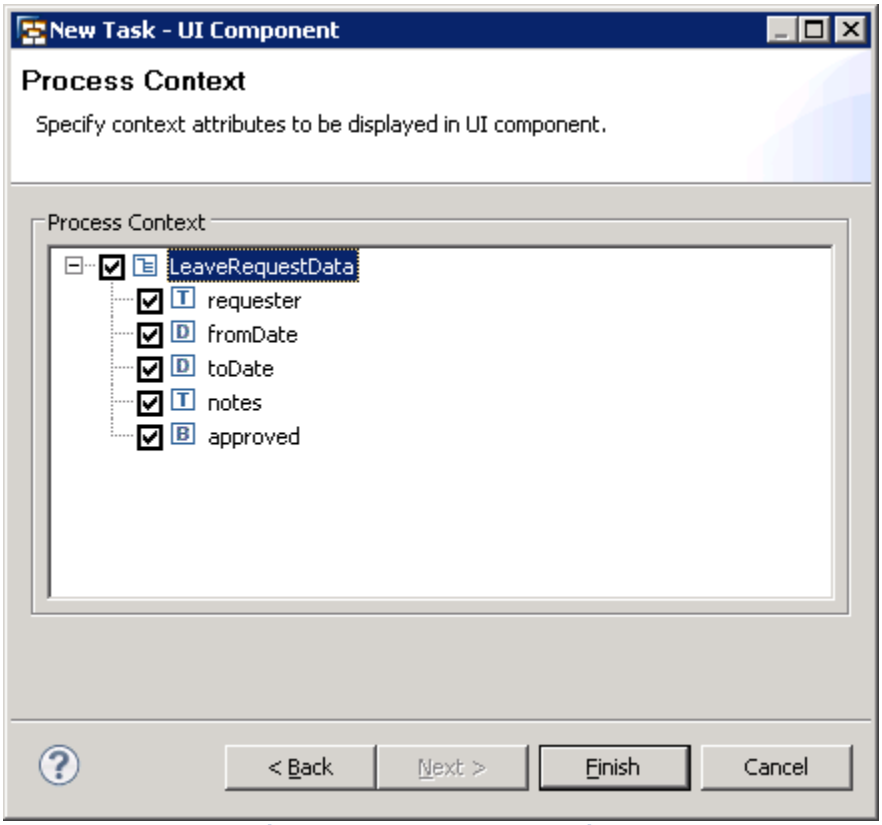

Figure 8: Create Approve Task UI

Now, do the same for the rework task, naming it *ReworkLeaveRequestTask* and choose the following process context:

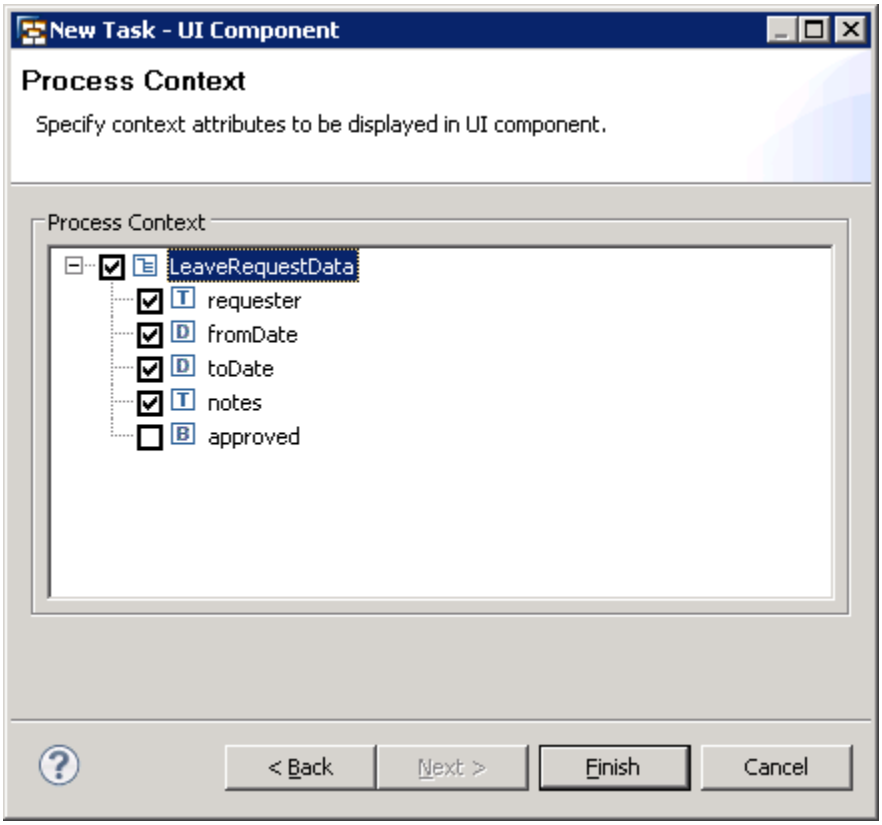

Figure 9: Create Rework Task UI

At this point, you have to do the usual things like assigning potential owners to the tasks, defining user texts that are to be shown in UWL or your custom work list. I leave it to you as an exercise to come up with a good design here – for the purpose of this exercise, all roles are set to *Administrator* anywhere a role is required and it is assumed you log in with this user (see chapter 1.1 for details).

