

1 SmartFrog Web Components

1.1.1

The SmartFrog Web components provide a set of components for defining web applications (WAR applications), Enterprise (EAR) applications and Servlets for deployment onto a Java Application Server such as JBoss, WebLogic or Apache Geronmio, or a Java Web Server such as Jetty or Tomcat. It also includes a liveness page component, that can retrieve an HTTP page from a server as its liveness test, integrating web page visibility with application liveness.

These components are still a work in progress, with the consequent warning that all interfaces and component descriptions are unstable.

The goal is to enable abstract descriptions of WAR, EAR and servlet applications which can then be deployed directly to any application server. To this end, the descriptions of application and web servers are abstract -they cannot be instantiated. A non-abstract implementation of the server component descriptions is needed to deploy WAR/EAR/Servlet applications.

At the time of writing, there are the following implementations

- Deploy-by-copy, that can deploy into any application server that supports deploy-by-copy deployment.
- Jetty, in the separate [sf-jetty](#) JAR
- Tomcat, in in the separate [sf-tomcat](#) JAR
- Cargo, in the separate [sf-cargo](#) JAR

Of these, Jetty is the most stable, and is included in current distributions. Deploy-by-copy supports deployment into any mainstream application server, but does not address starting and stopping the server. A separate Java component is usually used for that task.

1.2 Architecture

An [ApplicationServer](#) represents the web server or app server into which Web applications may be deployed. Different application server implementations extend this template to provide support for deploying such items. The specific types of deployable [ApplicationServerContext](#) are:

- WAR file: a web application. All implementations should support this. Declared in a [webApplication](#) component.
- EAR file: an Enterprise application, which requires a full J2EE system.
- SAR file: (currently) JBoss specific service application.
- Servlet Context: a context into which individual servlets can be declared. This is not as self contained as the others, but permits much more low-level configuration within SmartFrog deployment descriptors themselves, so eliminating much of the need for WAR file and [web.xml](#) configuration descriptor. Declared in a [ServletContext](#) component. Items that may be deployed within such a context must extend [ServletContextComponent](#), the two standard options being [Servlet](#) and [MimeType](#). These components have a [context](#) attribute that must be bound to the context into which they are to be deployed.

Other extensions of [ApplicationServerContext](#) may also be created and deployed into a server.

Every [ApplicationServerContext](#) must be bound to an [ApplicationServer](#); this is done by setting the [server](#) attribute to a [LAZY](#) reference to the specific server.

When dynamically deploying applications to an application server, the full URL to the specific servlet, SOAP endpoint or other part of the system may be unknown until deployment time. For this reason, every component that can be deployed has an [absolutePath](#) attribute. This is set at run-time to the full path of the specific

component.

1.3 Implementation

The `ApplicationServer` component is abstract; specific implementations are needed for every supported application server. These implementations provide implementation-specific classes to deploy components within the server. Each component that be deployed within an application server, and each servlet deployed within a `ServletContext`, must have a matching *delegate* class provided by the server-specific implementation. The portable components used in the deployment descriptors retrieve these delegate implementations from the server to which they are bound, and the heavy lifting of deployment is then delegated to the implementations.

The effect of this design is that one can describe WAR, EAR, SAR files and servlet applications within deployment descriptors, and deploy these applications to different application servers, merely binding them to different target systems.

1.4 Components

1.4.1 ApplicationServer

This is an abstract type that different application servers must extend/implement.

