Building a Basic iRODS Web Application using Jargon

IRODS User Group Meeting 2012

Mike Conway – DICE

www.irods.org

Introduction

iRODS [1] provides a unique and powerful platform for developing policy-managed, data-intensive applications. The flexibility of this policy-driven approach to data enables iRODS to serve as the foundation for any number of applications in data-driven science, digital preservation, digital humanities, enterprise data management, and other areas. A common requirement across these areas is to create custom user interfaces and portals that extend iRODS data to a target audience. Using the iRODS grid and rule engine in combination with available tools, frameworks, and client API libraries, it is possible to develop a wide range of client interfaces.

Jargon [2] is a blanket term for the Java client API libraries for iRODS, as well as associated GUI and client interfaces built on top of the Jargon libraries. The client development efforts had the original stated goal as follows:

Jargon, and the integration possibilities that it will enable, has the goal of making the IRODS Data Grid as familiar to developers as a database or messaging middleware platform, and a dependable tool to help manage the expanding need for secure sharing and preservation of data. [3]

The overarching goal for Jargon has been to adopt current best practices, and to make development of interfaces on top of the iRODS platform as familiar as possible. It is up the to the reader to decide if that goal is being met!

This paper will use the now-standard Spring MVC framework, along with standard Java Servlets to build a sample iRODS web interface. This exercise will touch on obtaining the code, setting up a basic project, and configuring common components used in iRODS web interfaces.

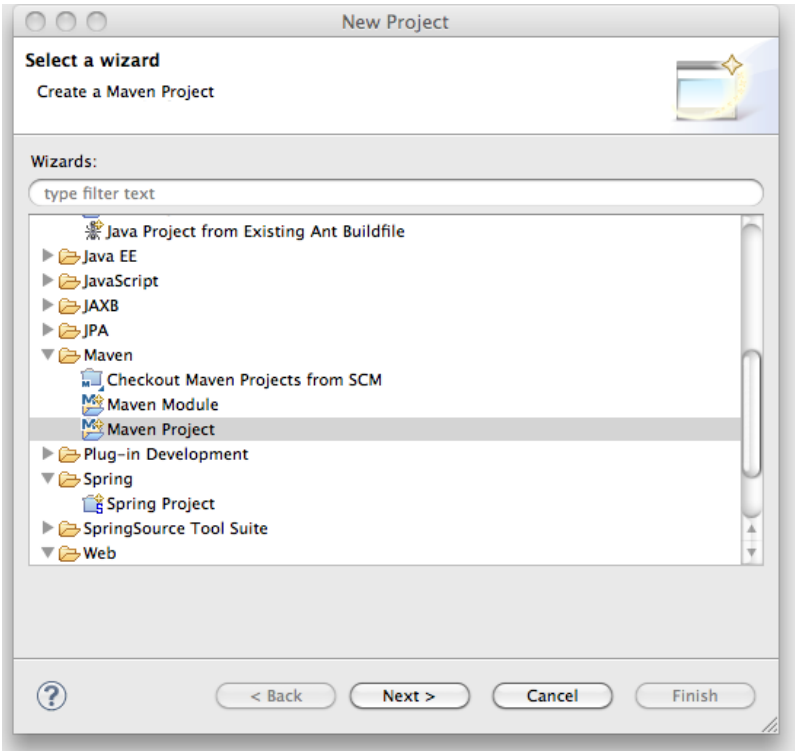
Setting up a Basic Project

DICE has adopted Maven [4] as the primary Java build and distribution tool. Maven can manage the process of obtaining dependencies (such as the Jargon-core libraries), building and packaging, as well as testing and generating necessary project artifacts. Once your project is configured with Maven, you may add the RENCI Maven repository and the required Jargon dependencies will be automatically provisioned in your project.