Also, make sure you add the boundary event to the rework task. In case you chose to name the rework task as I did, the boundary event you want to pick should read *ReworkLeaveRequestErrorEvent*.

4.4 Data Flow, Part 2

With tasks in place, we can close off data flow definition. We need to do the following things: (1) do an input mapping from our data object into the approval task, (2) an output mapping from the approval task back to the data object, and (3) repeat both of these steps for the rework task. Be careful not to accidentally overwrite the approved flag during the output mapping of the rework task. Here is one example for the approval input mapping:

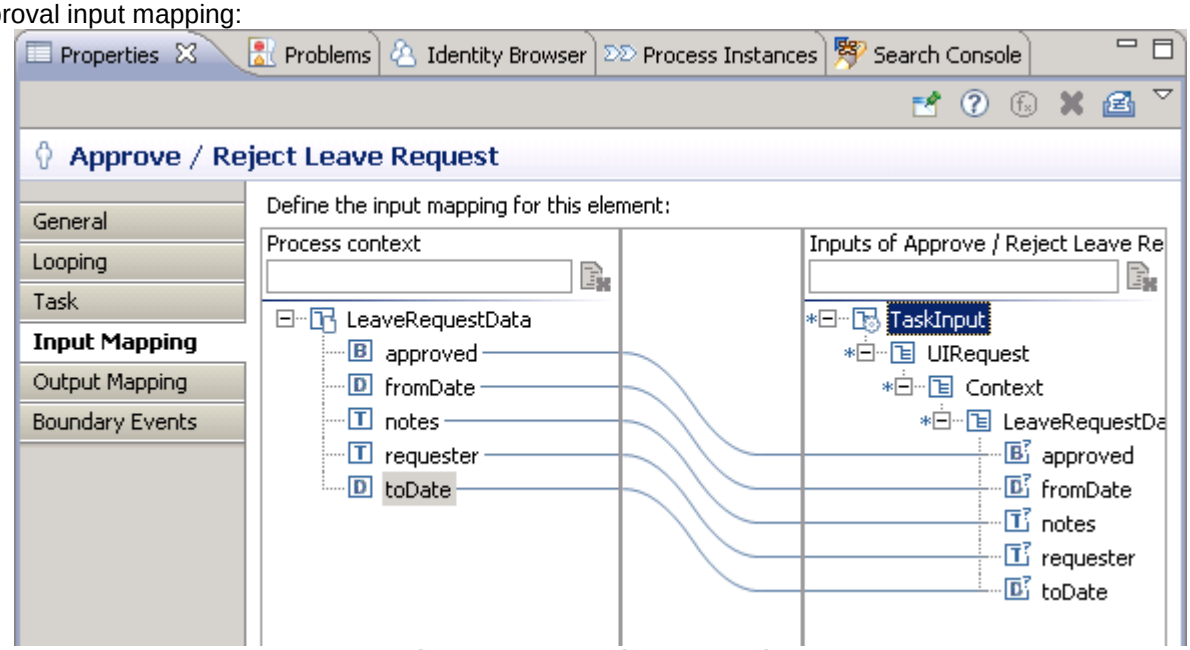

Figure 10: Approval Input Mapping

4.5 Sending a Notification

Finally, we want to send a notification in case the leave request was approved. To shorten things a bit, here is the configuration that should do the trick for our scenario:

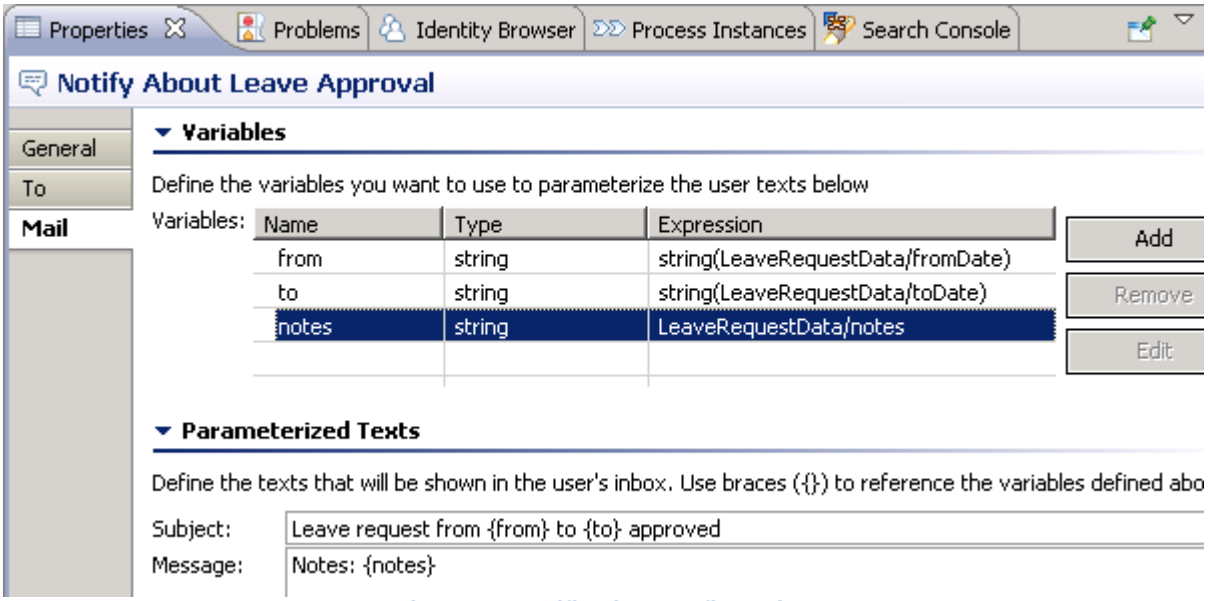

Figure 11: Notification Configuration

Remember you need to have your NW AS Java configured properly in order to receive notification mails (see chapter 1.1 for details).

