Creating RESTful Webservices with Spring MVC & Domain Objects

Introduction

The goal of this document is to provide a simple guide of how to implement RESTful Web Services using Spring MVC, Java Domain Objects, and Hibernate. Through examples, we will hopefully display various good practices and standards for future developers to use when implementing related Web Services. Along the way we will also cover basic Spring and Spring MVC concepts that will be essential for implementing said Web Services.

The first part of this document will consist of a review of simple terms and concepts that we will need before jumping into actual code examples. After the review we will begin implementing our simple example of a Web Service using Spring MVC. The code examples will be for the most part based around the Spring MVC Controller. We will start with a very simple Controller example which will evolve and become more complex as the document goes on. The code examples included in this document will be for the most part based around the TM Domain Object TmOrder and related Domain Objects such as TmClient, TmAddress, and TmOrderLine.

Basic Topic & Concept Review

First things first. Before jumping straight into implementing RESTful webservices, we need to nail down some simple terms and concepts to build upon during the remainder of document. This way we will all be on the same page.

What is REST and what constitutes a RESTful Webservice?

REST (Representational State Transfer) is an architectural style (not a standard, specification, or technology) concerning the concepts of servers and clients with the focus of resources and resource state. In a "RESTful" environment a Client would request information about the state of a resource from the Server and the Server would respond to the Client with information regarding the state of the specified resource. Each resource being the source of some stored information which would be given its own URI (universal resource identifier) for the purpose or data manipulation (through CRUD actions in our case). RESTful web services would take advantage of much of the already existing HTTP concepts and functionality including HTTP methods, HTTP Return Statuses, and HTTP Request/Response Headers. RESTful Requests would explicitly use HTTP methods which would map to CRUD methods in the following manner:

HTTP Method	CRUD Action
GET	Read / Retrieve
POST	Create
PUT	Update
DELETE	Delete

REST also uses a convention for URIs. For an example resource Resource model the URI pattern may resemble a URL like this: "`http://myhost:4500/resources`". In addition each of the HTTP methods may map to URL conventions in the following form:

HTTP Method	Example Resource URI
GET	http://myhost:4500/resources or http://myhost:4500/resources/1
POST	http://myhost:4500/resources
PUT	http://myhost:4500/resources/1
DELETE	http:/.myhost:4500/resources/1

What is MVC?

MVC is a commonly used architectural pattern for user interfaces based around three main object roles; **Models**, **Views**, and **Controllers**. The roles are used as follows:

- A **Model** is a Resource or Domain Object that is used to execute a desired operation of CRUD action upon. The Models for our purposes will be RedPrairie's already existing Domain Objects.
- A **View** is the visual representation of the Model displayed to the user via the client. The Views for our purposes will be handled strictly by the Spring MVC framework.
- A **Controller** is the handler for all User Requests. It decides which action is to be performed upon the model and performs it. This document will mainly be concerned with implementing Controllers.

The following image depicts the structural relationship between the three roles where the dashed arrows imply a role having knowledge of the another roles.

▪

What is Spring MVC?

Spring MVC is a Spring framework that uses MVC style and concepts for web service support and implementation. The framework is based around Spring's Dispatcher Servlet. The Dispatched Servlet is the main orchestration object for Request and Response Handing between server and client. Based on the Http Request the Dispatcher Servlet receives, it will hand off the request to the proper Controller method for processing and execution. When the request is finished being processed, the Dispatcher Servlet will also take care of creating a Http Response object to be passed back to the client.

Presently MOCA has already added support for Spring MVC and Web Services in it's trunk. This support allows a user to register their custom Controllers with the Dispatcher Servlet and MOCA by simply declaring their Controller class or classes as Beans within the Spring Application Context. Once the Controller is defined as a bean in the Application Context, it will automatically be registered and eventually used. Later on in the document I will explain how to register your Controller.

What are Domain Objects?

Domain Objects, or more specifically Java Domain objects are POJO-esque (Plain Old Java Objects) objects that represent data stored in one or more tables in a database. Some examples of these Domain Objects in the RedPrairie's source code include TmOrder, TmClient, or TmAddress. As explained at the beginning of this document the code examples in this document will for the most part be related to TmOrder. The Domain Objects will be instantiated using data stored in the database by using Hibernate and DAO's to access the data.

What is Hibernate? And what are DAOs?

Hibernate is one implementation of the data access layer based around DAO's (Data Access Objects) used to access data stored in databases

and eventually to build persistent Java objects. In order to access the data, Hibernate maps Relation Database Data to Object Oriented Domain Models (in our case the TM models) with the intent of eventually performing CRUD operations upon them. For most developers and users, this layer will be hidden or abstracted out into simply generic Data Access Objects.

Luckily for us, these pieces of the puzzle have already been implemented in TM (the TmOrder DAO has been implemented using the interface TmOrderDAO and implementation file TmHibernateOrderDAO). For this reason, not much time will be spent on Hibernate.

Basic Web Application & Spring Concepts

The WAR File and Application Deployment

The WAR File is a Web Application Archive file (much like a JAR file) is a file type that is used to deploy the web applications or services. This file is built using a "`web.xml`" file which defines the structure and configuration of the web application along with the type of servlet the web application will use.

For building our WAR files, MOCA will by default provide the "`web.xml`" file which specifies the use of Spring Dispatcher Servlet instance. This file also references context configuration files for our new Dispatcher Servlet instance; "`webservices.xml`" and "`webservices-config.xml`". These two files define how the Spring Application Context is to be populated. The "`webservices-config.xml`" defines MOCA related application information that every servlet will need. The "`webservices.xml`" file captures custom web service or servlet related information. I will discuss the "`webservices.xml`" file and how it will be used for our purposes in the next section in more depth. The WAR file may also contain any classes related to the web service application such as the custom Controllers.