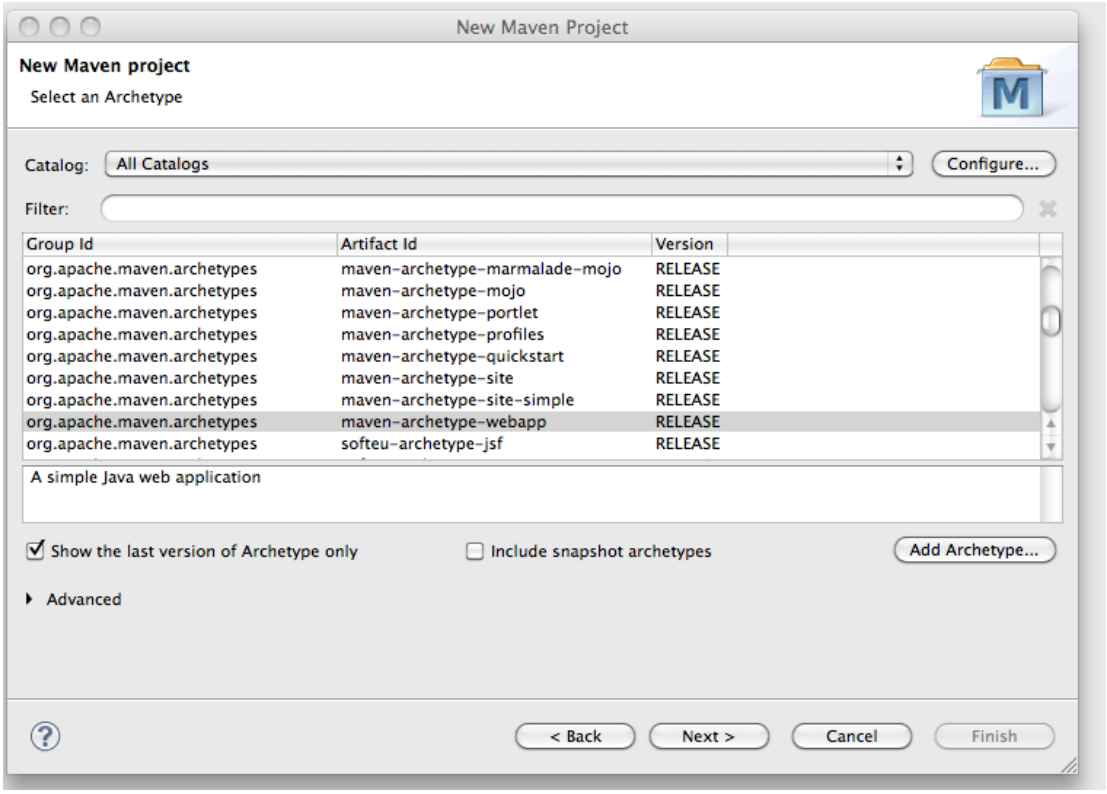
Step 1 – Create a skeleton Maven project

Prerequisites: Maven needs to be installed on your computer. Eclipse needs to be installed on your computer.

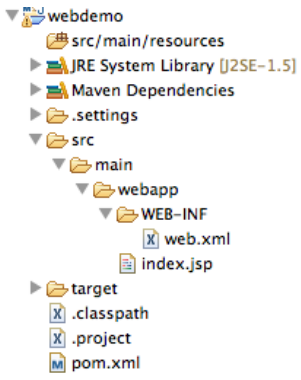
We will be using Eclipse to develop our project. The first step will be to create a basic Maven Simple Webapp project. In Eclipse, select File/New Project to bring up the New Project Dialog. Select Maven Project



Advance through the wizard to select a Maven archetype (a preset that defines the skeleton to be created)



Fill in the group id and artifact id. For this exercise, the group org.irods.jargon and and artifact id of webdemo were used. This should create the project in Maven as shown below. Note that the Maven archetype creates a skeleton web application, as well as a project build file, or POM, called pom.xml.



Our Maven POM will be used to define other libraries that our web interface will depend on. One framework that this demo uses is Spring [5]. Spring is a 'dependency injection' [6] framework that allows applications to be easily developed by wiring components together. Spring enhances the simplicity and testability of otherwise complex applications.

You should be able to now select your project and run it, displaying an initial 'Hello World' from the index.jsp page.

Now we need to add dependencies on Spring to our project [7] by adding some repositories for Spring

and for Jargon artifacts:

```
<repositories>
		<repository>
			<id>ibiblio.repository</id>
			<name>ibiblio.repository</name>
			<url>http://mirrors.ibiblio.org/pub/mirrors/maven2</url>
			<releases>
				<enabled>true</enabled>
			</releases>
			<snapshots>
				<enabled>false</enabled>
				<updatePolicy>never</updatePolicy>
				<checksumPolicy>fail</checksumPolicy>
			</snapshots>
		</repository>
		<repository>
			<id>renci.repository</id>
			<name>renci.repository</name>
			<url>http://ci-dev.renci.org/nexus/content/repositories/public</url>
			<releases>
				<enabled>true</enabled>
				<updatePolicy>always</updatePolicy>
				<checksumPolicy>warn</checksumPolicy>
			</releases>
			<snapshots>
				<enabled>false</enabled>
				<updatePolicy>never</updatePolicy>
				<checksumPolicy>fail</checksumPolicy>
			</snapshots>
		</repository>
		<repository>
			<id>jboss</id>
			<name>jboss.repository</name>
			<url>https://maven.atlassian.com/content/repositories/</url>
			<releases>
				<enabled>true</enabled>
				<updatePolicy>always</updatePolicy>
				<checksumPolicy>warn</checksumPolicy>
			</releases>
			<snapshots>
				<enabled>false</enabled>
				<updatePolicy>never</updatePolicy>
				<checksumPolicy>fail</checksumPolicy>
			</snapshots>
		</repository>

	</repositories>
```

And then Spring and Jargon dependencies:

```
<dependencies>
		<dependency>
			<groupId>org.irods.jargon</groupId>
			<artifactId>jargon-core</artifactId>
			<version>${jargon.version}</version>
		</dependency>
		<dependency>
			<groupId>org.springframework</groupId>
```

```
            <artifactId>spring-aop</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${spring.version}</version>
        </dependency>
        <dependency>
            <groupId>javassist</groupId>
            <artifactId>javassist</artifactId>
            <version>3.8.0.GA</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>${spring.web.version}</version>
            <type>pom</type>
        </dependency>
    </dependencies>
```

Your project should now be able to find and use the Jargon libraries.