At this point, you should be able to trigger of a leave request process instance and run it to completion. Try and test this using the process start UI reachable through NWA.

5 Your First Ruby UI

So we now have a fully working leave request process running, including tasks and auto-generated WebDynpro Java UIs – great. Let's now try to get going with our own custom UIs.

If you didn't create the EAR DC, Web DC, and External Library DC so far, now is a good time to do so. Pay special attention to the public parts and dependencies between DCs (see chapter 3).

After that, download the libraries shown in Figure 12: Publish as Archive from the Web and copy them to the libraries folder of the External Library DC and add them as an archive to the *archives* assembly and *api* compilation public part, respectively:

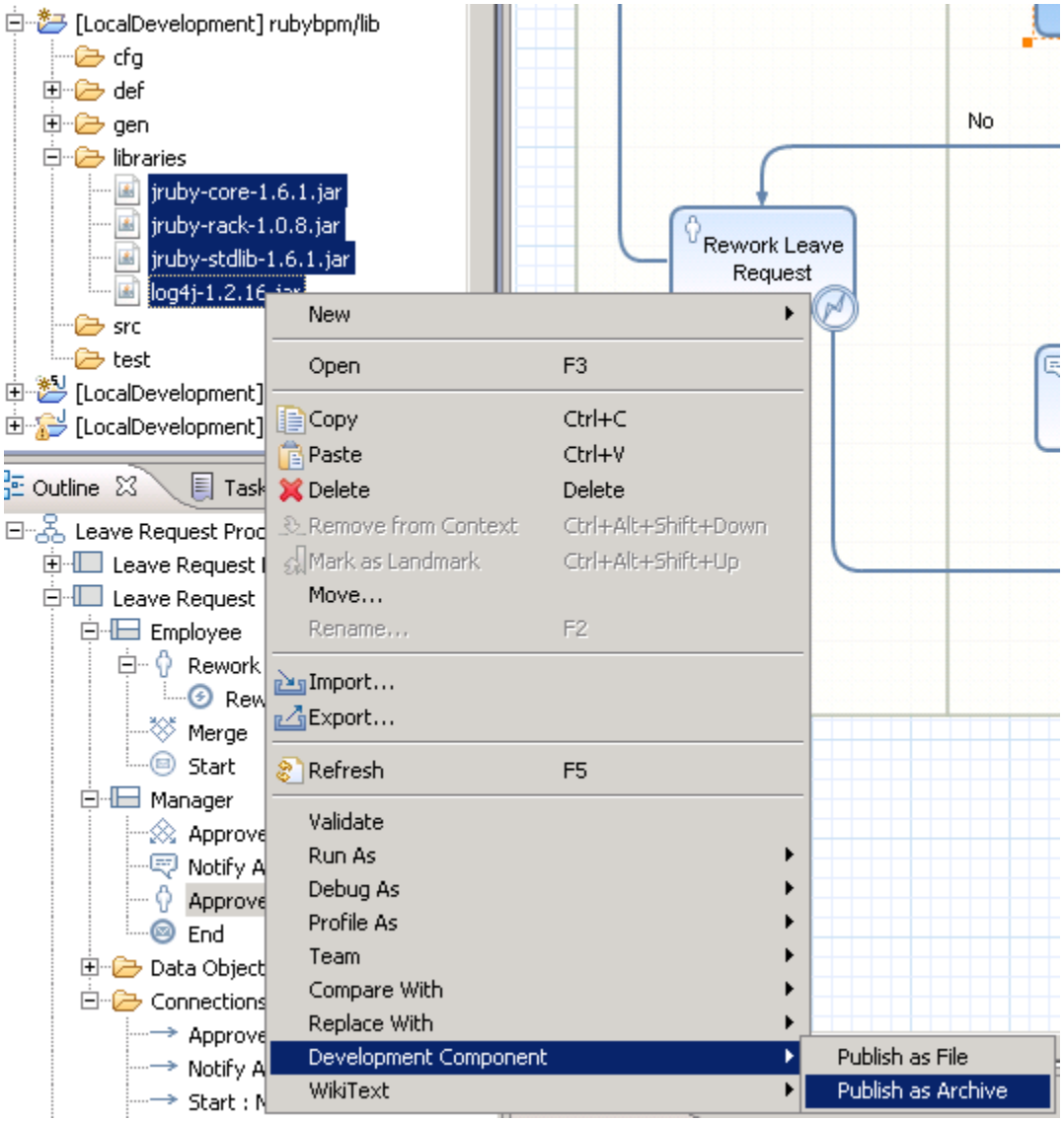

Figure 12: Publish as Archive

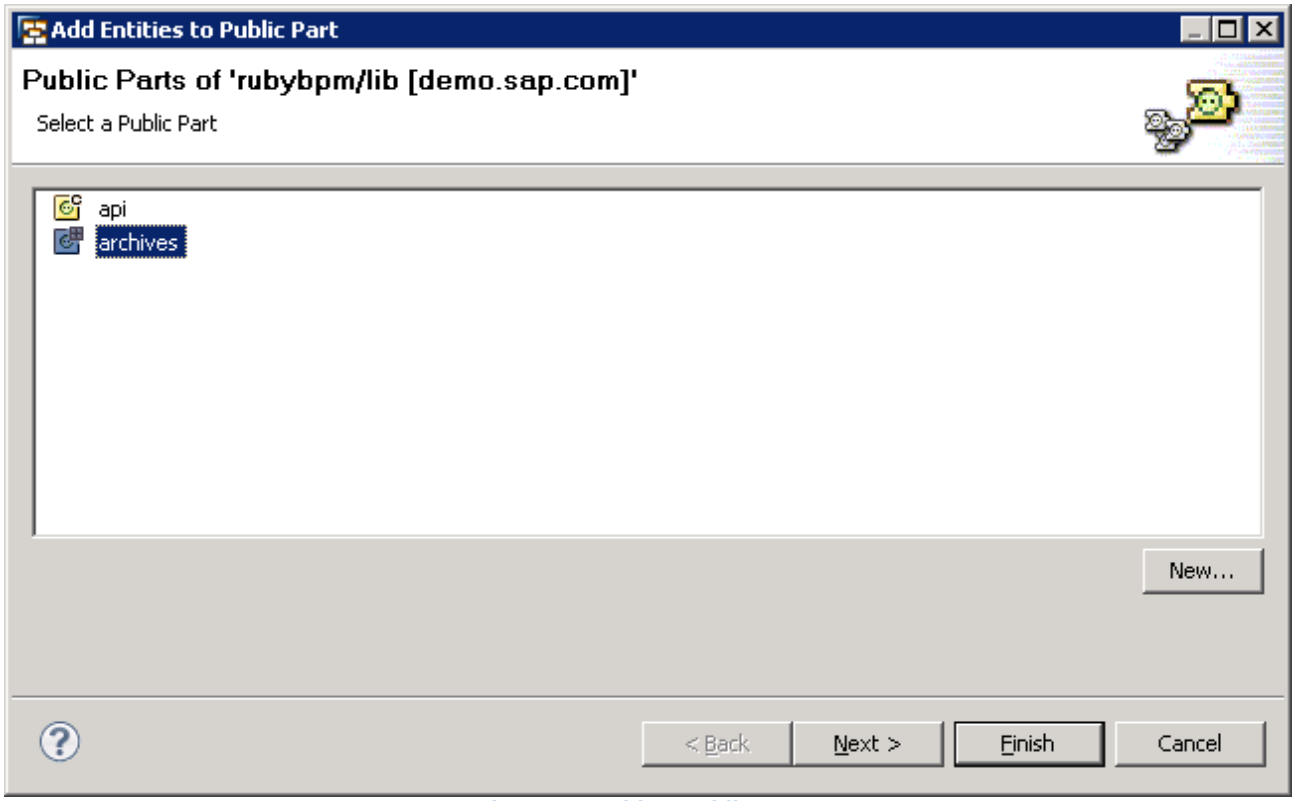

Figure 13: Add to Public Parts

That's it for the external libraries. The next step is a bit more involving. I initially came up with the structure of this DC using Warbler, a tool that assembles a standard ruby application into a Java Web archive together with all of its dependencies, and then migrating bit by bit to a DC structure. I will not