Xtivia Services Framework (XSF)

Programming Guide

Version 1.1.0

October 2015

For questions or comments please email *xsf@xtivia.com*

Release History

1.0.0	October,2015	Initial Release
1.1.0	October, 2015	Upgraded Jackson library version to 2.6.3 Added support for @Route annotations on ICommand methods

Table of Contents

Introduction

XSF is a framework that XTIVIA has created (and used in multiple client engagements) that enables the rapid development of custom REST services in Liferay.

As we know the current trend in web application development is toward Single Page Applications (SPAs), where the majority of application functionality is implemented in JavaScript that executes in the browser. SPAs then communicate with REST APIs that act as proxies for a wide range of enterprise services and data stores. XSF provides a means by which we can rapidly develop these kinds of REST services in a Liferay environment.

While it's true that one can create "resource-based" services via the *serveResource()* method and "resource IDs" in a portlet, this approach is very different from the traditional annotation-based approaches used in JAX-RS/Jersey/Spring based REST development. Further, the *"serveResource()"* approach becomes cumbersome when all of the resulting REST services for an application are spread across multiple portlet implementations and resulting WARs. What is preferable is to find a way to build a singular, annotation-based set of REST services that can be leveraged by a number of different portlet applications but deployed and managed as a single artifact.

Finally, its important that any REST services developed for Liferay are able to leverage the Liferay roles and permissions model in order to control access to the services. The access control model in Liferay is very rich and provides a high-level of both control and management over access to elements of the User Interface (UI). With XSF we can leverage this same model to control whether or not those same permissions and access control mechanisms are also applied to access to REST services. This becomes a critical element for SPA-based solutions in Liferay.

To summarize then these were the key goals for XSF (and any resulting REST services that are developed using it):

- Support the development of individual REST endpoints as simple annotated Java objects (POJOs).

- Enable the testing of these POJO-based services in a basic JUnit environment that does not require a full Liferay (or even web container) environment for unit testing.

- Provide a declarative mechanism for defining "routes" to each of the REST endpoints that maps the URI and HTTP method to a particular endpoint implementation (POJO). For readers who have used existing web frameworks such as Rails, Sinatra, Grails, Ratpack, Django, etc. this should be a familiar concept.

- Be Liferay-aware. REST services have particular value in a Liferay-server environment when they can access the logged-in Liferay user and leverage Liferay APIs such as roles and permission-checking, etc. Then the same permissions used to control access to use interface elements can also be used to control access to services by the UI.

- Leverage Liferay SDKs and hot-deployment support. In effect this means that the services are contained in a Liferay portlet plugin WAR. The benefit is that this provides the maximum amount of support in terms of Liferay features and also enables services to be hot-deployed during development.

Hello World, XSF Style

To show how easy it is to create new REST services using XSF we will walk through the steps required to build a first “Hello World” service using the framework. To do this we need to execute the following steps:

- Create a new Maven project using the available XSF archetype (details for this are described later).
- Create a new Java package named *xsf.samples* and add the following simple class to that package:

```
package xsf.samples;
[imports omitted for brevity]

@Route(uri="/hello/world/{last}/{first}", method="GET")
public class HelloWorldCommand implements ICommand {

  @Override
  public CommandResult execute(IContext context) {

    Map<String,String> data = new HashMap<String,String>();

    //inputs from path parameters
    String firstName = context.find("first");
    String lastName = context.find("last");
    data.put("first_name", firstName);
    data.put("last_name", lastName);
    return new CommandResult(data);
  }
}
```

- Edit pom.xml to set the *liferay.home* property to point to your locally installed Liferay server or bundle.
- Run “***mvn install***”.
- After the build completes and WAR deployment to Liferay open your browser and enter *http://localhost:8080/delegate/xsfapp/hello/world/bloggs/joe*. You should see the following JSON returned:

```
{
   "succeeded":true,
   "data" : {"first_name":"Joe", "last_name":"Bloggs"},
   "message": ""
}
```

We will revisit this example later in more detail but notice how easy it was to create the REST service, have the service emit JSON with no Java-to-JSON work required on our part, specify the desired endpoint and define path parameters via a simple Java annotation (@Route), and use path parameters from the URL via simple map-like access in the provided context object.