Step 3 – Configure the Web Application to use Spring MVC

Within DICE, the ability to develop iRODS applications using the Spring Framework and dependency injection has been a guiding principle in designing Jargon. Using Spring, we will configure the primary iRODS components in a typical web application. Spring is described this way:

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments. [5]

Spring MVC is a Model View Controller framework for web application development. Spring MVC is a good choice for iRODS web interfaces, though any framework that can work with Spring, such as

Struts, will work. We need to add some special XML configuration to our project so that Spring can be started, and the objects we need can be wired together. We can start by setting up the basic parts of a Spring MVC application, based on the SpringSource step-by-step tutorial [7]. This is not a Spring tutorial, so refer to this tutorial on the web:

http://static.springsource.org/docs/Spring-MVC-step-by-step/part1.html

For the purposes of the tutorial, we will pick up with a Spring MVC web application configured as in the tutorial at this location.

A Diversion – iRODS as an Application Platform

It is important to think of iRODS as a platform, and to understand the capabilities of iRODS. The iRODS platform has a wide range of server-side capabilities, including the ability to manage policy and to operate on data using policy hooks, deferred and periodic rule execution, and user submitted rules. IRODS can also manage data and user supplied metadata across federated grids, while maintaining a rich catalog of system metadata. One could argue that the overlap between data intensive applications, digital preservation applications, and policy-driven data management on a distributed, multi-organizational scale is new territory. This means that the best practices and design patterns for application development on such a platform are being defined in part by readers of this document! As applications of some complexity are developed on iRODS, it is important to look at the capabilities of iRODS as well as the client application, and to properly place functionality at the appropriate tier. This is even more important when iRODS is one tier of a multi-tier application.

Jargon

Bearing this in mind, it is useful to think of the actual data in iRODS, the metadata catalog, and the rule engine as components, with a global logical namespace, and a transparent, distributed backbone. The Jargon API allows developers to access the capabilities of each of these components to create data intensive, policy managed applications. Using Jargon libraries in a web application is (purposely) not very different from standard development practices for other back-end frameworks, this is intentional. iRODS-specific functionality is represented by a framework with a few familiar design patterns, a POJO domain model, and a Java file system abstraction.

Jargon communicates with iRODS by establishing a socket connection, and invoking iRODS services using the iRODS XML protocol [8].The Jargon core library functions primarily as an implementation of the iRODS protocol and packing instructions [9], as well as an implementation of the connection between iRODS and the client over which the iRODS packing instructions and related data are exchanged.

Jargon hides the complexity of connection handling and protocol management from the end-user through a set of abstractions. Jargon is designed so that API users do not see network connections or packing instructions. Rather, all interfaces are through Access Objects, Java I/O library abstractions, and POJO (Plain old java object) value objects. iRODS has a wide range of capabilities, so currently not all iRODS functions are exposed in Jargon, though over time the feature coverage continues to grow. The Jargon-core framework is built with this continued expansion in mind.

Step 3 – Configure Base Jargon Components for Your Application

Jargon connects to iRODS via a TCP/IP socket, typically to port 1247. This main communications channel is used to send and receive iRODS packing instructions per the iRODS client protocol. Your web application needs to set up an manage a small set of core components. These components will manage connections to iRODS, allow you to find out about the iRODS and Jargon environment, and to create the API components to interact with iRODS (the data, the metadata catalog, and the rule engine), and to represent iRODS data as Java I/O files and streams.

What we need to do is set up these base components within our application context. The ServletContext context of a Servlet application is defined like so:

Complementing the request scope, a ServletContext instance allows for server-side objects to be placed in an application-wide scope.. This type of scope is ideal for placing resources that need to be used by many different parts of a Web Application during any given time. [10].