go into details of each concept and configuration file here. After all, this is not a Ruby tutorial. The simplest would be to get the source files accompanying this article, extract the rubybpm/rbui DC, and – after backing up old files – copy the following things to your own DC:

- WebContent/WEB-INF/config.ru — JRuby-Rack config file
- WebContent/WEB-INF/web.xml — Java Web config file
- WebContent/WEB-INF/web-j2ee-engine.xml — SAP AS Java config file
- WebContent/WEB-INF/gems/**/*.* — Ruby gems we'll depend on
- WebContent/WEB-INF/public/**/*.* — Public CSS, JavaScript, and images
- WebContent/WEB-INF/utils/**/*.* — Some Ruby helpers we use
- source/**/*.* — Some Java helpers we use, Log4J configuration

Make sure to change the display-name property in web.xml back to the correct value (*LocalDevelopment~rubybpm~rbui~demo.sap.com* in our example).

Then, create a file WebContent/WEB-INF/rubyweb.rb and folder WebContent/WEB-INF/views. We'll write our UI controller code and place our view templates here, respectively.

Open rubyweb.rb and add the Sinatra code snippet given in chapter 2. Deploy the EAR DC to your server and point your application to <rbui app root>/hi (e.g. http://localhost:50000/demo.sap.com~rubybpm~rbui/hi). You should see a hello world message.