<i>Attribute Name</i>	<i>Description</i>	<i>Optional/Mandatory</i>
<code>supportsWAR</code>	Boolean that indicates whether or not this server supports WAR files (i.e .Web applications)	Mandatory
<code>supportsServletContext</code>	Boolean that indicates whether or not this server supports servlet contexts	Mandatory
<code>supportsEAR</code>	Boolean that indicates whether or not this server supports EAR files.	Mandatory
<code>supportsSAR</code>	Boolean that indicates whether or not this server supports SAR files.	Mandatory

1.4.2 ApplicationServerContext

Base template/interface for anything that can be deployed into an application server

<i>Attribute Name</i>	<i>Description</i>	<i>Optional/Mandatory</i>
<code>server</code>	LAZY reference to the server to which this context is bound	Mandatory

Run-time properties

Attribute Name	Description
<code>absolutePath</code>	Base path of this context within the application server. This is the path exposed as part of the URL

1.4.3 WebApplication

Declares a web application

Attribute Name	Description	Optional/Mandatory
<code>contextPath</code>	Context Path for the web application	Mandatory
<code>warFile</code>	Path of the web application	Mandatory
<code>resourceId</code>	Resource id for the web application	Optional
<code>server</code>	Server to which this web application is attached	Mandatory

Run-time properties

Attribute Name	Description
<code>absolutePath</code>	Base path of this context

1.4.4 ServletContext

A servlet context is an application context into which servlets can be deployed. It is more low-level than a WAR file, but permits very tight integration with SmartFrog, and detailed configuration of deployed servlets within a SmartFrog deployment descriptor.

Attribute Name	Description	Optional/Mandatory
<code>contextPath</code>	Context Path for the servlet context	Mandatory
<code>resourceBase</code>	Directory containing static files and JSP pages base for the servlet context	Mandatory
<code>server</code>	Server to which this servlet context is attached	Mandatory
<code>classPath</code>	Classpath for the servlet context	Optional

Run-time properties

Attribute Name	Description
<code>ipaddr</code>	IP Address of the context (currently determined from the network card/address that the SmartFrog daemon is listening on)
<code>absolutePath</code>	Base path of this context

2 Health Checking Components

2.1.1 LivenessPage

The liveness page retrieves a web page on a liveness check. If the retrieval fails, an exception is thrown.

```
LivenessPageSchema extends Schema {
  //either
  //full URL to the page
  url extends OptionalString;
  //or
  //port of the page; default 80
  port extends OptionalInteger;
  //host of the page; default 127.0.0.1
  host extends OptionalString;
  //protocol, default http
  protocol extends OptionalString;
  //page
  page extends OptionalString;
  //query list of things that get turned into queries -without escaping.
  queries extends OptionalVector;

  //and any of

  //flag to set if you want any error text fetched from
  //the remote site. This is good for diagnostics.
  fetchErrorText extends OptionalBoolean;
  //response code below which the fetch is an error
  minimumResponseCode extends OptionalInteger;
  //error code above which an error has occurred,
  //default is 299.
  maximumResponseCode extends OptionalInteger;
  //flag to follow redirects
  followRedirects extends OptionalBoolean;
  //check frequency. This is the number of pings between checks
  //and so lets us probe less often than normal. default=1
  checkFrequency extends OptionalInteger;

  //flag to say that the check is on/off; useful during development
  enabled extends OptionalBoolean;
}
```

The liveness page component does not deploy anything, but whenever a liveness check is made of it, it will attempt to synchronously download a page from the web server.

The component can either be configured with a full URL to a page, or by specifically configuring the hostname, port, protocol and page of a URL, along with an optional vector of query strings.

Attribute Name	Description	Optional/Mandatory
url	Complete URL to retrieve. e.g. "http://smartfrog.org/"	Mandatory unless page is set.
port	Port to probe. Only used if url is not set.	Optional-default=80
host	Host to probe. Only used if url is not set.	Optional -default "127.0.0.1"
protocol	Protocol to use -anything supported by java.net.URL is allowed. Only used if url is not set.	Optional -default="http"
page	Page under a host to probe. Only used if url is not set.	Mandatory unless url is set

