# Enabling Web Based Management of SmartFrog Systems

Derek Mortimer

*September 2006*

## Aims and Objectives

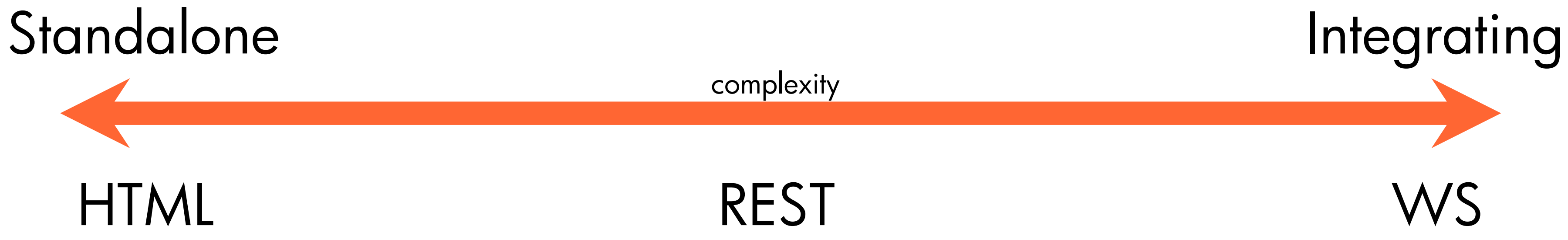
- Enable the management of a running SmartFrog system over the Internet using well known mechanisms

- The system should:
  - Provide, minimally, the functionality within the SmartFrog Management Console in terms of resource management
  - Require no modification of existing SmartFrog systems in order to enable management
  - Be sufficiently extensible and customisable

- Three main areas of focus initially defined

## Initial Suggestions

- Three initial areas of focus
  - HTML Web Based, HTML form interaction
  - REST Web Based, HTTP+XML interaction
  - Web Services Based, XML+SOAP+WSDL interaction



  - For simplicity and ubiquity, an HTML web based interface was opted for an initial prototype.

## Issues with HTML interface development

- Became apparent the HTML interface was a very open ended project
- For AJAX functionality, XML representations of the system resources would need to be readily available, typical REST systems represent all of their resources as XML documents
- A REST based interface had very concrete definitions in terms of functionality and was deemed to be more useful to SmartFrog in terms of integration

- As a result, design focus was switched to the REST Web Based Interface

## REST Principles

- REST (Representational State Transfer) dictates that a system be modelled in terms of its resources and not its operations
- Allows us to create a direct mapping of a SmartFrog tree onto resources referenced by unique URIs
- Resources are acted upon in a semantically correct manner using only a subset of the standard HTTP 1.1 requests (namely, GET, PUT*, POST* and DELETE)
- These methods allow us to create, modify and remove resources within a SmartFrog tree

***** Due to their similarity, the meaning of PUT and POST are taken from the ATOM specification, to update/create and create respectively.***

## SmartFrog REST URI Scheme

- The final URI format is as follows:

/<sfHost>/<sfPort>/<resourcePath>

/localhost/3800/sfDefault/sfProcessName

- Prior URI schemes included an entry for resource type between the port and resource path. This, however, meant the client required prior knowledge of the resource type. Including it in the representation of a resource allows for greater flexibility later on and makes for a more intuitive URI scheme.

## XML Representations

- GET requests within a REST system must yield meaningful representations of the resources being represented. It was decided to keep these representations as general as possible. Everything within a SmartFrog system is represented by a <resource> node within an XML document.

- In addition to representations of the resources, XML documents would be needed to report results and exceptions to the client. These come in the form of <result> and <exception> tags.