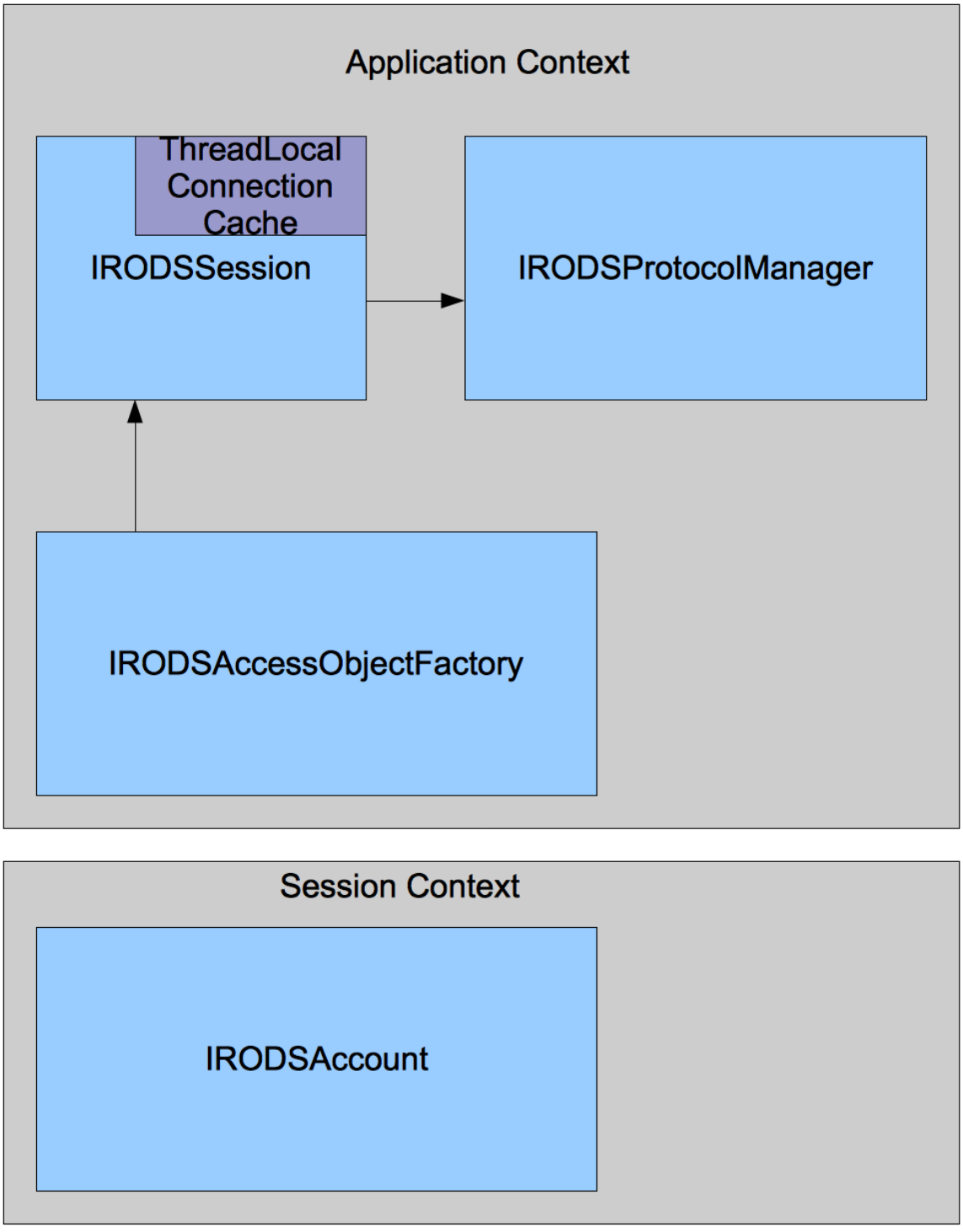
The Spring framework will (under the covers) use the ServletContext to store the shared infrastructure for iRODS-enabled Jargon applications.

In order to connect to iRODS, the following components are used, and are represented in the org.irods.jargon.core.connection.* package:

- **IRODSSession** - This is a central object that manages the connection to iRODS. A client will, under the covers, ask IRODSSession to return a an IRODSCommands object. Jargon maintains a cache of connections in a ThreadLocal. This means that multi-threaded applications typically never share a connection between threads. The connection handling that IRODSSession does should not typically be a concern of API users, except in special circumstances. The IRODSSession is an expensive object that also holds other information, such as the current properties that control Jargon behavior, a cache of extensible metadata mappings, and other objects that should only be created once in an application.
- **IRODSProtocolManager** - This is an interface that defines an object that can give a connected IRODSCommands object to the IRODSSession upon request. The IRODSSession then returns the IRODSCommands object when done, and it is the responsibility of the IRODSProtocolManager to dispose of or passivate a connection when done. The interface can be implemented to just return a new connection every time (the default), or it can be extended to create proxy connections, a connection pool or cache, or other special behaviors.
- **IRODSAccount** - This class defines a principal who is attempting to connect to a given iRODS host and zone. This object will be created at login, and used to obtain connections to different iRODS servers during the processing of servlet requests.

The **IRODSSession** and **IRODSProtocolManager** are defined such that they may be wired together using Spring or another inversion of control container. There should only be one instance of each in an application, created at startup, and torn down at application shut-down. These objects should not be created for every request in a servlet application.

The following diagram shows the simple configuration that will be wired up using the Spring framework:


Application Context
ThreadLocal
Connection
Cache
IRODSSession
IRODSProtocolManager
IRODSAccessObjectFactory
Session Context
IRODSAccount

As you can see from the above figure, the IRODSAccessObjectFactory, wired in to the ServletContext by Spring, will be available to each Servlet. This base factory object can return connected instances of 'Access Objects' as well as java.io file and stream objects. The IRODSAccessObjectFactory is wired in with references to an IRODSSession object. The flow for connection handling is as follows:

1. An access object is requested from an IRODSAccessObjectFactory by passing in an IRODSAccount that will represent the target iRODS host and user information.
2. The IRODSSession object will look into a ThreadLocal pool, and see if a connection is established. If a connection is available for the current Thread, this connection will be returned.
3. If a connection is not available, the IRODSSession will request a connection from the IRODSProtocolManager. The IRODSProtocolManager is an interface, and this interface can be implemented to create a new connection per request, or it could operate as a pool, cache, or a source of proxy connections.
4. Once the close() method is called in the Access Object, or from the IRODSAccessObjectFactory, the connection is returned by the IRODSSession and removed from the cache. The connection is returned to the IRODSProtocolManager where the connection is disconnected, or the connection may be passivated in a pool or cache.