Installation and Setup

XSF is a framework based on Spring MVC that executes within a Liferay portlet plugin. XSF is distributed in two parts (both available via Maven Central):

- A JAR file containing the XSF library code (i.e., Spring MVC controller,framework classes, annotations, and other code elements) .
- A Maven archetype used to build a sample XSF application (WAR) that includes all required Liferay configuration files, and sample service implementations (including all of the samples from this document). The archetype generates a project (POM) that references the XSF JAR artifact described above.

Regardless of whether you generate an XSF project using the Maven archetype or edit the provided sample project, the ultimate artifact that is created by the build process is a Liferay plugin WAR file that can be (hot) deployed to Liferay using the standard deployment approach.

Using the XSF JAR

To use the XSF JAR artifact in custom build configurations you should add the dependency below your application POM. Note that this approach requires you to set up and configure the Spring context and Liferay deployment descriptor files manually (thus the use of the Maven archetype for starting new XSF applications is strongly recommended.)

```
<dependency>
  <groupId>com.XTIVIA.tools</groupId>
  <artifactId>xsf</artifactId>
  <version>${xsf.version}</version>
</dependency>
```

To use the XSF Maven archetype to create a new XSF project you can execute a command similar to the following:

```
mvn -X archetype:generate \ -DarchetypeGroupId=com.xtivia.tools \
-DarchetypeArtifactId=xsf_sample_app-archetype \
-DarchetypeVersion=1.0.0 \
-DgroupId=com.mycompany
-DartifactId=xsfapp
-Dversion=1.0.0-SNAPSHOT \
-Dpackage=com.mycompany.xsfapp \
-DinteractiveMode=false
```

As a first-time alternative to using the Maven archetype you may want to visit the XSF repository on GitHub (*https://github.com/xtivia/xsf*) where you can download a sample project that represents the output of a prior execution of the archetype above. This is an easy way to begin experimenting with XSF as you can

simply download and unpack the ZIP for this repository, customize the settings in the POM (see below), and then run '***mvn install***' to begin using the sample XSF application with your local instance of Liferay.

Customizing the Maven POM

Once you have generated a new XSF project using the archetype described above there are two Maven POM properties that you will likely want to adjust to adapt your specific Liferay development and deployment environments.

As generated the XSF distribution and artifacts will reference version 6.2.3 of the Maven artifacts for Liferay CE (which are publicly available via Maven Central), so if you are using another version of Liferay you will need to download and install the required Maven artifacts and adjust the supplied POM (or a Maven profile) to change the ***liferay.version*** property.

Optionally, you can also set the ***liferay.home*** property in your POM (or profile) to point to your targeted Liferay server/bundle and then when you execute "***mvn install***" the generated WAR will be copied into into your Liferay server's *deploy* directory (in addition to placing the file in your local Maven repository). Since this is likely only useful in development environments this property is disabled by default and must be explicitly enabled/set.

Using the XSF Samples

To validate that your distribution of the XSF sample project is property installed and functional, set the two properties described above to match your local environment, then open a terminal session in the XSF distribution and type "***mvn install***". You will see a number of messages in your terminal session as the sample application compiles and then deploys the XSF services portlet into your targeted Liferay installation.

If the build is successful you can then validate that the XSF samples are functional by opening up a browser and entering *http://localhost:8080/delegate/xsfapp/hello/world/bloggs/joe*. This will invoke one of the sample services provided with XSF and should result in the following JSON being returned to your browser:

```
{
    "succeeded":true,
    "data" : {"first_name":"joe", "last_name":"bloggs"},
    "message": ""
}
```

This example is fully described later in the document but receiving the JSON above is an indicator that your XSF installation is functional and ready for use in developing your own custom REST services. Now you can start developing your own custom services in this project and remove the Xtivia "Hello World" samples.

Developing Services in XSF

Hello World in XSF

We will review developing some simple "Hello World" style REST services using XSF. The examples we will discuss are provided as samples in the XSF distribution (located in the *xsf.samples* package). For our first service we will build a simple echo-style service that accepts path parameters from URI segments and returns a simple object to 'echo' those inputs.