Once the WAR file is created for the web application (generally it will be created using a build script and placed in the products "`webdeploy`" directory) the file needs to be copied to the environments "`LESDIR/webdeploy`" directory to be registered by MOCA. This directory is scanned at a regular interval (generally every 10 seconds) by MOCA to initialize web application instances.

The Spring Configuration File and The Application Context

The Spring Application Context is a collection of beans that can be instantiated at run-time while running an application. In our case we want to have our custom Controllers be one of the beans specified in the Application Context. This collection of beans is configured using a Spring Configuration file. Generally Spring users define these beans using an XML file. In our case, the Application Context will be built using the "`webservices.xml`" file in the web application's WAR file.

MOCA defines a default "`webservices.xml`" file that performs a component scan on the package "`com.redprairie`" searching for any classes that are Controllers (classes that are annotated with the @Controller annotation) and adding as beans in the Application Context. MOCA is setup in this fashion because the way that the Dispatcher Servlet is configured, any Controllers that are defined in the Application Context are automatically registered for use in the web application.

For our custom implementation we are going to override MOCA's "`webservices.xml`" file in order to perform a component scan on the package "`com.redprairie.tm.tmws`" searching for classes that are Configuration Files (classes that are annotated with the @Configuration annotation). This way we can use a Spring Java Configuration file to manually define beans that we will need in our web service's Application Context, including all of the Controllers. I will show the evolution of the Java Configuration file that we will be using along side of the evolution of the Controller we will be implementing.

Commonly Used Spring MVC Annotations

Before getting into the Controller code here are a few annotations commonly used in Spring MVC and what the function of each is:

Spring MVC Annotation	The Annotation's Function
@Controller	This class type annotation informs Spring's Dispatcher Servlet that the annotated class is plays the MVC role of Controller. This means that the Dispatcher Servlet can map and pass HTTP Requests to methods in this class to be processed.
@Configuration	This class type annotation informs Spring that the annotated class is a Configuration file. This informs Spring that beans that are defined in this class are to be used in the Application Context and can be instantiated at run-time
@RequestMapping	This class type or method annotation is a helper annotation which helps Spring's Dispatcher Servlet map incoming HTTP Requests to the proper Controller methods. There are many options within this annotation which can help in the mapping process but the main two will be value and method. The value option of this annotation refers to a URI extension or pattern to match from the HTTP request. The method signifies which HTTP Request Methods can be mapped to the specific class methods.
@RequestBody	This Controller method parameter annotation signifies that the contents of the HTTP Request Body are to be injected into the annotated method parameter field. For example, a method may have the signature "public String getMyStuff(@RequestBody HttpServletRequest request)" which means that the HTTP Request's Body's contents will be injected into the HttpServletRequest field to be used in the method.

@ResponseBody	This annotation can be used as a Controller method return type or Controller method parameter annotation. When used as a Controller method return type annotation, this means that the object returned by the Controller method is to be injected into the HTTP Response Body to be returned to the client. When used as a Controller method parameter annotation, it simply means that the HTTP Response body is to be injected into the parameter of the method that is annotated. This will generally be used to manually set information on the Response Body.
@PathVariable	This Controller method parameter annotation is used to extract values from the HTTP Request's URI to be used in the Controller method. URI Templates and Regular Expressions are used to extract these values and eventually inject them into the method parameter that is annotation with this annotation.
@ResponseStatus	This method annotation tells Spring's Dispatcher Servlet to inject this specified Response Status into the HTTP Response's Response Status Header provided that the method was executed successfully.

The Simple Resource Controller

The Resource Controller Code

Now after all that high level conceptual information we are finally ready to see some code examples. Here is a theoretical example (Resource Controller cannot be found anywhere in the source code) of what a simple bare bones Resource Controller may look like. The following code is of course assuming that there are implementations of the Resource Model, the ResourcePK Primary Key object, and the ResourceDAO data access object in the source code.

ResrouceController.java

```
@Controller
@RequestMapping("/resources")
public class ResourceController {

    public ResourceController(ResourceDAO dao) {
        _dao = dao;
    }

    @RequestMapping(value = "", method = RequestMethod.GET)
    public @ResponseBody
    List<Resource> getResources() {
        return _dao.readAll();
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public @ResponseBody
    Resource getResource(@PathVariable("id") Integer id) {
        return _dao.read(new ResourcePK(id));
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateResource(@PathVariable("id") Integer id,
@RequestBody Resource resource) {
        if (resource.getID() == null || resource.getID() != id) {
            throw Exception("The Resource provided does not match
the URI specified Resource ID.");
        }
        _dao.save(resource);
    }

    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public void createResource(@RequestBody Resource resource) {
        _dao.save(resource);
    }

    @RequestMapping(value = "/{id}", method =
RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.OK)
    public void deleteResource(@PathVariable("id") Integer id) {
        Resource resource = _dao.read(new ResourcePK(id));
        if (resource != null) {
            _dao.delete(resource);
        }
    }

    private final ResourceDAO _dao;

}
```

The Java Resource Configuration File

Here is a simple example of what the Spring Java Configuration file would like using the Resource Controller above. At this point only two beans are defined. We define a ResourceDAO bean to used as a constructor argument for our ResourceController.

