

BPM Console Reference

1.0.0.GA

Heiko Braun

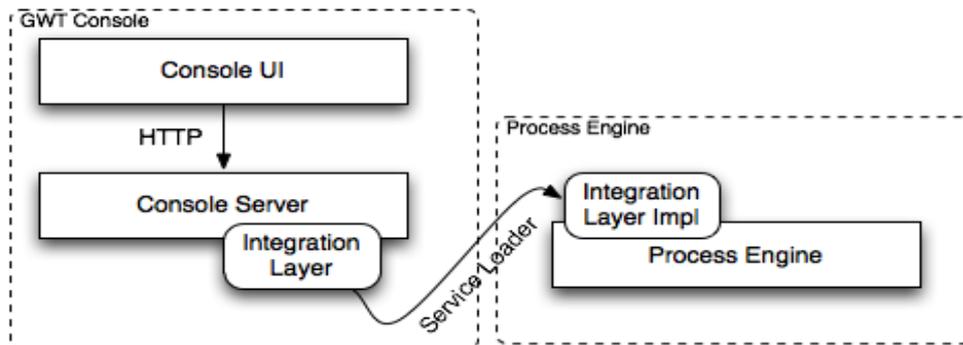
Table of Contents

Technical Overview.....	3
Main components.....	3
Integration with the process engine.....	3
Deployment Artifacts.....	3
Classloading scopes.....	4
Workspace framework.....	4
Workspace API.....	4
Workspace configuration.....	4
Build profiles.....	5
Console server plugins.....	5
Plugin loading.....	5
Management capabilities.....	5
Process Management.....	5
Process life cycle.....	5
Instance life cycle.....	5
Process Activity.....	6
Instance Data.....	6
Process forms.....	6
Task Management.....	6
Users, Groups and identity management.....	6
Task life cycle.....	6
Task forms.....	6
Reporting.....	6
Default reports.....	6
Customizing report templates.....	7
Default reports.....	7
Appendix A: FormDispatcherPlugin.....	7
Default context information.....	7
Dynamic render context	8
Appendix B: Report server.....	8
Console integration.....	8
The BIRT runtime.....	8
Report templates.....	8
Appendix C: Authentication and access.....	8

Technical Overview

Main components

The console consists of three distinct parts: The console UI, the console server and an [integration layer](#). The later decouples the actual process engine from the server module:



The console UI is an AJAX web application that solely uses HTTP to communicate with the sever module. The server module itself, presents a REST facade to the console UI and integrates the actual process engine.

Integration with the process engine

The process engine is decoupled through an integration layer. The integration API is part of the console project, while the actual implementation of that layer resides with the process engine. At runtime the server module uses a [service loader mechanism](#) to access the process engine through the integration layer.

The integration layer allows different process engines to be managed with the same console and prevents changes in the process engine to require changes in the management console.

Deployment Artifacts

Although installation of the console is usually covered when installing the process engine, you might need to know which parts go where, especially when porting the console to a different container.

```
Bonanova:jboss-5.0.0.GA hbraun$ find ./ -name "gwt*"
./server/default/deploy/jbpm/gwt-console-server.war
./server/default/deploy/jbpm/gwt-console.war
./server/default/lib/gwt-console-rpc.jar
./server/default/lib/gwt-console-server-integration.jar
```

A quick scan of an example installation reveals four console related artifacts. Two web applications (the console UI and the console server) as well as two shared libraries: the data model shared between all layers and the integration layer API.

Component	Deployment Artifact
Console UI	gwt-console.war
Console Server	gwt-console-server.war
Domain model	gwt-console-rpc.jar
Integration Layer	gwt-console-server-integration.jar

Classloading scopes

The console UI is completely decoupled from the server since it uses HTTP to access the backend (remember it's AJAX). However the console server and process engine need to share the same classloading scope, otherwise the service loading mechanism doesn't work. The two remaining artifacts, rpc.jar and server-integration.jar should go into a shared parent scope, because they need to be available to every layer.