Side note: if you prefer using a dedicated Ruby editor with syntax highlighting, look out for the Eclipse DLTK which comes with a sample Ruby editor (http://www.eclipse.org/dltk/).

6 Accessing the BPM Public API

6.1 BPM Public API

The BPM Public API is the entry point to programmatically accessing NW BPM and allows you to address both task and process-related aspects. We won't cover it in detail here. For details, refer to the online help on http://help.sap.com/javadocs/.

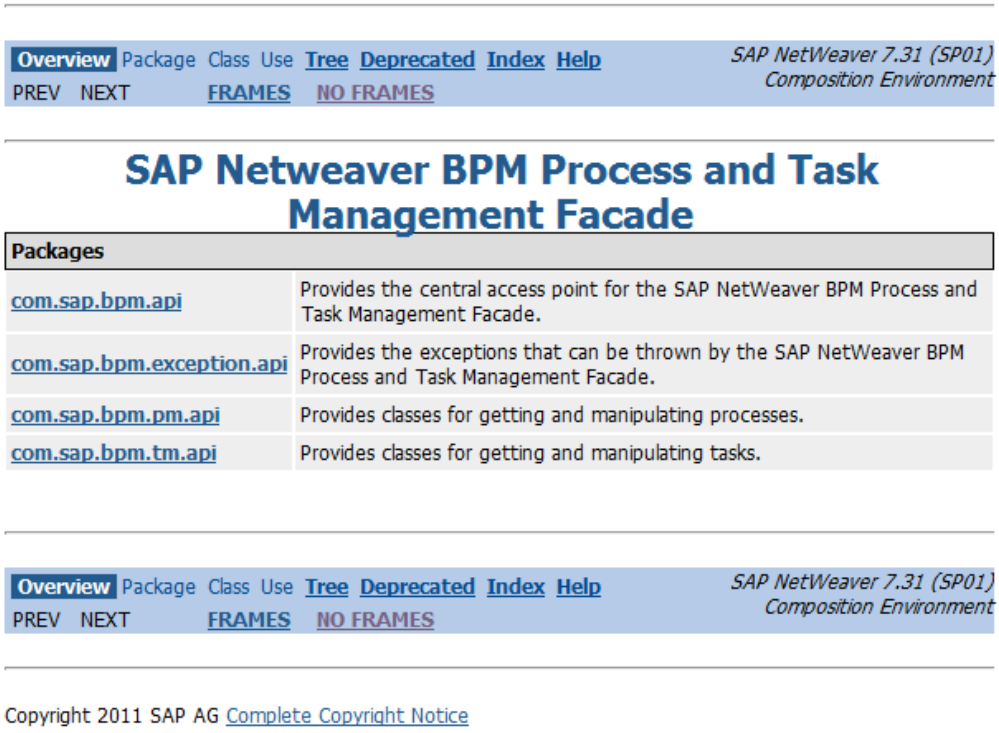

Figure 14: BPM Public API

6.2 A Simple Task List

Now that the basic JRuby setup is done, it is time for something more BPM-like. The first thing we want to try is building our own worklist. We will need to configure authentication to retrieve tasks for a particular user. The files we copied in chapter 5 where a good starting point to get the project configured properly and provide some basic helpers that will ease development.

As we'll do real coding now, it might be a good idea to discuss a practical but important aspect of developing with JRuby: when to use Java and when Ruby. Obviously, JRuby offers support for both. That means, you can decide what pieces of functionality you want to develop in plain old Java and what pieces are better done with Ruby. There is no strictly technical answer to this question. In the end, it comes down to a matter of taste. I prefer to use Java when I heavily access Java APIs anyway (such as the BPM Public API), and then create thin wrappers that I call from Ruby to implement the UI logic. This is what you will find in this exercise.

To begin with, create a Java class TaskManager and place it in the rubybpm/rbui DC. Implement it as follows (package name and imports omitted; you can find the fully implemented version of this class in the test.rubyweb.tasks Java package):

```
public class TaskManager {

      static Logger logger = Logger.getLogger(TaskManager.class);
```

```
	public TaskAbstract[] getMyTasks() {
		try {
			TaskInstanceManager taskInstanceManager = BPMFactory
					.getTaskInstanceManager();
			HashSet<Status> statuses = new HashSet<Status>();
			statuses.add(Status.READY);
			statuses.add(Status.RESERVED);
			statuses.add(Status.IN_PROGRESS);
			Set<TaskAbstract> myTasks = taskInstanceManager
					.getMyTaskAbstracts(statuses);

			return myTasks.toArray(new TaskAbstract[0]);
		} catch (BPMException e) {
			logger.error(e.getMessage(), e);
		}

		return new TaskAbstract[0];

	}
}
```

This will yield an array of ready, reserved, and in-process task abstracts for the current user. Now, we only need to invoke this method from our Ruby code and render the information in a nice HTML table. To do so, open rubyweb.rb and add another GET handler:

```
get '/tasks/?' do
  include_class 'com.foo.bar.TaskManager' #change accordingly

  tm = TaskManager.new
  @tasks = tm.my_tasks

  erb :tasks
end
```

This requires some explanation. When the Sinatra Web application receives a GET request, all registered GET handlers are matched against the request's URL (to be precise, the portion of the URL relative to the Web application root URL). When a matching handler is found, it is executed.