From the perspective of the developer, the important elements are:

1. Wire in the IRODSAccessObjectFactory so that it is available to all servlets.
2. Use the IRODSAccessObjectFactory from an application context.
3. Store the IRODSAccount after log-in in a session variable and use the this IRODSAccount to create new objects from the IRODSAccessObjectFactory.
4. Use a filter or interceptor to manage references to the IRODAccessObjectFactory and to obtain the IRODSAccount from the session during request processing, and to call close() on the IRODSAccessObjectFactory after request processing.

Using these basic concepts, it becomes relatively easy to process servlet requests by creating appropriate Jargon access objects and java.io files and streams using Spring. Note that such 'wiring' could also be accomplished manually at application start-up, or through another dependency injection framework. The following is a snippet of Spring configuration that wires the objects as in the diagram above.

```
<bean id="irodsConnectionManager"
                class="org.irods.jargon.core.connection.IRODSSimpleProtocolManager"
                factory-method="instance" init-method="initialize" destroy-method="destroy">
        </bean>

        <bean id="irodsSession"
                class="org.irods.jargon.core.connection.IRODSSession" factory-method="instance">
                <constructor-arg
                        type="org.irods.jargon.core.connection.IRODSProtocolManager"
ref="irodsConnectionManager" />
        </bean>

        <bean id="irodsAccessObjectFactory"
                class="org.irods.jargon.core.pub.IRODSAccessObjectFactoryImpl">
                <constructor-arg ref="irodsSession"></constructor-arg>
        </bean>
```

As you can see, the IRODSProtocolManager implementation (for a new connection per request) is created. An IRODSSession object is then created that has a reference to the IRODSProtocolManager

that will distribute connections. Finally, an IRODSAccessObjectFactory is created, and a reference to IRODSSession is wired in. As each servlet is invoked during request processing, this IRODSAccessObjectFactory will be used to obtain services connected to iRODS. For the purposes of our demo, we will add this Spring configuration snippet to the springapp-servlet.xml file created in step 3. This can be found in the src/main/webapp/WEB-INF directory.

Note the following messages in the log at application start-up:

```
2233 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Creating shared instance of singleton bean 'irodsConnectionManager'
2234 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Creating instance of bean 'irodsConnectionManager'
2598 [pool-2-thread-1] INFO  org.irods.jargon.core.connection.IRODSSimpleProtocolManager  - creating
simple protocol manager
2600 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Eagerly caching bean 'irodsConnectionManager' to allow for resolving potential circular references
2601 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Invoking init method  'initialize' on bean with name 'irodsConnectionManager'
2627 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Finished creating instance of bean 'irodsConnectionManager'
2629 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Creating shared instance of singleton bean 'irodsSession'
2630 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Creating instance of bean 'irodsSession'
2893 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Returning cached instance of singleton bean 'irodsConnectionManager'
2930 [pool-2-thread-1] INFO  org.irods.jargon.core.connection.IRODSSession  - IRODS Session creation,
loading default properties, these may be overridden...
2947 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
2951 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Creating shared instance of singleton bean 'irodsAccessObjectFactory'
2951 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Creating instance of bean 'irodsAccessObjectFactory'
2951 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Returning cached instance of singleton bean 'irodsSession'
3215 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Eagerly caching bean 'irodsAccessObjectFactory' to allow for resolving potential circular references
3216 [pool-2-thread-1] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory  -
Finished creating instance of bean 'irodsAccessObjectFactory'
```

Another Diversion – Access Objects in Jargon

The ICAT (the iRODS system catalog) is a database that contains system maintained metadata. This catalog has abstractions for things like zones, resources, data objects, collections, and users. Jargon maps these ICAT domains to POJO (plain old Java) value objects. These domain objects can be found in the org.irods.jargon.core.pub.domain.* package. In order to access and modify ICAT metadata for such domains, there are Access Objects.

Access Objects are the primary public interface for iRODS services. The Access Objects are found in the org.irods.jargon.core.pub package, and are the primary place to look for iRODS functions. Access objects are based loosely on the DAO pattern (and the connection handling via a ThreadLocal is based on the general pattern of Hibernate DAO objects). Access Objects use the connection sharing mechanism provided by the IRODSSession objects. Access objects in the same thread, for each connection described in an IRODSAccount, will automatically share the same connection. The Access

Objects are meant to be compos-able, so that an access object for services involving an iRODS Resource may use an access object for running an iRODS rule, and executing an iRODS General Query. It is important to note that iRODS does not have a concept of transactions, so do not expect access objects to manage rollback for an involved operation. This sort of effect can be accomplished using iRODS rules, with recovery steps, but is not part of the iRODS protocol. As Jargon evolves, more client operations will migrate to iRODS rules invoked on the server. It is important to be aware of this limitation.

In addition to services that access and modify the ICAT domain model, Access Objects are available for most iRODS services. This includes services to run rules, to execute server-side procedures, to work with general queries, and the like.