Attribute Name	Description	Optional/Mandatory
queries	A vector of name-value pairs that are used to build a query string. For example, ["size","200","owner",LAZY owner"] could be turned into the query string "?size=200&owner=something", assuming that LAZY owner resolved to "something". There is currently NO ESCAPING of strings in this query.	Optional, default=nu11
fetchErrorText	Boolean flag to indicate that the response text should be received when there is an error.	Optional -default=true
minimumResponseCode	These two values define the range of HTTP response codes that constitute a successful fetch. By default, the range 200<=299 is deemed a success	Optional -default=200
maximumResponseCode		Optional -default=299
mimeType	A list of mime types that are allowed for the response. e.g ["text/html","text/xml"]	Optional.
followRedirects	Boolean that controls whether redirects (3XX codes) should be followed	Optional
checkFrequency	Frequency of checking the page; allows HTTP pages to be checked less often than other liveness pages. This is useful if the check is slow or CPU-intensive on the server.	Optional -default=1
Enabled	Boolean to enable/disable checking	Optional -default=true

Here are the default options:

```

LivenessPage extends WorkflowPrim {
  livenessPageSchema extends LivenessPageSchema;
  sfClass "org.smartfrog.services.www.LivenessPageComponent";
  fetchErrorText true;
  maximumResponseCode 299;
  minimumResponseCode 200;
  followRedirects true;
  checkFrequency 1000;
  enabled true;
  checkOnStartup false;
  checkOnLiveness true;
  host "127.0.0.1";
  protocol "http";
  port 80;
  page "/";
}

```

The default options probe the local host at `http://127.0.0.1:80/`, checking on liveness tests but not startup, following redirects, and failing if the ultimate HTTP response was not in the range 200-299.

Note that `"localhost"`, `"127.0.0.1"`, and the IPv6 equivalent `"::1/128"` all bind to loopback address of a system, not to any external interface of a server. As loopback interfaces are normally not firewall protected, probing for pages over the loopback address does not verify that a firewall is not interfering with remote access to a page. To verify that a page is remotely accessible, the best techniques are either deploy the checker to a remote host, or to relay the request via an HTTP proxy. This component uses the current proxy of the JVM, so

will go via proxy if the JVM is so configured.

Future enhancements, for which contributions from interested users are encouraged, include:

- asynchronous retrieval of pages
- Post-download validation. This could be declaring a minimum size of a page, or a required mime type
- XML validation and parsing of a downloaded document. At its most sophisticated, XPath expressions could be resolved against the downloaded content and turned into resolvable attributes. This would enable run-time extraction of content from XML messages.
- POST support and url-forms-encoded encoding of the query string.
- escaping of GET query parameters.

2.1.2 Examples

Here is a liveness page that checks for a server being present on port 8080.

```
TomcatLivenessPage extends LivenessPage {
    port 8080;
}
```

The `LivenessPage` component implements the `Condition` interface, with `condition.evaluate()` returning true if the remote URL can be retrieved and is considered valid. This enables it to be used in any complex test that the conditional workflow components can describe. For example, the test for a server on port 8080 could be negated in a condition, which would be true when there was no running server:

```
NoRunningServer extends Not {
    condition extends TomcatLivenessPage;
}
```

This specific condition is actually used in SmartFrog's tomcat web server tests, skipping the tests if an application server is already running on the target port.

2.2 WaitForPage

The `waitForPage` component extends Liveness page to block until a page is available. This can be used in workflows, to block a successor operation until the liveness condition is met.

```
WaitForPage extends LivenessPage {
    WaitForPageSchema extends Schema {
        //timeout in milliseconds
        timeout extends Integer;
    }
    sfClass "org.smartfrog.services.www.WaitForPageImpl";
    checkFrequency 500;
    fetchErrorText false;
    checkOnLiveness false;
    sfShouldTerminate true;
}
```

A timeout controls how long the component should wait before deciding that a web server is not going to become available, in which case the component terminates with an error. Here is an example use, blocking until Tomcat listens on port 8080

```
WaitForTomcat extends WaitForPage {
    port 8080;
    timeout 30000;
}
```

2.3 ServletContext Components

These are components that are only valid when bound to a servlet context.

2.3.1 Servlet

Declares a servlet that handles requests sent to a specific part of the servlet context's tree, or to all files that match a specific mapping pattern.