What we now do in our coding is importing the Java class we previously implemented and creating a new instance we can work with. We then invoke the getMyTasks() method and store the array of tasks in a member variable. Note how JRuby automatically translates between *snake_case* used in Ruby and *camelCase* found in Java as well as getter/setter convention differences between the languages so you can use the syntax that's common in each language. Finally, we render a view template that we yet have to write.

Member variables have a special meaning in Sinatra: they are available when rendering view templates. Applied to our example, we have retrieved a bunch of tasks and stored them in the member variable @tasks. Let's see how we can make good use of that.

Go to the WebContent/WEB-INF/views directory, create a file called tasks.erb, open it, and implement the following piece of HTML/Ruby code:

```
<html>
<head><title>My Task List</title></head>
<body>
<h2>My Tasks</h2>
<% if @tasks.any? %>
<table style="border:1px solid">
```

```
      <thead>
          <tr>
            <th>Subject</th>
            <th>Priority</th>
            <th>Status</th>
            <th>Escalated?</th>
            <th>Created</th>
          </tr>
      </thead>
      <tbody>
<% @tasks.each do |t| %>
          <tr>
            <td><%= t.presentation_name %></td>
            <td><%= t.priority.label %></td>
            <td><%= t.status.label %></td>
            <td><%= t.escalated? ? "Yes" : "No" %></td>
            <td><%= Time.at(t.created_time.time / 1000).
                       strftime("%B %d, %Y, at %I:%M%p") %></td>
          </tr>
<% end %>
      <tbody>
</table>
<% else %>
<p>There are no tasks for you.</p>
<% end %>
</body>
</html>
```

As you can see, we access the tasks variable we filled in the GET handler above and iterate over all tasks, one by one, and render a table row on an HTML page. After deploying your application and pointing your browser at <rbui app root>/tasks you should read a message that there are no tasks for you or, if you did create a leave request before (see chapter 4), a simple table with some information about tasks for you:

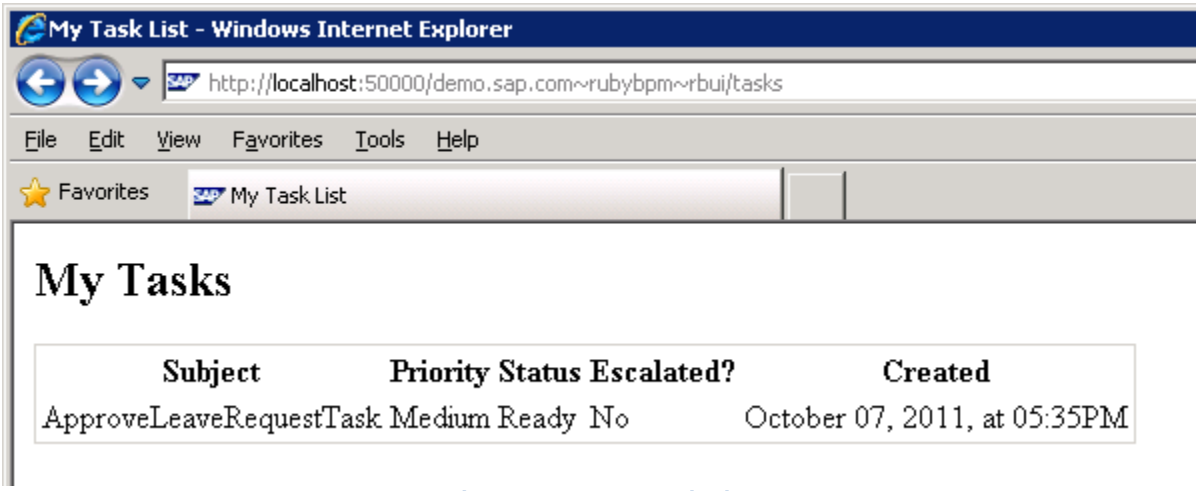

Figure 15: Empty Task List

The next thing we'll hence do is provide a user interface through which we can do exactly that.

6.3 Triggering Leave Requests

With our basic task list being in place, it is time to provide a more user-friendly and less generic way to create leave requests than that offered through the process start UI in NWA.

To start with, create an additional GET handler inside rubyweb.rb:

```
get '/leave_request/?' do
  erb :leave_request_new
end
```

As before, this allows us to render a custom Web page when the browser sends a GET request to <rbui app root>/leave_request. We will use this to show an input form for a leave request.

Next, create a file leave_request_new.rb in folder WebContent/WEB-INF/views and drop the following HTML table snippet:

```
<table>
<form action="<%= url "leave_request" %>" method="post">
    <tr><td><label for="fromDate">From:</label></td>
      <td><input type="text" name="fromDate" id="fromDate" /></td></tr>
    <tr><td><label for="toDate">To:</label></td>
      <td><input type="text" name="toDate" id="toDate" /></td></tr>
    <tr><td><label for="notes">Notes:</label></td>
      <td><textarea name="notes" id=”notes”
        cols="50" rows="5"></textarea></td></tr>
    <tr><td><input type="submit" value="Submit" class="button" /></td></tr>
</form>
</table>
```

Pointing at <rbui app root>/leave_request you should be presented with a small Web form as shown below:

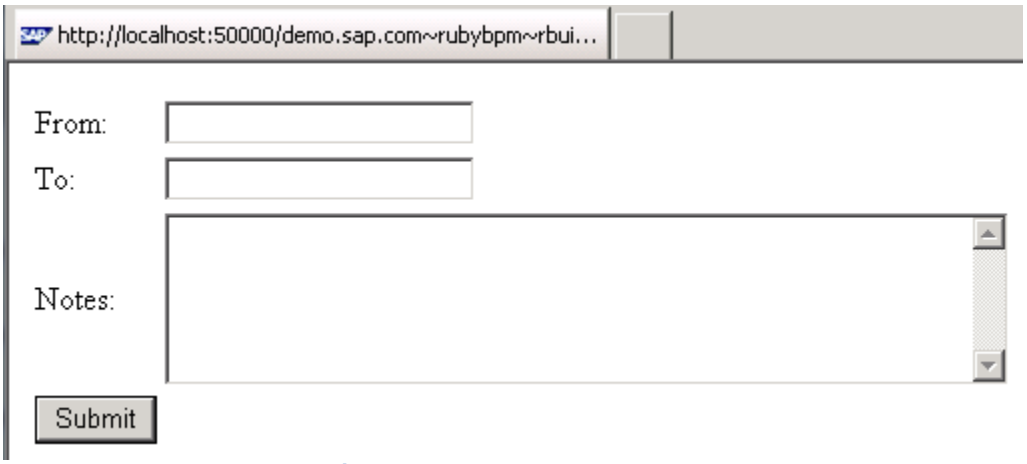

Figure 16: Create Leave Request

If you try creating a leave request via this form you will notice this leads to nothing but an HTTP 404 message. For this to go away we need to actually handle the POST request. Back to your rubyweb.rb module, insert a Sinatra POST handler as follows:

```
post '/leave_request/?' do
```

```
  include_class 'test.rubyweb.processes.ProcessManager'

  pm = ProcessManager.new

  begin
    request = populate_leave_request(params)
    @from_date = request[:from_date]
    @to_date = request[:to_date]
    @notes = request[:notes]

    process_id = pm.start_leave_request(@from_date, @to_date, @notes)
    "<h4>Leave request with id #{process_id} has been successfully
created.</h4>"
  rescue Exception => e
    "<h4>Leave request could not be processed. See error log for details.</h4>"
  end

end

def populate_leave_request(params)
  result = {}

  from_date_str = params[:fromDate]
  from_year = from_date_str[0..3].to_i
  from_month = from_date_str[5..6].to_i
  from_day = from_date_str[8..9].to_i
  result[:from_date] = Time.local(from_year, from_month, from_day)

  to_date_str = params[:toDate]
  to_year = to_date_str[0..3].to_i
  to_month = to_date_str[5..6].to_i
  to_day = to_date_str[8..9].to_i
  result[:to_date] = Time.local(to_year, to_month, to_day)

  result[:notes] = params[:notes]

  result
end
```

As before, we first import the Java class we want to use and create an instance we can interact with. Since the code is rather lengthy, I'll refrain from writing the class from scratch but use the implementation provided with the accompanying code instead (test.rubyweb.processes.ProcessManager).With the process manager in place, we then parse the form data and populate a call to the process manager to create a new leave request. If all goes well, we show a success message; otherwise, a basic error message is provided.

The Java class uses hard-coded constants to create a leave request process. You might have to adapt those in case the DC name your process model resides in is not named rubybpm/bpm[1] (vendor: demo.sap.com) and/or the name of your process is not Leave Request Process.

When you now create a new leave request, your work list should show new tasks accordingly upon refresh.

6.4 Processing Tasks

By now, you can create leave requests and you can look at a list of tasks you have to work on. What's missing is a way for you to actually process the tasks with custom UIs, too.

[1] The technical name of a DC (which you need to provide to the BPM public api) equals the name of the DC you entered in the DC creation wizard with all slashes replaced by tildes: rubybpm/bpm → rubybpm~bpm.

The basic mechanism is the same as in the previous chapter so I won't repeat things here. It's probably easier to look at the code directly (test.rubyweb.tasks.TaskManager). The Java code is a bit more involved and requires a solid understanding of the generic SDO API used in BPM to complete or cancel tasks, for example. Refer to the official BPM documentation in this regard.

The picture below has some details on how you can approach the task of finding out which SDO objects need to be created to make the public API happy at runtime. It helps tremendously to look at the output mapping of your human activities and inspect the tool tips of the nodes below the UIResponse node for type information. As the types are identical between input and output data objects at a certain level, you can at least reuse the type information from the input data object which is already present from the time you created the task instance. You then only need to fill in the values for those fields you will be accessing downstream in the process flow.