To illustrate, we can look at the services that operate on an iRODS user. The domain package has a User object to represent the entity in the ICAT, and there is a corresponding UserAO access object to operate on the iRODS user. This is illustrated by this snippet from the UserAO object:

```
/**
         * Add the given user to iRODS
         *
         * @param user
         *          {@link org.irods.jargon.core.pub.domain.User} with information
         *          on the user to be added.
         * @throws JargonException
         * @throws DuplicateDataException
         *           thrown if the user already exists.
         */
        void addUser(User user) throws JargonException, DuplicateDataException;

        /**
         * List all users.
         *
         * @return <code>List</code> of
         *       {@link org.irods.jargon.core.pub.domain.User}
         * @throws JargonException
         */
        List<User> findAll() throws JargonException;
```

As you can see, the methods take plain Java objects as parameters, and return domain objects as results. Using access objects in this way, iRODS functionality is easy to access, and no low-level details are exposed. We will use this UserAO access object to generate a list of users in a sample web page.

Step 4 – Using Jargon to Process a Request

For the purpose of this demonstration, we will create a controller for Spring MVC that will present a list of iRODS users. This controller will be wired together with the IRODSAccessObjectFactory we created in step 3. We will be using the Spring MVC reference documentation to create this new controller, available here: http://static.springsource.org/spring/docs/2.0.x/reference/mvc.html.

Let's start by adding a new skeleton controller in the springapp.web package called UserListController. We can start with the following:

```
public class UserListController extends AbstractController {

	private IRODSAccessObjectFactory irodsAccessObjectFactory;

	@Override
	protected ModelAndView handleRequestInternal(HttpServletRequest request,
			HttpServletResponse response) throws Exception {
		ModelAndView mav = new ModelAndView("userlist.jsp");

		mav.addObject("userList", "Hello World!");
		return mav;
	}

	/**
	 * @return the irodsAccessObjectFactory
	 */
	protected IRODSAccessObjectFactory getIrodsAccessObjectFactory() {
		return irodsAccessObjectFactory;
	}

	/**
	 * @param irodsAccessObjectFactory
	 *		the irodsAccessObjectFactory to set
	 */
	protected void setIrodsAccessObjectFactory(
			IRODSAccessObjectFactory irodsAccessObjectFactory) {
		this.irodsAccessObjectFactory = irodsAccessObjectFactory;
	}

}
```

Note that this controller is configured to receive a reference to an IRODSAccessObjectFactory. This is then wired in with this addition to the springapp-servlet.xml. Note that, along with the bean definition, there is some additional configuration such that the bean name matches the URL path that will fire off the controller:

```
<!--  add a bean name handler to route requests to the user list controller -->

	<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

	<bean name="/userlist.htm" class="springapp.web.UserListController">
		<property name="irodsAccessObjectFactory" ref="irodsAccessObjectFactory"/>
  </bean>
```

The irodsAccessObjectFactory instance was wired in with Spring in step 3. Now we need to have the Controller respond to a query by generating an iRODS Access Object from the IRODSAccessObjectFactory when a request is made. For brevity's sake, we will take one short cut. Instead of putting together a login service, we will assume that an IRODSAccount has been created from a login page, and this IRODSAccount object has been stored in the session context. For this demo, we will create the IRODSAccount during the request processing, as this is outside of the scope of understanding how to configure an iRODS web application.

So now, our controller method looks something like this:

```
@Override
protected ModelAndView handleRequestInternal(HttpServletRequest request,
                    HttpServletResponse response) throws Exception {
            ModelAndView mav = new ModelAndView("userlist.jsp");

            IRODSAccount irodsAccount = IRODSAccount.instance("localhost", 1247,
                                "test1", "test", "", "test1", "test1-resc");
            UserAO userAO = irodsAccessObjectFactory.getUserAO(irodsAccount);
            List<User> users = userAO.findAll();

            mav.addObject("userList", users);
            return mav;
}
```

The handleRequestInternal method will fire when the /userlist.htm URL is encountered. This method will use the injected IRODSAccessObjectFactory to create a UserAO Access Object that is connected to the iRODS zone specified in the IRODSAccount object given as a parameter to the irodsAccessObjectFactory.getUserAO() method. At the time that UserAO is created, the IRODSSession is consulted to open or reuse a connection to the iRODS server. The findAll() method of UserAO will return a List of User domain objects. Since these User objects are just plain java objects, we can pass them to the rendering JSP page for presentation. We will put this JSP in place in a later step.

As you can see, the bulk of this tutorial has dealt with the configuration necessary to introduce an IRODSAccessObjectFactory into the servlet controller handler method. Once this is done, the actual Jargon code is quite minimal. What remains is the management of the connection to iRODS. It is a recommended practice to follow a session per request pattern in mid-tier development. That is, connections should be opened and closed in the span of processing one web request. As stated earlier, the IRODSProtocolManager has the responsibility of managing caches or pools of connections, and any optimization that is desired in managing connection life-cycles should be accomplished there. For our purposes, we will need to close the iRODS connection after the request finishes. We can use the interceptingFilter pattern [11]. This pattern allows request processing to be wrapped with pre and post-processing. Pre-processing can be used to check for authentication and to retrieve a stored IRODSAccount from the servlet session context. Post-processing can be used to ensure that all connections to iROD are closed. We will implement the latter case in this tutorial.