Once XSF has been installed in your development environment (see above) developing a new REST service is as simple as creating a Java class, implementing a single method within that class, and then adding an annotation to the class to configure the routing information for the class/service.

XSF uses the well-known "command" design pattern. Each endpoint/service should be implemented in a class that implements the *ICommand* interface, an interface that contains a single method named *execute()*.

The *execute*() method accepts a single "context" parameter that provides access to inputs needed by the command (path parameters, session parameters, query parameters, etc.) and must return an instance of a *CommandResult* object. We will have more to say on the context object later in this article, but for now think of it as a "door" to the execution environment of the command; at runtime this environment is provided by XSF itself, but it can easily be populated by unit tests to fully exercise the functions and error-handing of the command itself.

A *CommandResult* object contains three fields:

- A *succeeded* (boolean) field indicating whether or not the execution of the command succeeded
- A *message* (String) field that can be used to supply more information when the command fails
- A *data* (Object) field that contains the returned Java object (payload). XSF will marshal the entire *CommandResult* into JSON, including the contents of the *data* field. XSF uses the Jackson library for JSON marshalling; the recommended approach is to use simple value objects for marshalling information to/from the service.

So with the information above as a backdrop we can now begin the construction of our echo service. The listing below provides the implementation of this service, which is provided in the XSF distribution in *com.xtivia.xsf.samples.HelloWorldCommand*. This service is invoked with a URL that looks something like *http://localhost:8080/delegate/xsfapp/hello/world/Bloggs/Joe*, where the last two path segments represent a last name and first name respectively.

```
package xsf.samples;
[imports omitted for brevity]

@Route(uri="/hello/world/{last}/{first}", method="GET")
public class HelloWorldCommand implements ICommand {
```

```
  @Override
  public CommandResult execute(IContext context) {

    Map<String,String> data = new HashMap<String,String>();

    //inputs from path parameters
    String firstName = context.find("first");
    String lastName = context.find("last");
    data.put("first_name", firstName);
    data.put("last_name", lastName);
    return new CommandResult(data);
  }
}
```

Let's first examine the @Route annotation at the top of the class. You will see that the *uri* value in this annotation describes the 'route' that is used to invoke this command. It does not include the host information, nor does it include the */delegate/xsfapp* portion of the full URI. For now consider those portions of the overall URI as fixed (although the *xsfapp* element can be changed via the subcontext init-parameter in the WEB-INF/web.xml file).

Also note that the route definition indicates that the last two segments of the URI are the path parameters, namely *last* and *first*. XSF will route any inbound request that matches this URI to this command, and further, XSF will parse the URI for these parameters and make them available to the command's *execute()* method via the supplied context parameter.

We can see our command/service accessing these parameters in the first two statements of the *execute()* method. Invocations are made to the *find()* method of the supplied context; *find()* is a typesafe generic method that will return a value of matching type, or null if a value cannot be found (or is not null but does not match the requested type). You can think of the context as a specialized implementation of the *Map* interface that proxies information from the request, session, parameters and overall Liferay execution environment.

Finally our command needs to construct an object to return as output from the service. In this case we will use a simple *HashMap* object; in subsequent examples we will demonstrate how to return custom Java objects from our services. In all cases our commands only need to set the desired return object into the *data* field of a *CommandResult* object and XSF will handle the marshalling to JSON!

We simply insert the values we originally received as inputs (with error handling intentionally not present for brevity) into our output object, set this object into the *data* field of the *CommandResult* object and then set the *succeeded* flag to true before exiting the method. And we're done!

Now if we invoke our service using something like *curl* or a REST testing tool (or even the browser in this case since this service is invoked via GET) with the URL *http://localhost:8080/delegate/xsfapp/hello/world/Bloggs/Joe* we will receive the JSON shown in the listing below as our output.

```
{
    "succeeded":true,
    "data" : {"first_name":"joe", "last_name":"bloggs"},
    "message": ""
}
```

Hello World, Part 2

A slightly more enhanced version of our service is provided in *com.xtivia.xsf.samples.HelloWorldCommand2* shown in the listing below (and included in the sample XSF distribution). This command example is very similar to our first one both in terms of route definition and implementation, but demonstrates a couple of additional features of XSF in code we add near the end of its *execute()* method.