# Components Representation

```
<resource name="rootProcess" type="component"
  class="org.smartfrog.sfcore.processcompound.ProcessCompoundImpl"
  href="http://rest.service/localhost/3800/">

    <!-- example of each type of attribute -->
    <resource name="childApplication" type="component"
      href="http://rest.service/localhost/3800/childApplication" />
    <resource name="myDescription" type="description"
      href="http://rest.service/localhost/3800/myDescription" />
    <resource name="sfHost" type="attribute"
      href="http://rest.service/localhost/3800/sfHost" />
    <resource name="rootProcess" type="reference"
      href="http://rest.service/localhost/3800/rootProcess" />

</resource>
```

## Components Representation

- The XML representation for a component description is identical to that of the deployed component representation apart from the *type* attribute in the root element.

- The reason for this is that component description objects are traversable in exactly the same manner as components but the client is still provided with a simple mechanism for differentiating between the two.

# Attributes Representation

- A GET request performed on any object within a SmartFrog system that is not a traversable object will yield the following XML response:

```
<resource name="sfProcessName" type="attribute"
  class="java.lang.String"
  href="http://rest.service/localhost/3800/sfProcessName">
    rootProcess
</resource>
```

- The contents of the tag is simply the result of the target resources *toString* method.

## Handling of Reference Objects

- If the target named in the URI is an instance of a SmartFrog Reference (i.e. a *Reference* object) then the action taken depends upon the HTTP method being invoked and an optional client-passed parameter.

- If the request is a GET, the default behaviour is that the reference will actually be resolved transparently to the client. This behaviour can be overridden through the use of the *followReferences=false* querystring parameter in which case a response identical to the previous slide will be sent with a different *type* attribute in the root element.

- For PUT, POST and DELETE requests, the *Reference* object itself will always be acted upon.

## Operation Results

- A successful PUT, POST or DELETE operation will yield an XML response in the following format:

```
<result>
    <code>DELETED</code>
    <message>
        The specified component was DELETED successfully.
    </message>
</result>
```

- The two portions of the result are intended to be machine-friendly and human-friendly respectively.

## Exceptions

- An exception will yield an XML response in the following format:

```
<exception
    class=“fully.qualified.ExceptionName”
    cause=“fully.qualified.CauseExceptionName”>
        <message>An exception was thrown by the system</message>
        <trace>
            fully.qualified.ExceptionName (in foo.bar())…
        </trace>
</exception>
```

- Special exceptions may warrant their own HTTP status code but all others will yield an HTTP 500 Internal Server Error response

## Request XML Documents

- In order to facilitate the creation of new resources via PUT and POST, some form of XML request needs to be submitted to the server. This request takes the following format:

```
<description type="<TYPE>" language="sf">
<![CDATA[
        // SmartFrog Parseable Content
]]>
</description>
```

- The *type* attribute is allowed to be *component*, *description*, *attribute* or *reference* and is simply used to refine the parsing method called within the system.

## Enabling Management of Existing Systems

- In order to enable management of all resources within a system, we must think about what makes a resource 'Restful'.
  - The ability to modify the resource through a GET, PUT, POST or DELETE request.
  - The ability to perform a GET request implies the resource must be able to export an XML representation of itself.
- These points can be used to define an interface that manageable resources must implement in order to be operated upon by the REST system.

## The Restful Interface

Restful
<<Interface>>

```
public void doGet(HttpRequest, HttpResponse)
public void doPut(HttpRequest, HttpResponse)
public void doPost(HttpRequest, HttpResponse)
public void doDelete(HttpRequest, HttpResponse)
public Document getXMLRepresentation()
```

Any object within the SmartFrog tree that implements this interface becomes directly manageable by the REST system.

But what about resources that don't export this interface?