ResourceConfiguration.java

```
@Configuration
public class ResourceConfiguration {

    @Bean
    public ResourceController resourceController() {
        return new ResourceController(resourceDAO());
    }


    @Bean
    public ResourceDAO resourceDAO() {
        return new ResourceDAO(null);
    }

}
```

What does this Code Actually Do?

This Controller is configured so that any HTTP Requests that are sent to the Dispatcher Servlet in the form of "http://<server-host>/resources/*" will be handled by methods in this Controller. This Controller accounts for examples of all of the CRUD operations to be performed on the Resource Model.

The first method "getResources" uses the DAO to read and returns a List containing all of the Resources from the database. This method is an examples of the Read/Retrieve CRUD action or the GET HTTP method. Any GET Request to URL's in the form "`http://<hostname>:<hostport>/resources`" will be mapped to this method and return the list of Resources.

The second method "getResource" uses the DAO to read and return a single Resource from the database that is specified by an Integer retrieved from the URL to be used as a Resource ID. This method is also an example of the Read/Retrieve CRUD action. Any GET Request to URLs in the form will "`http://<hostname>:<hostport>/resources/1`" be mapped to this method and return in this case the Resource with the Resource ID of 1.

The third method "updateResource" takes method parameters of an Integer specified in the URL to be used as a Resource ID value, and a Resource Model specified in the Request body holding the values on the Resource to be updated. This method is an example of the Update CRUD action and the HTTP PUT method. This simple method just does a simple check to make sure that the method parameters make sense, and then uses the DAO to save the provided Resource Model. By convention this method returns void and sets a HTTP Response Status of "No Status" or 204 if executed successfully. Any PUT Request to URLs in the form of "`http://<hostname>:<hostport>/resources/1`" with a Request Body containing a Resource will be mapped to this method and in this case update the Resource with a Resource ID of 1.

The fourth method "createResource" takes a parameter of a Resource Model provided in the HTTP Request Body. This method is an example of the Create CRUD action and the HTTP POST method. This method simply uses the DAO to save the Resource Model given values in the database. By convention the method returns the HTTP Response Status "Created" or 201. Any POST Request to URLs in the form of "`http://<hostname>:<hostport>/resources`" with a HTTP Request body containing a Resource value will be mapped to this method and will simply create a new Resource entry in the database with the provided Resource Model.

The final method "deleteResource" takes a single Integer parameter specified in the URL to be used as the Resource ID. This method is an example of the Delete CRUD action and the HTTP DELETE method. This method tries find the Resource with the Resource ID specified. If the Resource exists, the method will use the DAO to delete the resource, otherwise the rest of the method is ignored. By convention this method returns the HTTP Response Status of "OK" or 200. Any DELETE Requests to URLs of the form "`http://<hostname>:<hostport>/resources/1`" will be mapped to this method and in this case delete the Resource with a Resource ID of 1.

The Initial TmOrder Controller

The TmOrder Controller Code

Now lets try converting the Resource Controller into a code example that uses code we can actually find in RedPrairie's source code, TmOrder. Here is what a simple Order Controller would look like:

OrderController.java

```
@Controller
@RequestMapping("/orders")
public class OrderController extends AbstractController<TmOrder,
OrderPK> {

    public OrderController(TmOrderDAO dao) {
        _dao = dao;
    }

    @RequestMapping(value = "", method = RequestMethod.GET)
    public @ResponseBody
    List<TmOrder> getTmOrders() {
        return _dao.readAll();
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public @ResponseBody
    TmOrder getTmOrder(@PathVariable("id") Integer id) {
        return _dao.read(new OrderPK(id));
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateTmOrder(@PathVariable("id") Integer id,
@RequestBody TmOrder order) {
        if (order.getID() == null || order.getID() != id) {
            throw Exception("The Order provided does not match the
URI specified Order ID.");
        }
        _dao.save(order);
    }

    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public void createTmOrder(@RequestBody TmOrder order) {
        _dao.save(order);
    }

    @RequestMapping(value = "/{id}", method =
RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.OK)
    public void deleteTmOrder(@PathVariable("id") Integer id) {
        TmOrder order = _dao.read(new OrderPK(id));
        if (order != null) {
            _dao.delete(order);
        }
    }

    private TmOrderDAO _dao;

}
```

This code example has the exact same functionality as the Resource Controller except for the fact that it uses TmOrder as a Model for CRUD operations. We use in this example the OrderPK type for TmOrder Primary Keys and the TmHibernateOrderDAO for the Tm Order data access object. All Requests that use URLs of the form "`http://<hostname>:<hostport>/orders/*`" will be mapped to this Controller for processing.

The Java Order Configuration File

OrderConfiguration.java

```
@Configuration
public class OrderConfiguration {

    @Bean
    public OrderController orderController() {
        return new OrderController(orderDAO());
    }

    @Bean
    public TmOrderDAO orderDAO() {
        return new TmHibernateOrderDAO(null);
    }

}
```

Using Projection Proxies For Return Objects

Problem

What if for your return objects you don't care about all of the fields on the TmOrder? There are after all dozens of fields on the TmOrder Domain Model. What if in fact you don't care about any of the fields and instead just want to use links?