Code has been added to demonstrate retrieving query parameters from the request; note that the technique is the same as before in terms of interrogating the supplied context for the desired parameter. So our code does not need to worry with boilerplate logic to retrieve path parameters vs. request query parameters; instead this is all handled by XSF and made available to services/commands via the supplied context. This also has the additional benefit of making commands easy to test; the unit tests can easily mock values for testing and place them into an input context – the context object supports all of the 'write' methods for a Map as well.

In our case the code tests for the presence of a query parameter named *mname* and if it found echoes the value back in the return object. If the query parameter was not supplied on the request a default value is returned instead.

```
package xsf.samples;
[imports omitted for brevity]

@Route(uri="/hello/world2/{last}/{first}", method="GET")
public class HelloWorldCommand2 implements ICommand {
    @Override
    public CommandResult execute(IContext context) {
        Map<String,String> data = new HashMap<String,String>();

        //inputs from path parameters
        String firstName = context.find("first");
        Validate.notNull(firstName,"Required path param=firstName not found");
        String lastName = context.find("last");
        Validate.notNull(lastName,"Required path param=lastName not found");
        data.put("first_name", firstName);
        data.put("last_name", lastName);

        // optional input from query string
        data.put("middle_name", "NMN");
        String middleName = context.find("mname");
        if (middleName != null) {
```

```
            data.put("middle_name", middleName);
        }

        //input based on logged-in Liferay user
        User user = context.find(ICommandKeys.LIFERAY_USER);
        data.put("user_email", "Not authenticated");
        if (user != null) {
            data.put("user_email", user.getEmailAddress());
        }

        return new CommandResult().setSucceeded(true).setData(data).setMessage("");
    }
}
```

The final portion of our *execute()* method provides an early taste of integration with Liferay elements in XSF-based services. In this case the service leverages the fact that XSF will determine if the current user is logged into Liferay or not and if so will place a copy of the Liferay *User* object that represents the logged-in user into the context for subsequent access by services/commands.

The listing below provides two examples of URLs to invoke our service and the JSON that results from these respective invocations.

```
URL: http://localhost:8080/delegate/xsfapp/hello/world2/Bloggs/Joe?mname=Lee" (not logged in)
JSON returned:

{
  "succeeded":true,
  "data": {
    "first_name":"Joe",
    "middle_name":"Lee",
    "last_name":"Bloggs",
    "user_email":"Not authenticated"
  },
  "message":""
}

URL: http://localhost:8080/delegate/xsfapp/hello/world2/Bloggs/Joe (after log in)
JSON returned:

{
  "succeeded":true,
  "data":{
    "first_name":"Joe",
    "middle_name":"NMN",
    "last_name":"Bloggs",
    "user_email":"xsf@xtivia.com"
    },
```

```
    "message":""
}
```

Hello World, Part 3

Until now our examples have been based on simple GET requests where all inputs are supplied in the URI, and where the returned object is a standard Java Map class. In our final example we will demonstrate a POST based service and will use custom application objects to define the input as well as the output for the service. Our third example is provided in *com.xtivia.xsf.samples.HelloWorldCommand3,* shown below and included in the XSF distribution. Note that in the interest of space we have not included the source listings for the *SampleInput* and *SampleOutput* classes but these are also available in the XSF distribution.

```
package com.xtivia.xsf.samples;
[imports omitted for brevity]

@Route(uri="/hello/world3/{id}", method="POST",
           inputKey="inputData", inputClass="com.xtivia.xsf.samples.model.SampleInput")
public class HelloWorldCommand3 implements ICommand {

    @Override
    public CommandResult execute(IContext context) {

        SampleOutput output = new SampleOutput();

        //inputs from path parameters
        String id = context.find("id");

        //inputs from posted JSON (marshalled to Java object)
        SampleInput input = context.find("inputData");
        if (input == null) {
            return new CommandResult().setSucceeded(false).setMessage("Input missing");
        }

        output.setId(id);
        output.setCount(input.getInputNumber()+1);
        output.setText(input.getInputText().toUpperCase());

        Calendar calendar = Calendar.getInstance();
        calendar.setTime(input.getInputDate());
        output.setDayOfWeek(calendar.get(Calendar.DAY_OF_WEEK));
        output.setMonth(calendar.get(Calendar.MONTH));

        return new CommandResult().setSucceeded(true).setData(output).setMessage("");
    }
}
```