## REST Wrapper Factory class

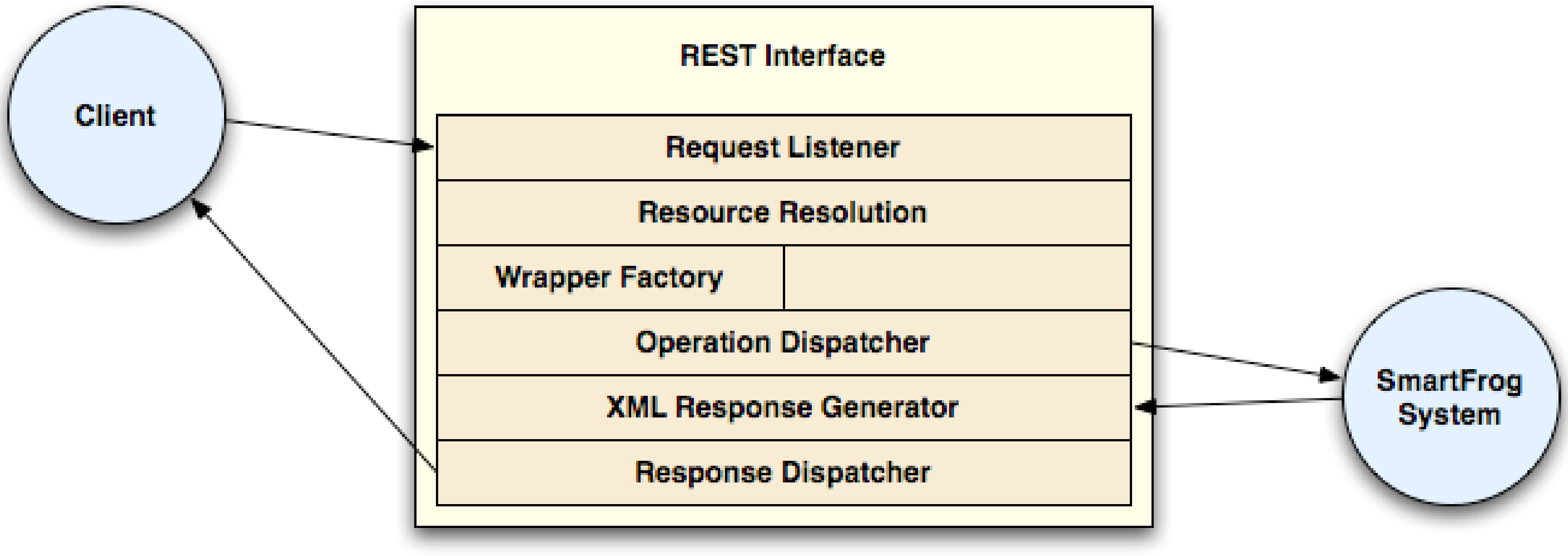
- In order to facilitate management of existing resources without the need to alter them, a factory class was created that, when discovering a resource, will package it up in a wrapper object which implements the Restful interface and will act on behalf of the subject contained within.

- Initial implementation contains two wrapper classes, one to deal with Attributes/References and the other to deal with Component/Descriptions

## Extensibility

- Because the operations are defined in terms of the Restful interface, it is entirely feasible to create a new component type which directly implements the interface.
- This would allow you full control over the action taken by any HTTP request such as customising the XML response generated for the client.
- This is also the reason the resource type is not explicitly stated within the URI, everything in the system is deemed a "SmartFrog Resource" with further differentiation left to the user through XML processing

# System Layout



The execution path of the application flows from top to bottom within the REST Interface.

## Development/Known Issues

- Un-serialisable HttpServletRequest and HttpServletResponse classes
- Missing class stubs even when components descend from known parents (e.g. Prim or Compound)
- Unserialisable components within a SmartFrog tree that cannot be exported or inspected across a network
- These give rise to:

## Future Possibilities

- Schema based validation of incoming and outgoing XML documents

- Possible use of multiple co-operative REST interfaces placed on known SmartFrog daemons to ensure representations of all resources could be exported as the resource reference would always be handled locally

- Construction of a Web Service interface onto a SmartFrog system leveraging the experience gathered throughout this project. Web Services lend themselves to the idea of a set of co-operative endpoints on multiple SmartFrog daemons invent