Solution

A solution to this issue it to use Projection Proxy Objects. When using Projection Proxies as return objects for Controller methods it is easy to return a subset of the fields found on the TmOrder Model. A Projection Proxy Object is a CGLIB proxy object that is made up of three things:

- A Projection Interface
- An underlying data object.
- A Projection Proxy Method Interceptor

A **Projection Interface** is a Java Interface the extends the placeholder interface TmWebResource. The TmWebResource interface signifies that the Projection Proxy is to not be treated normally but to be treated as a Web Resource. The Projection Interface is made up of a subset of the getter methods found on the Domain Object thus being a Projection of the Domain Object. An example of what might be used as a Projection Interface for a TmOrder object is:

A Sample TmOrder Projection Interface

```
public interface Order extends TmWebResource {
      public Integer getID();

      public String getOrderNumber();

      public String getTmsPlanningStatus();

      public String getWarehouseID();
}
```

The Projection Interface outlines which fields from the Domain Object are desired to be included in the return object. Each getter method found in the Projection Interface correlates to a field on the Domain Object. In the above example the fields ID, orderNumber, tmsPlanningStatus, and warehouseID are the fields to be put in the return object.

The **Underlying Data object** can be any Java object but for our purposes we will limit it to being a Domain Object.

A method interceptor is CGLIB object which will intercept any method called on the proxy for custom processing. A **Projection Proxy Method Interceptor** will intercept any of the Projection Interface methods called and handle them accordingly. The Projection Interface methods will generally be called when serializing the data into the output format. The default Projection Proxy Method Interceptors map methods declared on the Projection Interfaces to methods on the underlying data objects. Not all methods are mapped. The default Interceptors will only map methods that are named in the form of "getSomething" or "isSomething". Once the Projection Interface method is mapped to the underlying data method, we can then access the underlying data values that we wish to return. The default Method Interceptors map methods between interface and object using reflection and naming conventions. Therefor when using the default Interceptors, all of the methods on the Projection Interface must be equivalent to the corresponding method on the Domain Object except for return type. Return Type is allowed to differ in order to allow the Projection Proxies to have nested Projection Proxy objects.

So for example, say we are using the Projection Interface above, a TmOrder object, and the default Projection Proxy Method Interceptors, and we create a Projection Proxy. Then, when we call the "getID()" method on the newly created Projection Proxy, what would be returned is the ID of the underlying TmOrder object.

The Updated Order Controller

OrderController.java Using Projection Proxies

```
@Controller
@RequestMapping("/orders")
public class OrderController extends AbstractController<TmOrder,
OrderPK> {

    public OrderController(TmOrderDAO dao, ProxyProvider
proxyProvider) {
        _dao = dao;
        _proxyProvider = proxyProvider;
    }

    @RequestMapping(value = "", method = RequestMethod.GET)
    public @ResponseBody
    List<Order> getTmOrders() throws ProxyCreationException {
        List<Order> orders = new ArrayList<Order>();
        for (TmOrder order : _dao.readAll()) {
            orders.add(_proxyProvider.createProxy(Order.class,
```

```
order);
        }
        return orders;
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public @ResponseBody
    Order getTmOrder(@PathVariable("id") Integer id) throws
ProxyCreationException {
        TmOrder order = _dao.read(new OrderPK(id));
        return _proxyProvider.createProxy(Order.class, order);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateTmOrder(@PathVariable("id") Integer id,
@RequestBody TmOrder order) {
        if (order.getID() == null || order.getID() != id) {
            throw Exception("The Order provided does not match the
URI specified Order ID.");
        }
        _dao.save(order);
    }

    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public void createTmOrder(@RequestBody TmOrder order) {
        _dao.save(order);
    }

    @RequestMapping(value = "/{id}", method =
RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.OK)
    public void deleteTmOrder(@PathVariable("id") Integer id) {
        TmOrder order = _dao.read(new OrderPK(id));
        if (order != null) {
            _dao.delete(order);
        }
    }

    public interface Order extends TmWebResource {
        public Integer getID();

        public String getOrderNumber();

        public String getTmsPlanningStatus();

        public String getWarehouseID();
    }

    private final TmOrderDAO _dao;

    private final ProxyProvider _proxyProvider;
```

```
}
```

The **ProxyProvider** simply an object that creates a Projection Proxy given a Projection Interface and an underlying data object. The method interceptor is determined by searching through a map of classes to Method Interceptor types to see if there are custom defined Method Interceptors for the Class type that the proxy is being created for.

The Updated Spring Java Configuration File

The Spring Java Configuration File includes the additional beans for the ProxyProvider and the default Proxy Interceptor class ProxyInterceptor. This configuration assumes that the ProxyProvider extends the interface ApplicationContextAware and explicitly asks for the ProxyInterceptor bean when creating proxies.

OrderConfiguration.java Using Projection Proxies

```
@Configuration
public class OrderConfiguration {

    @Bean
    public OrderController orderController() {
        return new OrderController(orderDAO(), proxyProvider());
    }

    @Bean
    public TmOrderDAO orderDAO() {
        return new TmHibernateOrderDAO(null);
    }

    @Bean
    public ProxyProvider proxyProvider() {
        return new ProxyProvider();
    }

    @Bean
    @Scope("prototype")
    public ProxyInterceptor proxyInterceptor() {
        return new ProxyInterceptor(proxyProvider());
    }

}
```

Providing Query Paramters

Problem

In a SQL query, one can provide query parameters to filter the results down to a set of results that matches a given criteria. What if i want that same functionality in the web service I am creating?

Solution

By using the HttpServletRequest query string field we can provide query parameters in the URL to be used much like the query parameters in an SQL query. For instance, for an example URL "http://myhost:4500/ws/tm/orders?client=MY_CLIENT&shipToAddress=MY_ADDRESS", the portion of the URL after the question mark would be considered the query string. By formatting the query string into valid arguments to be used with the Hibernate query and also implementing a new Hibernate method for using the arguments, we can provide the same functionality that SQL queries provide with query parameters.