Note that in this example we add two additional attributes related to the marshalling of an inbound object in the @Route annotation, *inputKey* and *inputClass*. The former is used to instruct XSF what key to use when storing the inbound object in the context, and the latter defines what Java class should be used to marshal the inbound JSON string. XSF currently uses the Jackson library for all of its JSON marshalling (XSF in fact supports a pluggable architecture for marshalling; out-of-the-box it supports JSON but provides an extension mechanism for a implementing custom marshaller if, for example, you wanted to use XML instead of JSON.)

The Jackson library is quite flexible in terms of its marshalling support as we have seen in the previous examples where we returned a Map object and it was converted to JSON with no additional effort on our part. Embedded child objects and embedded collections are supported quite nicely by Jackson, however, we would recommend as a best practice that you attempt to use the "value object" pattern for objects used to marshal JSON to/from the client.

In our sample service/command we obtain the input object from the client, do some minor manipulations on fields from that object, and then set the modified values back into an object to be returned to the client. Note than in the case of the returned object we do not need to specify any additional metadata about the returned class. All reasonably straightforward value objects can be marshalled into JSON by interrogation of the class definition for the returned object.

The listing below provides an example of a URL invocation that might be used to trigger the execution of this service, as well as a sample JSON input and the corresponding JSON output:

```
URL: http://localhost:8080/delegate/xsfapp/hello/world3/2742

    Input JSON:
        {
          "inputText" : "foobar",
          "inputNumber" : 22,
          "inputDate" : "2015-01-06T20:23:38"
    }

    Output JSON:
    {
          "succeeded":true,
          "data": {"id":"2742", "text":"FOOBAR", "month":0, "dayOfWeek":3, "count":23},
          "message":""
    }
```

Implementing CRUD (Resource) Services

On many occasions the technique we have described thus far is sufficient for implementing any services you will need. On occasion, however, you may wish to implement a full CRUD (Create, Read, Update and Delete) service that aligns with the Fielding's notion of a "resource".

As an example let's say that we wish to implement a full REST endpoint at the URI */delegate/xsfapp/people*. In this scenario we also want to implement a full CRUD service, so that a GET to */delegate/service/people* would retrieve a list of all people, another GET request to */delegate/service/people/1* would retrieve the person with an ID of 1, a DELETE request to */delegate/service/people/5* would delete the person with an ID of 5, and so on according to the traditional REST metaphor. Further, it would be nice if we could implement all of these endpoints/routes in a single Java class file to facilitate simplicity of coding, code reuse, etc.

XSF facilitates this coding model with a simple extension to the programming techniques we have been using so far. The technique is illustrated in the sample listing excerpt below.

```
[imports omitted for brevity]
@Route(uri="/methods")
public class MethodRoutesCommand implements ICommand {

	@Route(uri="/get",authenticated=false)
	public CommandResult doGet(IContext ctx) {
		Map<String,String> data = new HashMap<String,String>();
		data.put("Hello", "World");
		CommandResult cr = new CommandResult();
		cr.setSucceeded(true);
		cr.setData(data);
		cr.setMessage("GET");
		return cr;
	}

	@Route(uri="/post/{last}/{first}", method="POST",
		   authenticated=true,
		   inputKey="inputData", inputClass="com.xtivia.xsf.core.web.TestResource")
	public CommandResult doPost(IContext ctx) {
		CommandResult cr = new CommandResult();
		cr.setSucceeded(true);
		TestResource tr_in = ctx.find("inputData");
		TestResource tr_out = new TestResource();
		tr_out.setText(this.getClass().getName());
		tr_out.setRate(tr_in.getRate().add(new BigDecimal(1)));
		cr.setData(tr_out);
		cr.setMessage("POST_METHOD"+tr_in.getText());
		return cr;
	}
```

```
	@Override
	public CommandResult execute(IContext context) {
		return Xsf.dispatch(this, context);
	}
}
```