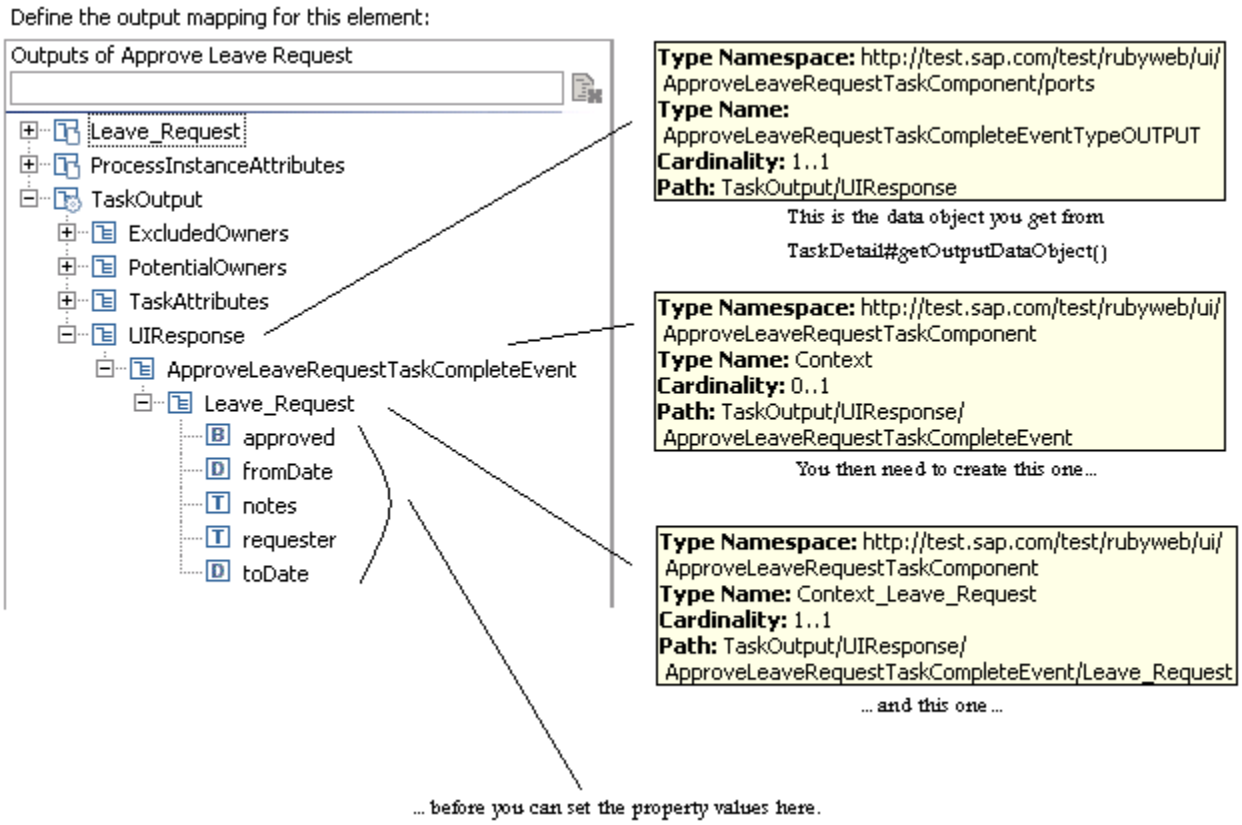

Figure 17: Dealing With Data Objects

7 Rounding Off

From here on you basically know everything you need to build great-looking custom UIs on BPM. Still, a few tips might help rounding things off here and there.

7.1 Adding More Convenience to the BPM Public API

The BPM Public API offers a solid entry point to programmatically accessing a BPM system and managing tasks and processes. It is under constant development and new features are being added with each new public release of NW BPM. Still, there are probably things that you might want to be different. What's more, as you are using Ruby, you might even want the API to feel more Ruby-like. JRuby offers a great way to achieve just that through the use of mix-in inheritance in combination with the JRuby proxy mechanism. The latter wraps each Java object with a corresponding Ruby proxy object. In essence, this allows you to add behavior to the Ruby proxy object by mixing in Ruby modules containing new methods and therefore literally enrich the corresponding Java object. The following picture visualizes this fact:

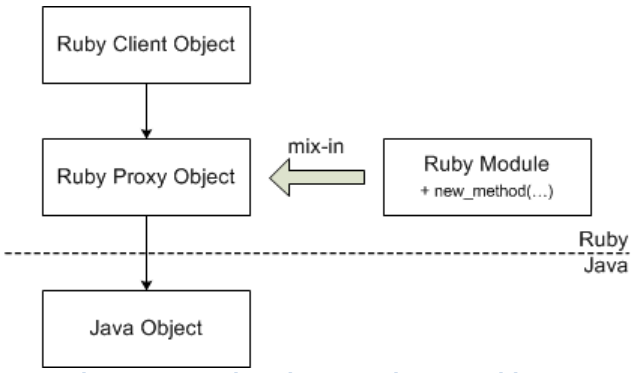

Figure 18: JRuby Mix-Ins and Proxy Objects

You can look at /WebContent/WEB-INF/utils/bpmutils.rb to see how this is being used in the demo code.

8 Conclusion

In this article, we showed how the BPM Public API can be flexibly combined with ground-breaking, new open-source UI technologies to provide for a beautiful, completely customized user experience of your NW BPM processes and composite applications in general. The principles applied can easily be adopted to develop custom user interfaces for a variety of different devices, in particular tablets and mobile phones – all without the need for additional infrastructure.

9 Related Content

SAP NW BPM Resource Center

Ruby Programming Language

JRuby

Sinatra Web Framework

10 Copyright