The new Hibernate method will be declared in the TmGenericDAO interface and defined in TmHibernateGenericDAO. This code example will not be the final version of this method seeing as the method only adds equality restrictions, but this is a good example to see how a method of this type might work.

TmHibernateGenericDAO.java

```
public class TmHibernateGenericDAO<TEntity extends
TmPrimaryKey<PK>,
        PK extends Serializable> implements TmGenericDAO<TEntity,
PK>, TmHibernateSessionSpecificDAO {

    ...

    public List<TmOrder> readAllWithArguments(Map<String, Object>
arguments) {
        Criteria criteria =
_hibernateSession.createCriteria(_instanceType);

        for (Entry<String, Object> entry : arguments.entrySet()) {
             criteria.add(Restrictions.eq(entry.getKey(),
entry.getValue()));
        }

        List<TmOrder> allObjects = criteria.list();
        return allObjects != null ? allObjects : new
ArrayList<TmOrder>(1);
    }

   ...

    protected Class<? extends TEntity> _instanceType;

    protected Session _hibernateSession;

}
```

With this piece in place we can update the Order Controller to take advantage of the query parameter functionality.

The Updated Order Controller

OrderController.java Using Query Parameter Functionality

```
@Controller
@RequestMapping("/orders")
```

```
public class OrderController extends AbstractController<TmOrder,
OrderPK> {


    public OrderController(TmOrderDAO dao, ProxyProvider
proxyProvider) {
        _dao = dao;
        _proxyProvider = proxyProvider;
    }

    @RequestMapping(value = "", method = RequestMethod.GET)
    public @ResponseBody
    List<Order> getTmOrders(HttpServletRequest request) throws
ProxyCreationException {
        Map<String, Object> argMap =
formatArguments(request.getParameterMap());

        List<Order> orders = new ArrayList<Order>();
        for (TmOrder order : _dao.readAllWithArguments(argMap)) {
            orders.add(_proxyProvider.createProxy(Order.class,
order);
        }
        return orders;
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public @ResponseBody
    Order getTmOrder(@PathVariable("id") Integer id) throws
ProxyCreationException {
        TmOrder order = _dao.read(new OrderPK(id));
        return _proxyProvider.createProxy(Order.class, order);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateTmOrder(@PathVariable("id") Integer id,
@RequestBody TmOrder order) {
        if (order.getID() == null || order.getID() != id) {
            throw Exception("The Order provided does not match the
URI specified Order ID.");
        }
        _dao.save(order);
    }

    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public void createTmOrder(@RequestBody TmOrder order) {
        _dao.save(order);
    }

    @RequestMapping(value = "/{id}", method =
RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.OK)
```

```
public void deleteTmOrder(@PathVariable("id") Integer id) {
    TmOrder order = _dao.read(new OrderPK(id));
    if (order != null) {
        _dao.delete(order);
    }
}

public interface Order extends TmWebResource {
    public Integer getID();

    public String getOrderNumber();

    public String getTmsPlanningStatus();

    public String getWarehouseID();
}

private final TmOrderDAO _dao;

private final ProxyProvider _proxyProvider;
```

```
    }
```

The functionality for the method "**formatArguments**" is hidden in the AbstractController class because of it's complexity. Conceptually all that is needed to be known about the method is that the method formats all of the incoming query parameters into valid query arguments for Hibernate. The request parameter map cannot be used directly because of cases of when complex arguments are encountered. For example, for a query concerning TmOrder objects, if a URL contained a parameter in the form "client=MY_CLIENT", Hibernate would not know how to use this information. The client with the client-id "MY_CLIENT" needs to be retrieved from the database and used an argument in the Hibernate query instead of the client-id string value.

Using Projection Proxies For Input Objects

Problem

Generally it is a good idea to use the same type of input objects as output objects. Shouldn't our web services use this idea an use input as Projection Proxies now that Projection Proxies are our output object types? But also if we are to use Projection Proxies as input, how will that work seeing as Projection Proxies are based around Interfaces?

Solution

To solve this problem we need to take advantage of Spring's Message Converter objects. A message converter is a Spring object that performs conversion between HTTP Requests or Response bodies and Java Objects. We will be using this ability to convert the contents of a HTTP Request body into a Projection Proxy Object. For this to work we will need to define our own custom Message Converter and also a new Projection Proxy Method Interceptor.

Below if the implementation of our custom message converter InputProxyMessageConverter. The message converter extends the Spring message converter MappingJacksonHttpMessageConverter which by default converts content type JSON. The new message converter reads in the contents of the request body and uses an Object Mapper to convert the request body into a Map of Strings to Objects. After this has been completed, the message converter creates a Projection Proxy for the map with the given Projection Interface type and returns the Proxy.