As we can see in the example listing above, multiple REST endpoints can be defined in a single Java class file by (1) attaching @Route annotations to the individual methods, (2) ensuring that the annotated methods implement the same signature as *execute*(), and (3) changing the code of your *execute()* method to invoke *Xsf.dispatch*() as in the code above.

Note that we have attached @Route annotations at both the class and method levels. This allows us to define the base portion of a URI at the class level, and then add additional method-specific routing information. So, for example, the *doGet()* method above will be invoked at the URI */delegate/xsfapp/methods/get*. Similarly the *doPost()* method could be invoked at */delegate/xsfapp/methods/post/Bloggs/Joe*.

Finally note that we can create a mixed authentication model using the combined @Route approach. In the example above the @Route annotation on the doGet() indicates that it is publicly available, but then the method for *doPost()* changes this to indicate that the endpoint used to create new records requires an authenticated user. Alternatively, an @Route can be used at the class level to set a default in terms of authentication, and then it can be overridden at the individual method level for cases that don't use the default. (see the following section on Authentication and Authorization for more information.)

XSF Authentication and Authorization Mechanisms

XSF is fully integrated with the Liferay authentication/authorization system. What this means to you as a services developer is that you can access information about the currently logged-in Liferay user from within your REST services, and you can also leverage mechanisms that allow you to create access control rules for services.

In its simplest form an XSF command is marked as requiring authentication via the *authenticated* attribute on the @Route annotation attached to it. The default value for this attribute is TRUE which means that by default each command that you create will require that the user be logged in to Liferay in order to access it. Commands that are intended to be used as public/guest routes should have this value set to FALSE (as can be seen in the "HelloWorld" commands provided in *com.xtivia.xsf.samples*.

Custom Authorization

Commands can indicate to XSF that they require some type of customized authorization by implementing the *IAuthorized* interface. This interface requires the implementation of a single method, *authorize()*, that must return a true/false value to indicate whether or not authorization was successful.

So essentially if a command implements the *IAuthorized* interface it is indicating to the XSF framework "yes, I require some customized authorization before I can be accessed", and the command is then responsible for implementing the required authorization. We can see an example of a custom authorization implementation in the listing below. The *authorize()* method invokes a static method in a helper class (provided in the distribution as a sample) that determines whether or not the logged in user is a Portal administrator.

```
package xsf.samples.auth;

[imports omitted for brevity]

@Route(uri="/needsomniadmin/echo/{last}/{first}", method="GET", authenticated=true)
public class OmniadminCommand implements ICommand, IAuthorized {

    @Override
    public CommandResult execute(IContext context) {

        Map<String, String> data = new HashMap<String, String>();

        //inputs from path paramters
        String firstName = context.find("first");
        String lastName = context.find("last");
        data.put("first_name", firstName);
        data.put("last_name", lastName);

        return new CommandResult(data);
    }
```

```
    @Override
    public boolean authorize(IContext context) {
       return XsfAuthUtil.isOmniAdmin(context);
    }
}   -
```

Use of the Context in Authorization Logic

When executing the *authorize()* method the supplied context object provides two important elements that custom authorizers can use to implement whatever unique access control requirements that they need. One of these has been discussed before and that is the User object representing the user that has logged in to Liferay (stored under the key *ILiferayCommandKeys.LIFERAY_USER).*

The other key object that can be accessed using the key value *ILiferayCommandKeys.LIFERAY_PERMISSION_CHECKER* is a Liferay permission checker object that can be used to determine whether or not the logged in user meets custom criteria in terms of assigned Liferay roles and permissions. The primary benefit provided by this facility is that you can now use Liferay roles and permissions to control access to your REST services in addition to their traditional application in managing access to elements of the user interface (i.e. pages and portlets).