We will create an interceptor filter implementation using the Spring framework, the filter will look like this:

```
public class SessionClosingInterceptor extends HandlerInterceptorAdapter {

        private IRODSAccessObjectFactory irodsAccessObjectFactory;

        /*
         * (non-Javadoc)
         *
         * @see
         * org.springframework.web.servlet.handler.HandlerInterceptorAdapter#postHandle
         * (javax.servlet.http.HttpServletRequest,
         * javax.servlet.http.HttpServletResponse, java.lang.Object,
         * org.springframework.web.servlet.ModelAndView)
         */
```

```
	@Override
	public void postHandle(HttpServletRequest request,
			HttpServletResponse response, Object handler,
			ModelAndView modelAndView) throws Exception {
		irodsAccessObjectFactory.closeSessionAndEatExceptions();
		super.postHandle(request, response, handler, modelAndView);
	}

	/**
	 * @return the irodsAccessObjectFactory
	 */
	public IRODSAccessObjectFactory getIrodsAccessObjectFactory() {
		return irodsAccessObjectFactory;
	}

	/**
	 * @param irodsAccessObjectFactory
	 *		the irodsAccessObjectFactory to set
	 */
	public void setIrodsAccessObjectFactory(
			IRODSAccessObjectFactory irodsAccessObjectFactory) {
		this.irodsAccessObjectFactory = irodsAccessObjectFactory;
	}

}
```

There are two key elements to this filter. First, the filter takes an IRODSAccessObjectFactory as an injected dependency. Second, the filter hs implemented the postHandle() method, and in this method, a call is made to the IRODSAccessObjectFactory closeSessionAndEatExceptions() method. If you recall, the IRODSSession object keeps a ThreadLocal cache of connections. Since this filter method will be invoked by the same Thread that is processing the servlet request, the close method will clear any connections in the ThreadLocal cache that were opened during the request processing.

In order to wire our interceptor into our application, we will update the springapp-servlet.xml file so that it reflects the following state:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

 <bean name="/hello.htm" class="springapp.web.HelloController"/>

 <bean id="irodsConnectionManager"
		class="org.irods.jargon.core.connection.IRODSSimpleProtocolManager"
		factory-method="instance" init-method="initialize" destroy-method="destroy">
	</bean>

	<bean id="irodsSession"
		class="org.irods.jargon.core.connection.IRODSSession" factory-method="instance">
		<constructor-arg
			type="org.irods.jargon.core.connection.IRODSProtocolManager"
ref="irodsConnectionManager" />
	</bean>

	<bean id="irodsAccessObjectFactory"
		class="org.irods.jargon.core.pub.IRODSAccessObjectFactoryImpl">
		<constructor-arg ref="irodsSession"></constructor-arg>
```

```
        </bean>

        <!--  add a bean name handler to route requests to the user list controller -->

        <bean id="handlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
         <property name="interceptors">
             <list>
                 <ref bean="sessionClosingInterceptor"/>
             </list>
         </property>
        </bean>

        <bean name="/userlist.htm" class="springapp.web.UserListController">
               <property name="irodsAccessObjectFactory" ref="irodsAccessObjectFactory"/>
  </bean>

  <!--  connection management interceptor -->
   <bean id="sessionClosingInterceptor"
          class="springapp.web.SessionClosingInterceptor">
        <property name="irodsAccessObjectFactory" ref="irodsAccessObjectFactory"/>
    </bean>
</beans>
```

You can note the addition of a 'sessionClosingInterceptor' bean, provisioned with a reference to the IRODSAccessObjectFactory, and this interceptor has been added to the handlerMapping bean definition.

Step 5 – Displaying iRODS Data on a JSP

A final step in our demonstration will be to use the POJO domain model in a JSP page, filling in a browser view of iRODS data. This is beyond demonstrating iRODS and Jargon, but useful to see the entire interaction.

In order to display our iRODS user list, we will create a JSP view corresponding to the view specified in our controller in step 4. This userlist.jsp file should look like the following:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
        <head>
<title>User Listing</title>
        </head>
        <body>
               <h1>iRODS User Listing</h1>

               <table border="1">
                      <thead>
                             <td>User Id</td>
                             <td>User Name</td>
                             <td>User Zone</td>
                      </thead>
                      <tbody>

                             <c:forEach items="${userList}" var="user">
                                    <tr>
                                           <td><c:out value="${user.id}" /></td>
                                           <td><c:out value="${user.name}" /></td>
```

```
                                    <td><c:out value="${user.zone}" /></td>
                                </tr>
                            </c:forEach>
                        </tbody>
                    </table>

        </body>
</html>
```

Note that we have added the JSTL tag library to our project. We need to add a dependency on JSTL to our pom.xml, as follows:

```
<dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>jstl</artifactId>
                <version>1.2</version>
</dependency>
```

The JSP simply takes the userList produced by Jargon, and displays an HTML table row for each User object, referring to various fields. The resulting HTML page will render as follows:

iRODS User Listing

User Id	User Name	User Zone
105531	addUserUpdatedInfoCommentZone	test1
10018	test3	test1
9002	rodsBoot	test1
105578	testRemoveUserFromGroupUserNotInGroup	test1
105575	testRemoveUserFromGroupUserNotExists	test1
105519	addUserTestUser	test1
9001	rodsadmin	test1
105556	testAddUserToGroupUserDoesNotExist	test1
10007	rods	test1
105528	addUserUpdatedZone	test1
105559	testAddUserToGroupTwice	test1
10004	public	test1
10012	test1	test1
105534	addUserUpdatedInfoCommentZoneNullZone	test1
105568	testAddDuplicateUserGroup	test1
105522	addUserUpdateComment	test1
10015	test2	test1
105525	addUserUpdateInfoTest	test1
105540	addUserDuplicateUser	test1

At this point, the request is rendered, and the interceptor filter we created has cleaned up any open connections. This completes the tutorial on building web interfaces with Jargon.