InputProxyMessageConverter.java

```
public class InputProxyMessageConverter extends
MappingJacksonHttpMessageConverter {

    public InputProxyMessageConverter(ProxyProvider proxyProvider)
{
        _proxyProvider = proxyProvider;
    }

    @Override
    public boolean canRead(Class<?> clazz, MediaType mediaType) {
        return supports(clazz) && canRead(mediaType);
    }

    @Override
    protected boolean supports(Class<?> clazz) {
        return TmWebResource.class.isAssignableFrom(clazz);
    }

    @SuppressWarnings("unchecked")
    @Override
    protected Object readInternal(Class<?> clazz, HttpInputMessage
inputMessage)
            throws IOException, HttpMessageNotReadableException {
        InputStream stream = inputMessage.getBody();
        Map<String, Object> map =
getObjectMapper().readValue(stream, HashMap.class);
        try {
            return _proxyProvider.createProxy(clazz, map);
        }
        catch (ProxyCreationException e) {
            throw new RuntimeException(e);
        }
    }

    private final ProxyProvider _proxyProvider;

}
```

However, for all of this to work properly a new type of Projection Proxy Method Interceptor must be defined. Presently the default Method Interceptors map Projection Interface methods to methods on the underlying data object. However, in this case the underlying data object is a Map and does not have methods that would map to the Projection Interfaces. Therefor a new type of Method Interceptor is defined to map Projection Interface methods to entries in the underlying data map. This way the underlying data values can still be retrieved for data manipulation. This new method interceptor type will be called InputProxyMethodInterceptor as it is used mainly for input Proxy types. And the former ProxyInterceptor object will be renamed OutputProxyMethodInterceptor as it is used primarily for output Proxy types.

But there is one more step before we are finished solving this problem. Now that we are using Projection Proxies as input objects, we need to implement a way to convert the proxies back into Domain Objects in order to use the DAOs. This step will be implemented in an object named ObjectProvider. Once that step is complete we are ready to show the updated version of the TmOrder Controller.

The Updated Order Controller

OrderController.java Using Input Projection Proxies

```
@Controller
@RequestMapping("/orders")
public class OrderController extends AbstractController<TmOrder,
OrderPK> {

    public OrderController(TmOrderDAO dao, ProxyProvider
proxyProvider, ObjectProvider objectProvider) {
        _dao = dao;
        _proxyProvider = proxyProvider;
        _objectProvider = objectProvider;
    }

    @RequestMapping(value = "", method = RequestMethod.GET)
    public @ResponseBody
    List<Order> getTmOrders(HttpServletRequest request) throws
ProxyCreationException {
        Map<String, Object> argMap =
formatArguments(request.getParameterMap());

        List<Order> orders = new ArrayList<Order>();
        for (TmOrder order : _dao.readAllWithArguments(argMap)) {
            orders.add(_proxyProvider.createProxy(Order.class,
order);
        }
        return orders;
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public @ResponseBody
    Order getTmOrder(@PathVariable("id") Integer id) throws
ProxyCreationException {
        TmOrder order = _dao.read(new OrderPK(id));
        return _proxyProvider.createProxy(Order.class, order);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateTmOrder(@PathVariable("id") Integer id,
@RequestBody Order order) {
        TmOrder ord =
_objectProvider.createModelFromProxy(TmOrder.class, order);

        if (ord.getID() == null || ord.getID() != id) {
            throw Exception("The Order provided does not match the
URI specified Order ID.");
        }

        _dao.save(ord);
    }
```

```
    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public void createTmOrder(@RequestBody Order order) throws
ObjectCreationException, ObjectRetrievalException {
        TmOrder ord =
_objectProvider.createModelFromProxy(TmOrder.class, order);

        if (order.getID() != null) {
            throw new
HttpClientErrorException(HttpStatus.BAD_REQUEST,
                "The user is not allowed to specify Order ID when
creating an Order.");
        }

        _dao.save(order);
    }

    @RequestMapping(value = "/{id}", method =
RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.OK)
    public void deleteTmOrder(@PathVariable("id") Integer id)
throws ObjectCreationException, ObjectRetrievalException {
        TmOrder order = _dao.read(new OrderPK(id));
        if (order != null) {
            _dao.delete(order);
        }
    }

    public interface Order extends TmWebResource {
        public Integer getID();

        public String getOrderNumber();

        public String getTmsPlanningStatus();

        public String getWarehouseID();
    }

    private final TmOrderDAO _dao;

    private final ProxyProvider _proxyProvider;

    private final ObjectProvider _objectProvider;
```

```
}
```

The functionality behind the **ObjectProvider** object is hidden because of it's complexity. The ObjectProvider essentially creates a model from the Projection Proxies by attempting to retrieve an already existing object from the database or creating a new object, and then continuing on to update the object with values set on the Projection Proxy. Reflection and naming conventions are very heavily used concepts in the ObjectProvider.

The Updated Spring Java Configuration File

Since the last version of the OrderConfiguration.java Configuration file there have been various changes. We are now including the ObjectProvider bean, the InputProxyInterceptor and OutputProxyInterceptor beans, the InputProxyMessageConverter bean, and we also have a new method for constructing a map of Classes to Projection Proxy Method Interceptors. This new method for mapping Classes to Projection Proxy Method Interceptors allows for customization in the Method Interceptors. By default the OutputProxyInterceptor and the InputProxyInterceptor will be used, however if a user wishes to customize the Method Interceptors, they may add a entry in the map mapping specific classes to custom implemented Interceptors.