For further details on the use of the supplied permission checker please refer to the sample *OrgAuthorizedCommand* class in the package *xsf.samples.auth*.

Authorizer Examples

We have provided a static class named *XsfAuthUtil.java* as an example helper class that provides a number of utility methods that could be used by custom authorizers to execute Liferay-specific permission checks. This file is located in the package *xsf.samples.auth.* In this same package you will find examples that implement authorization logic based on whether or not the user is in a portal admin role, is in a specific portal (regular) role, and even custom authorization based on (as an example) whether or not any of the user's assigned organization roles would grant them access to view blogs data. Please refer to these examples for assistance in developing your own custom authorization logic.

Unit Testing

As an example of how straightforward XSF makes it to write unit tests for the commands that provide your service endpoints the framework samples include a unit test case for the *HelloWorldCommand2* command (*HelloWorldCommand2Test.java* under */src/test*). An excerpt from this test code is show below:

```
@RunWith(MockitoJUnitRunner.class)
public class HelloWorldCommand2Test {

  // build a context to emulate what XSF would create
  CommandContext context = new CommandContext();

  // create an instance of the command to test
  HelloWorldCommand2 command = new HelloWorldCommand2();

  @Mock
  User mockLiferayUser;

  [other tests excluded for brevity]

  public void testSuccessWithNoQueryParamAndUser() {
    // simulate URL path parameters
    context.put("first", "Joe");
    context.put("last", "Bloggs");

    // mock out a logged-in user
    Mockito.when(mockLiferayUser.getEmailAddress()).thenReturn("xsf@xtivia.com");
    context.put(ILiferayCommandKeys.LIFERAY_USER, mockLiferayUser);

    CommandResult cr = command.execute(context);
    assertEquals(cr.isSucceeded(),true);
    Map results = (Map) cr.getData();
    assertNotNull(results);
    String firstName = (String) results.get("first_name");
    String lastName = (String) results.get("last_name");
    String middleName = (String) results.get("middle_name");
    String userEmail = (String) results.get("user_email");
    assertEquals(firstName,"Joe");
    assertEquals(lastName,"Bloggs");
    assertEquals(middleName,"Not Available");
    assertEquals(userEmail, "xsf@xtivia.com");
  }
}
```

A quick glance at the source code and comments for this test file demonstrates that the use of the *IContext's* map-based abstraction for reading (and writing) environment parameters makes it simple and

easy to create mocked environments that emulate both success and failure conditions with a minimum of effort on the part of the test writer. Note that this example also shows how we can easily use a framework like Mockito to create a “fake” Liferay User object for testing.

Customization

There are very few required changes needed to develop your own custom services instead of using the supplied Xtivia samples.

Changing the delegate/xxx portion of the URLs

By default all service invocations use *[host:port]/delegate/xsfapp/....* style URLs. While the *'delegate'* portion of the URLs can only be changed by changing your Liferay configuration, you can easily map the *'xsfapp'* portion of this URL to something of your own choice by modifying the *subcontext* init-parameter in *src/main/webapp/WEB-INF/web.xml*.

Using Spring in your application

As discussed earlier XSF is built on top of Spring and uses its own private application context XML file that is embedded inside the XSF JAR file. XSF does attach and include another application context file to this main (“parent”) file for your use in describing your own application-level Spring components.

This file is named *xsf-app-context.xml* and is (must be) located under the *META-INF* directory *in src/main/resources*. Note that XSF expects this file to be present even it is empty so please do not delete or remove it.

You may use this file to add whatever application artifacts you wish to make available via Spring and they will then be available for use by your XSF services/commands.

Annotated Service Classes

By default the application POM file provided by XSF (either in the downloaded sample from GitHub or the archetype-generated application) includes plugin support for building a small XML file that XSF includes at runtime to load your annotated (i.e. @Route) command classes. When using this feature is optional, by using it as-is no work is required on your part as you add new packages and classes for services -- XSF will automatically discover and use them.

If, however, your build process or requirements drive the need to remove this feature from the POM then you will need to ensure that Spring discovers and loads your annotated classes. The simplest way to do this is to add a <*context:component-scan base-package*> element to your application Spring file (see above) that points to the package(s) that contain your annotated XSF commands.