Workspace framework

What we consider the workspace, is basically the main layout of the console UI, including the main navigation on the left, the header, the message panel at the bottom and the editor pane on the middle. The workspace and its contents are abstracted through a [workspace API](#). The workspace API allows you to add different editors to the workspace. Each editor represents a particular use case or management capability.

The console follows a “one size fits many” approach. Aiming at reuse where applicable and allowing for proprietary extension when needed. The extension points are split into build time extensions, aka plugins to the actual UI and runtime extensions that allow replacement of server side functionality that the console UI relies upon.

NOTE: If you are not familiar with GWT at all, it makes sense to read the [GWT introduction](#) before diving into the following sections.

Workspace API

The workspace API addresses extensions to the console UI itself. A workspace is split into editors that contain views. Each editor provides a navigation to the main menu on the left. At build time the console assembles the workspace based on the plugins available as maven dependencies. In order to extend the console UI, you solely need to provide an editor implementation that was build against the workspace API.

Workspace configuration

A simple property file controls the actual workspace composition. It's part of the build profile chosen a contains a list of editors that should be included when assembling the final web application.

```
# the default workspace.cfg

org.jboss.bpm.console.client.SettingsEditor
org.jboss.bpm.console.client.process.ProcessEditor
org.jboss.bpm.console.client.task.TaskEditor
org.jboss.bpm.console.client.report.ReportEditor
org.jboss.bpm.console.client.engine.EngineEditor
```

Build profiles

Customization of the console happens at build time through the use of profiles. It's actually maven profiles that are triggered by a system property:

```
mvn -Dconsole.profile=jbpm
```

If you are looking into extending the console, then a custom build profile would be the right point to start. It does not only specify the workspace configuration, but also allows to pull in arbitrary dependencies required for the editors you provide. I.e. custom editor implementations that are published as maven artifacts.

Console server plugins

The server module provides hooks to replace or even remove certain functionality with regard to the process engine itself. Unlike the [integration API](#) these plugins are expected not to be available. The console knows about the plugins and can hide certain functionality based on the plugin availability. Any of the default BPM management functionality that we expected to be customized or not available in all cases has been modeled as a server module plugin.

Plugin loading

Server module plugins are loaded using the [service loader mechanism](#) and thus can easily be replaced by simply exchanging the jar files available to the server module classpath. We don't go into the details here, but a good example is the [FormDispatcherPlugin](#).

Management capabilities

Keep in mind that the console was [designed to be extensible](#). Although it ships with a set default editors for managing a process engine, it's very likely that it doesn't match all your requirements. Reuse where applicable and extend where necessary. However that doesn't mean the default management capabilities are cast into stone. Active discussion and feedback should help to improve the out of the box experience over time.

Process Management

The process management editor allows to manage both process definitions and process instances.

Process life cycle

Process definitions can be suspended and resumed, which depends on the state of the deployment they are associated with. For further explanations on the lifecycle of deployment and process entities please refer to the process engine user guide.

Instance life cycle

Process instances can be started, terminated or deleted. Termination means the instance will be ended, while deletion will force removal of the instance and all related entities, i.e. history information.

Process Activity

If the process deployment does contain a process diagram, you are able to access a graphical representation of the instance activity. Internally this is handled by the [GraphViewer](#) plugin.

Instance Data

Inspecting current process state (aka variables) is available in a read-only mode.

Process forms

If the process is associated with a form that should be used to start new instances, the console will request your input based on the form template attached to the deployment. Similar to the GraphViewer this option will only be available if the deployment contains a form template and the process references it. For further information please consult the process engine user guide.

Task Management

The task editor provides access to group and user task lists based on [currently authenticated user](#). You have the ability to claim and assign tasks as well as to provide data to the process through the use of task forms.

Users, Groups and identity management

The actual identity management is controlled by the process engine and the identity management solution it uses. Querying for tasks relies on the current principal. That means the group tasks presents task available to either current principal or one of the groups it belongs to.

Task life cycle

Currently the console uses a simplified task life cycle model. A task can either be open, or assigned to somebody. Open tasks are available to a group of users which then can claim the task and hence assign it to themselves. Releasing a task means “opening” it again.