OrderConfiguration.java Using Projection Proxies as Input

```
@Configuration
public class OrderConfiguration {

    @Bean
    public OrderController orderController() {
        return new OrderController(orderDAO(), proxyProvider(),
objectProvider());
    }

    @Bean
    public TmOrderDAO orderDAO() {
        return daoFactory().getTmOrderDAO();
    }

    @Bean
    public ProxyProvider proxyProvider() {
        return new ProxyProvider(interceptorMap());
    }

    @Bean
    @Scope("prototype")
    public InputProxyInterceptor inputProxyInterceptor() {
        return new InputProxyInterceptor(proxyProvider());
    }

    @Bean
    @Scope("prototype")
    public OutputProxyInterceptor outputProxyInterceptor() {
        return new OutputProxyInterceptor(proxyProvider());
    }

    public Map<Class<?>, Map<ProxyType, Class<? extends
TmInterceptor<?>>>> interceptorMap() {
        Map<Class<?>, Map<ProxyType, Class<? extends
```

```
TmInterceptor<?>>>> interceptorMap =
            new HashMap<Class<?>, Map<ProxyType, Class<? extends
TmInterceptor<?>>>>();

        // Default mapping for objects... Input and Output proxy
instances...
        Map<ProxyType, Class<? extends TmInterceptor<?>>> map =
new HashMap<ProxyType, Class<? extends TmInterceptor<?>>>();
        map.put(ProxyType.INPUT, InputProxyInterceptor.class);
        map.put(ProxyType.OUTPUT, OutputProxyInterceptor.class);
        interceptorMap.put(Object.class, map);

        return interceptorMap;
    }

    @Bean
    public ObjectProvider objectProvider() {
        return new ObjectProvider(objectFactory(), daoFactory());
    }

    @Bean
    public TmDAOFactory daoFactory() {
        return TmFactoryServer.getTmDAOFActory();
    }

    @Bean
    public TmObjectFactory objectFactory() {
        return TmFactoryServer.getTmObjectFactory();
    }

    @Bean
    public InputProxyMessageConverter inputProxyMessageConverter()
{
        return new InputProxyMessageConverter(proxyProvider());
    }
```

```
}
```

Providing Links in Return Objects

Problem

What if i would like links to be returned in the return objects? After all REST does promote the use of links...

Solution

The solution is very easy. The default OutputProxyMethodInterceptors are implemented in such a way that when a method is called on the Projection Interfaces that is annotated with the @WebResourceLink annotation, the Method Interceptor knows to return a link for the method. The OutputProxyMethodInterceptor uses a LinkBuilder object to build the links. For our purposes the process of how the link is built is not essential knowledge, we just know that it works. With the new functionality a Projection Interface may look like this:

Order.java Projection Interface Using Link Functionality

```
public interface Order extends TmWebResource {
      @WebResourceLink(RelType.self)
      public String getSelf();

      public Integer getID();

      public String getOrderNumber();

      public String getTmsPlanningStatus();

      public String getWarehouseID();
}
```

The WebResourceLink annotation takes one argument, the enumerated type RelType (Relative Link Type) defining various relative link types such as self, first, last, next, previous... Using this Projection Interface when invoking the Read-All Order Controller method may produce output resembling this JSON string:

Sample JSON Output


```
[
    {
        "orderNumber":"ORD0000401",
        "self":"http://eknapp1:4500/ws/tm/orders/30",
        "tmsPlanningStatus":"BOOKING"
        "warehouseID":"----"
        "id":30
    },
    {
        "orderNumber":"ORD0000801",
        "self":"http://eknapp1:4500/ws/tm/orders/31",
        "tmsPlanningStatus":"BOOKING"
        "warehouseID":"----"
        "id":31
    },
    {
        "orderNumber":"ORD0000802",
        "self":"http://eknapp1:4500/ws/tm/orders/32",
        "tmsPlanningStatus":"BOOKING"
        "warehouseID":"----"
        "id":32
    }
]
```


The self links will provide a link to potentially view more detailed information about each TmOrder object.

Using Fields In the Projection Proxies That Aren't On The Domain Object

Problem

What if there is a need to return a field that is not on the Java Domain Object? For example, what if there is a composite field that I would like to return?

Solution

The solution to this issue is to extend the default Output Proxy Method Interceptor. By doing this, there can be finer control over the fields in the result object. As an example, what if we had a Projection Interface that we wanted to use that was resembles:

Order.java Projection Interface With a Non-Domain Object Method

```
public interface Order extends TmWebResource {
      @WebResourceLink(RelType.self)
      public String getSelf();

      public Integer getID();

      public String getOrderNumber();

      public String getTmsPlanningStatus();

      public String getWarehouseID();

      public String getCompositeKey();
}
```

The Non-Domain Object method "getCompositeKey" could correlate with a desired field named "compositeKey" that is a concatenation of the ID, orderNumber, and warehouseID fields on the TmOrder object. You could define a new Output Proxy Method Interceptor to handle this method properly. This new Proxy Method Interceptor may resemble this sample code:

MyOutputProxyInterceptor.java

```
public class MyOutputProxyInterceptor extends
OutputProxyInterceptor<TmOrder> {

    public MyOutputProxyInterceptor(ProxyProvider proxyProvider,
            ObjectProvider objectProvider, LinkBuilder
linkBuilder) {
        super(proxyProvider, objectProvider,linkBuilder);
    }

    @Override
    public Object handleMethod(Method method) throws
ProxyCreationException,
            ProxyMethodInterceptionException,
LinkCreationException,
            ObjectCreationException {
        if (method.getName().matches("getCompositeKey")) {
            TmOrder order = getDataObject().
            return order.getID().toString() +
order.getOrderNumber() + order.getWarehouseID();
        }
        else {
            return super.handleMethod(method);
        }
    }

}
```

This Method Interceptor is limited to use only TmOrder objects as underlying data objects. But in the case that we create an Projection Proxy with a TmOrder object , the above Projection Interface, and our new Output Proxy Method Interceptor, if the method "getCompositeKey" is called on the Projection Proxy, the Method Interceptor will return a concatenation of the order ID, the orderNumber, and the warehouseID fields on the underlying data object.

This is just a simple example, but other more complicated scenarios are just as easy to show since the Method Interceptors give the user large amounts of control over the return object.