Invoking XSF Services Using BASIC Authentication

By default invoking XSF services that require authentication from outside the browser is not enabled. In other words, as shipped the XSF framework will automatically detect and manage Liferay login/logout sequence from the browser but will not recognize BASIC Authentication headers when invoked from a standalone progam.

This can be enabled, however, by simply making a small additon to your application's Spring context file (*xsf-app-context.xml*, see prior section). To enable the use of BASIC Authentication simply add the following entry to the file:

```
<bean class="com.xtivia.xsf.liferay.LiferayBasicAuthDecorator"/>
```

This will enable the discovery of a bean that is shipped with XSF and will look for and process BASIC Authentication headers. Note that BASIC Authentication will be ignored if a Lifeay login session is present (i.e. Liferay logins take precedence).

By default, the credentials presented in the BASIC Authentication headers are presumed to be an email associated with a Liferay user. If instead you wish to treat these credentials as Liferay screen names use the following form in your Spring context file:

```
<bean class="com.xtivia.xsf.liferay.LiferayBasicAuthDecorator">
  <property name="useScreenName" value="true"/>
</bean>
```

Finally this feature in XSF leverages and uses the authentication pipeline in Liferay, so you could use the properties associated with that pipeline to constrain access to only certain host IPs. You will need to ensure that the URL route associated with your application (e.g. */delegate/xsfapp*) is permitted for access in the authentication pipeline.

Summary and Additional Features

Admittedly our provided examples are simple, but hopefully we have given you insight into how easy it can be to set up a suite of application REST services using XSF.

XSF services provide a rich mechanism for the development of REST services in Liferay including, among other things, use of the Liferay permissions model for services. XSF also supports a number of other valuable features (not documented here) including:

- integration with Liferay logging
- the use of command "chains"
- plugging in your own custom marshaller (e.g. to use XML instead of JSON)
- dynamic routing for cases where the mapping of URIs to commands is not known until runtime (i.e., dynamic dispatching).

Future updates may also add a portlet to the distribution for use in management and visualization of XSF service throughput and error handling. If you have questions or interest regarding the use of any of these advanced features in XSF please email us at *xsf@xtivia.com*.

Licensing

XSF is licensed via LGPL so that you can use it freely in developing your own Liferay application services.

For those interested in more information regarding XSF please email us at **xsf@xtivia.com**.

Appendix A, XSF Context Built-Ins

The following table shows the “built-ins” for an XSF context, i.e., variables that are included by default in every XSF context object:

Key	Description
ICommandKeys.HTTP_SESSION	Returns the *HttpSession* associated with the current request.
ICommandKeys.HTTP_REQUEST	Returns the *HttpServletRequest* associated with the current request.
ICommandKeys.HTTP_RESPONSE	Returns the *HttpServletResponse* associated with the current request.
ICommandKeys.SERVLET_CONTEXT	Returns the *ServletContext* associated with the current request.
ICommandKeys.PATH_PARAMETERS	Returns a map of the path parameters associated with the curent request. For example, if the Route associated with the current request had a URI like */hello/world/{last}/{first}*, this map would include keys for ‘last’ and ‘first’ with the values that had been supplied for each.
ICommandKeys.ROUTING_INFO	Returns a copy of an XSF object that contains information about the route that was matched to this request. This is mostly useful to internal XSF objects and subsystems.
ILiferayCommandKeys. LIFERAY_USER	Returns the Liferay User object associated with the current request (or null if no user is currently logged in.)
ILiferayCommandKeys. LIFERAY_COMPANY_ID	Returns the Liferay Company ID associated with the current request (or null if no user is currently logged in.)
ILiferayCommandKeys. LIFERAY_PERMISSION_CHECKER	Returns a Liferay permission checker object for the current user (or null if no user is currently logged in.) Note that this object is (1) created lazily, i.e. is not actually created until the first request for it occurs within a single request and (2) is subsequently cached for a given request, so that any subsequent retrievals within the scope of the same request will retrieve the cached copy.