Conclusion

This tutorial is not intended to serve as a general web development primer. By going through the steps described here, you will have:

- Configured a Maven web project to include the Jargon libraries
- Used Spring to wire together foundational components of Jargon
- Utilized the foundational Jargon components, including the IRODSAccessObjectFactory, to obtain and invoke services in iRODS
- Utilized POJO domain objects to render iRODS data in an interface
- Used an intercepter model to manage a session-per-request model for iRODS connections

Using these elements, you may develop iRODS interfaces using many different frameworks and languages. The same concepts will apply whether using Struts, JSF, another MVC framework, or even REST-ful development frameworks. These concepts also apply when using Jargon with other dynamic JVM languages, such as Grails, or Clojure. For example, here is a snippet from the iDrop-web [12] project, illustrating a controller in the Grails language:

```
class SharingController {

	IRODSAccessObjectFactory irodsAccessObjectFactory
	IRODSAccount irodsAccount

	/**
	 * Interceptor grabs IRODSAccount from the SecurityContextHolder
	 */
	def beforeInterceptor = {
		def irodsAuthentication = SecurityContextHolder.getContext().authentication

		if (irodsAuthentication == null) {
			throw new JargonRuntimeException("no irodsAuthentication in security context!")
		}

		irodsAccount = irodsAuthentication.irodsAccount
		log.debug("retrieved account for request: ${irodsAccount}")
	}

	def afterInterceptor = {
		log.debug("closing the session")
		irodsAccessObjectFactory.closeSession()
	}

	/**
	 * Load the acl details area, this will show the main form, and subsequently, the table will be loaded via AJAX
	 */
	def showAclDetails = {

		def absPath = params['absPath']
		if (absPath == null) {
			log.error "no absPath in request for showAclDetails()"
			def message = message(code:"error.no.path.provided")
			response.sendError(500,message)
		}
```

```
                log.info("showAclDetails for absPath: ${absPath}")

                CollectionAndDataObjectListAndSearchAO collectionAndDataObjectListAndSearchAO =
irodsAccessObjectFactory.getCollectionAndDataObjectListAndSearchAO(irodsAccount)
                def retObj = collectionAndDataObjectListAndSearchAO.getFullObjectForType(absPath)
                def isDataObject = retObj instanceof DataObject
                boolean getThumbnail = false

                if (isDataObject) {
                        String extension = LocalFileUtils.getFileExtension(retObj.dataName).toUpperCase()
                        log.info("extension is:${extension}")

                        if (extension == ".JPG" || extension == ".GIF" || extension == ".PNG" || extension == ".TIFF" ||
extension == ".TIF") {
                                getThumbnail = true
                        }
                }

                render(view:"aclDetails",model:[retObj:retObj, isDataObject:isDataObject, getThumbnail:getThumbnail])
        }
```

In this example, note the familiar intercepter pattern, in this case obtaining an IRODSAccount from Spring Security, and closing the open connections at the end of request processing. The IRODSAccessObjectFactory is again used, as in the showACLDetails() method, to create Access Objects which can interact with iRODS services.

With the basic elements described above in place, developing custom interfaces for iRODS becomes an approachable task, and a task that can be tackled using the standard frameworks, patterns, and tools that represent the current state of the art in web and client development.

References

1. iRODS: integrated Rule Oriented Data System White Paper, Data Intensive Cyber Environments Group, University of North Carolina at Chapel Hill, University of California at San Diego September 2008 Rajasekar, A., M. Wan, R. Moore, W. Schroeder
2. Jargon, A Java client API for the DataGrid, https://code.renci.org/gf/project/jargon/
3. Enhancing IRODS Integration: Jargon and an Evolving IRODS Service Model. Data Intensive Cyber Environments Group, University of North Carolina at Chapel Hill, iRODS User Meeting – March 24-26, 2010, http://irods.org/pubs/Meeting_1003/irods_meeting_1003_evolution_mconway2.pdf
4. Maven, http://maven.apache.org/
5. Spring Framework, http://www.springsource.org/spring-framework#documentation
6. Inversion of control containers and the dependency injection pattern, http://www.itu.dk/courses/VOP/E2006/8_injection.pdf [PDF], M Fowler - Actualizado el – itu.dk
7. Spring MVC in Maven, http://eureka.ykyuen.info/2010/03/07/spring-mvc-in-maven/
8. Packing/Unpacking Scheme Used in iRODS Mike Wan, DICE
9. iRODS Packing Instruction Documentation, https://www.irods.org/index.php/Packing_Instruction_Documentation
10. Java Servlets, http://www.informit.com/articles/article.aspx?p=170963&seqNum=6
11. InterceptingFilter Design Pattern, http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html
12. iDrop-web project, https://code.renci.org/gf/project/irodsidrop/