The Final Order Controller And Configuration File

OrderController.java

```
@Controller
@RequestMapping("/orders")
public class OrderController extends AbstractController<TmOrder,
OrderPK> {

    public OrderController(ProxyProvider proxyProvider,
ObjectProvider objectProvider, LinkBuilder linkBuilder) {
        super(proxyProvider, objectProvider, linkBuilder);
```

```
    }

    @RequestMapping(value = "", method = RequestMethod.GET)
    public @ResponseBody
    List<Order> getTmOrders(HttpServletRequest request) throws
ProxyCreationException {
        Map<String, Object> argMap =
formatArguments(request.getParameterMap());

        List<Order> orders = new ArrayList<Order>();
        for (TmOrder order :
getDAO().readAllWithArguments(argMap)) {
            orders.add(createProxy(Order.class, order);
        }
        return orders;
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public @ResponseBody
    Order getTmOrder(@PathVariable("id") Integer id) throws
ProxyCreationException {
        TmOrder order = getDAO().read(new OrderPK(id));
        return createProxy(Order.class, order);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void updateTmOrder(@PathVariable("id") Integer id,
@RequestBody Order order) {
        TmOrder ord = createModelFromProxy(order);

        if (ord.getID() == null || ord.getID() != id) {
            throw Exception("The Order provided does not match the
URI specified Order ID.");
        }

        getDAO().save(ord);
    }

    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseStatus(HttpStatus.CREATED)
    public void createTmOrder(@RequestBody Order order,
HttpServeletResponse response) throws ObjectCreationException,
        ObjectRetrievalException, LinkCreationException {
        TmOrder ord = createModelFromProxy(order);

        if (ord.getID() != null) {
            throw new
HttpClientErrorException(HttpStatus.BAD_REQUEST,
                "The user is not allowed to specify Order ID when
creating an Order.");
        }
```

```
        ord.setID(((TmOrderDAO) getDAO()).generateNextOrderID());
        getDAO().save(ord);

        response.addHeader("location", createLink(RelType.self,
ord).getHref());
    }

    @RequestMapping(value = "/{id}", method =
RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.OK)
    public void deleteTmOrder(@PathVariable("id") Integer id)
throws ObjectCreationException, ObjectRetrievalException {
        TmOrder order = getDAO().read(new OrderPK(id));
        if (order != null) {
            getDAO().delete(order);
        }
    }

    public interface Order extends TmWebResource {
        @WebResourceLink
        public String getSelf();

        public Integer getID();

        public String getOrderNumber();

        public String getTmsPlanningStatus();

        public String getWarehouseID();
    }
```

```
}
```

OrderConfiguration.java

```
@Configuration
public class OrderConfiguration {

    @Bean
    public OrderController orderController() {
        return new OrderController(proxyProvider(),
objectProvider(), linkBuilder());
    }

    @Bean
    public ProxyProvider proxyProvider() {
        return new ProxyProvider(interceptorMap());
    }

    @Bean
    @Scope("prototype")
    public InputProxyInterceptor inputProxyInterceptor() {
        return new InputProxyInterceptor(proxyProvider(),
objectProvider());
    }

    @Bean
    @Scope("prototype")
    public OutputProxyInterceptor outputProxyInterceptor() {
        return new OutputProxyInterceptor(proxyProvider(),
linkBuilder());
    }

    public Map<Class<?>, Map<ProxyType, Class<? extends
TmInterceptor<?>>>> interceptorMap() {
        Map<Class<?>, Map<ProxyType, Class<? extends
TmInterceptor<?>>>> interceptorMap =
            new HashMap<Class<?>, Map<ProxyType, Class<? extends
TmInterceptor<?>>>>();

        // Default mapping for objects... Input and Output proxy
instances...
        Map<ProxyType, Class<? extends TmInterceptor<?>>> map =
new HashMap<ProxyType, Class<? extends TmInterceptor<?>>>();
        map.put(ProxyType.INPUT, InputProxyInterceptor.class);
        map.put(ProxyType.OUTPUT, OutputProxyInterceptor.class);
        interceptorMap.put(Object.class, map);

        return interceptorMap;
```

```
    }

    @Bean
    public ObjectProvider objectProvider() {
        return new ObjectProvider(objectFactory(), daoFactory());
    }

    @Bean
    public TmDAOFactory daoFactory() {
        return TmFactoryServer.getTmDAOFActory();
    }

    @Bean
    public TmObjectFactory objectFactory() {
        return TmFactoryServer.getTmObjectFactory();
    }

    @Bean
    public InputProxyMessageConverter inputProxyMessageConverter()
{
        return new InputProxyMessageConverter(proxyProvider());
    }

    @Bean
    public LinkBuilder linkBuilder() {
        Map<Class<?>, String> map = new HashMap<Class<?>,
String>();
        map.put(TmOrderImpl.class, "/orders/");
        return new LinkBuilder(map);
    }
```

```
    }
```

Further Information

The code examples included in this document are simplified in order to convey simple concepts related to implementing RESTful Web services. For the final versions of the code and more detail on the Java Objects I chose to leave out of the explanation, check out the TM feature branch: "`https://athena.redprairie.com/svn/prod/tm/features/tm-purerest`".

If there are any questions regarding the content of this document, feel free to email me at evan.knapp@redprairie.com.

Here and some links for further information of some of the topics discussed in this document:

- For further information on topics or uses of Hibernate check out Hibernate Documents.
- For further information on Spring read Spring's Core Reference Document.
- For further information on Spring MVC read Spring's Spring MVC's Reference Guide.
- For further information on REST read the Wikipedia Article or the IBM DeveloperWorks Article.