Task forms

One of the main use cases for task is to either review or provide process data. In both cases a task will be associated with a process instance and give access to its data in a read-only or read-write

fashion. Providing task forms to the console is delegated to the [FormDispatcher](#) plugin. Task forms will only be available if the FormDispatcher can resolve a form template related to a particular task instance.

Reporting

The reporting capabilities are based on [BIRT](#). All the console provides is a report server component for rendering report templates and integration with the actual UI. The process engine provides templates which you can either lean on, extend from or even replace them at all. The basic idea is that any kind report will require customization anyway, therefore we only provide the integration and out-of-box templates to get you started.

Default reports

The default reports are split a general system overview and process specific reports. While the later should allow you to analyze a specific process with regard to it's execution characteristics the system overview is more intended to spot derivations and exceptional situations at a glance. However keep in mind that this intended to be customized and enriched with your applications domain data.

Customizing report templates

The process engine provides a set of default BIRT templates that can be customized using the [BIRT report design tools](#). Appendix B contains instructions on how to [deploy them to BIRT runtime](#).

Default reports

Report	Template Name
General System Overview	overall_activity.rptdesign
Process Activity Summary	process_summary.rptdesign

Appendix A: FormDispatcherPlugin

The default form plugin implementation leverages the [freemarker templating library](#). It builds on the following constraints:

- Templates need to be suffixed *.ftl and be included with the deployment
- HTML forms need to provide the correct enctype: "multipart/form-data"
- Form field names become process variables names and vice versa
- A reserved field name for signaling execution upon task completion: "outcome"

Default context information

The form render context provides default context information useful for rendering templates: Currently that's `${form}` and `${outcome}`. These are used to provide runtime information to the form rendering.

Let's do an example:

```
<h2>Your employee would like to go on vacation</h2>

<form action="{form.action}"
  method="POST" enctype="multipart/form-data">    (1)

Number of days: ${number_of_days}<br/>    (2)

<hr>
In case you reject, please provide a reason:<br/>
<input type="textarea" name="reason"/><br/>    (3)

<#list outcome.values as transition>    (4)
  <input type="submit" name="outcome" value="{transition}">    (5)
</#list></form>
```

- 1) Accessing form action dynamically: 'form.action'
- 2) Referencing a process variables named 'number_of_days'
- 3) Create a new process variable named 'reason'
- 4) Access transitions dynamically: 'outcome.values'
- 5) Reserved field name to trigger execution: 'outcome'

Dynamic render context

As described above, some properties are provided at runtime, i.e. the actual form action parameter or the available outcomes (aka transitions). Some of those are required and cannot be derived at design time (the form action) others are just convenience (the available outcomes).

Appendix B: Report server

The console server integrates the BIRT runtime for providing reports on the process engine activity history. Reporting within the console actually breaks down into three pieces: Integration with console, integration of the BIRT runtime itself and the actual report templates.

Console integration

The actual console integration is covered by the default report editor and shouldn't be much of an issue.

The BIRT runtime

The BIRT runtime will usually installed along with the process engine or can be retrieved from the [BIRT website](#). It needs to be installed at a particular location that is expected by console server. Under JBoss it uses the server data directory for both accessing the templates and storing the results.

```
Bonanova:jboss-5.0.0.GA hbraun$ ll server/default/data/birt/
```

```
hbraun  staff      340 Jul  9 12:57 ReportEngine
hbraun  staff      170 Jul  9 12:58 output
hbraun  staff  150899 Jul  9 12:53 overall_activity.rptdesign
hbraun  staff      669 Jul  9 12:53 process_summary.rptconfig
hbraun  staff  153602 Jul  9 12:53 process_summary.rptdesign
```

Report templates

The report templates are provided by the process engine. However if you plan to [customize the default report templates](#), the BIRT data directory would be the place to put them.

Appendix C: Authentication and access

The console currently uses HTTP basic auth to access the console server. The server module itself is connected to JAAS domain, just like in any other web application. Currently there is access control implemented in the console UI.