Attribute Name	Description	Optional/Mandatory
name	Servlet name	Mandatory
pathSpec	Path specification: what URLs to bind to this servlet	Mandatory
className	Class name for the servlet	Mandatory
context	Reference to a servlet context e.g. LAZY PARENT:servlets	Mandatory
initParams	Vector of Initialisation parameters	Optional
initOrder	Order for servlet initialisation. <0 means "on demand"; servlets with a value >0 are executed in order	Optional, default is -1
mappings	Mappings (e.g. ["*.jsp", "*.jspx"])	Optional

Run-time properties

Attribute Name	Description
absolutePath	Base path of this context. This is the concatenation of the servlet context which the servlet is deployed into with the servlet's pathSpec , with any trailing * stripped off the end of the latter. It is not valid for filename based mappings such as pathSpec <code>"*.do"</code> ;

A servlet can be declared unbound to a specific context

```
#include "/org/smartfrog/services/www/components.sf"

/**
 * Base servlet for alpine
 */
AlpineServlet extends Servlet {
    name "alpine";
    contextPath "/alpine";
    className "org.smartfrog.projects.alpine.http.SoapPostServlet";
}
```

This servlet has to be bound to a specific servlet context by setting its [context](#) attribute.

2.3.2 MimeType

Declares a mime type within a servlet context. The mime type is removed when the application is undeployed.

Attribute Name	Description	Optional/Mandatory
extension	Extension, e.g. <code>"*.html"</code>	Mandatory

<i>Attribute Name</i>	<i>Description</i>	<i>Optional/Mandatory</i>
type	Mime type, e.g. "text/html"	Mandatory
context	Reference to a servlet context e.g. LAZY PARENT:servlets	Mandatory

3 Examples

3.1 The Alpine SOAP Stack under Jetty

Here is an example from the Alpine SOAP stack. It defines a compound component, [AlpineServerOnJetty](#), which starts Jetty listening on a defined port, and deploys the alpine servlet in a servlet context underneath.

```
AlpineServerOnJetty extends Compound {
    port TBD;
    home TBD;

    jettyServer extends CoreJettyServer {
        jettyhome LAZY PARENT:home;
        port PARENT:port;
        server "alpine";
    }

    listener extends JettyListener {
        listenerPort port;
        server LAZY jettyServer;
    }

    servlets extends ServletContext {
        resourceBase LAZY home;
        contextPath "/";
        server LAZY jettyServer;
    }

    /**
     * this is the servlet to which endpoints need to be bound
     */
    alpine extends AlpineServlet {
        context LAZY servlets;
    }
}
```

This component is still abstract; it needs to be given a port number and a home directory before deployment.

3.2 Example: An alpine server

```
DeployServer extends AlpineServerOnJetty {
    home LAZY PROPERTY java.io.tmpdir;
    port 8080;
    hostname LAZY servlets:ipaddr;
    livenessEnabled true;
}
```



```
//fetch the happy page every so often, just to see that we are happy
localliveness extends LivenessPage {
    enabled PARENT:livenessEnabled;
    host "localhost";
    port PARENT:port;
    page "happy";
    checkFrequency 1;
}

externaliveness extends localliveness {
    host LAZY PARENT:hostname;
}
}
```

This definition extends the previous example, making it something that can be deployed. It adds:

1. a port number, 8080
2. a LAZY binding to the temp directory of the system. This means that it will serve up the entire directory, which is a bit of a security risk. One of the file components should be used to create a subdirectory instead. Note that a LAZY PROPERTY is used; this ensures that the temp directory value on the target system is picked up, not that of the system used to initiate the deployment.
3. A liveness page that is bound to the happy page of the system. This is a page that is backed by code (JSP or servlet) that evaluates the local system state, and returns an error code (and ideally, human-readable diagnostics) when not. The `localliveness` component is bound to the local host, so goes through the loopback interface of the system, bypassing any firewall.
4. An `externaliveness` component that extends the `localliveness` page by connecting to the IP address of the parent's `hostname` attribute.