Exploring Hypermedia Support in Jersey

Marc Hadley
Oracle
Burlington, MA, USA
marc.hadley@sun.com

Santiago Pericas-Geertsen
Oracle
Palm Beach Grdns, FL, USA
santiago.pericasgeertsen@sun.com

Paul Sandoz
Oracle
Grenoble, France
paul.sandoz@sun.com

ABSTRACT

This paper describes a set of experimental extensions for Jersey[9] that aim to simplify server-side creation and client-side consumption of hypermedia-driven services. We introduce the concept of *action resources* that expose workflow-related operations on a parent resource.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; H.4.3 [**Information Systems Applications**]: Communication Applications; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia

1. INTRODUCTION

The REST architectural style, as defined by Roy Fielding in his thesis [3], is characterized by four constraints: (i) identification of resources (ii) manipulation of resources through representations (iii) self-descriptive messages and (iv) hypermedia as the engine of application state. It is constraint (iv), hypermedia as the engine of application state or HATEOAS for short, that is the least understood and the focus of this paper. HATEOAS refers to the use of *hyperlinks* in resource representations as a way of navigating the state machine of an application.

It is generally understood that, in order to follow the REST style, URIs should be assigned to anything of interest (resources) and a few, well-defined operations (e.g., HTTP operations) should be used to interact with these resources. For example, the state of a purchase order resource can be updated by POSTing (or PATCHing) a new value for its *state* field.[1] However, as has been identified by other authors [5][7], there are *actions* that cannot be easily mapped to read or write operations on resources. These operations are inherently more complex and their details are rarely of interest to clients. For example, given a *purchase order resource*, the operation of setting its state to `REVIEWED` may involve a number of different steps such as (i) checking the customer's credit status (ii) reserving inventory and (iii) verifying per-customer quantity limits. Clearly, this *workflow* cannot be equated to simply updating a field on a resource. Moreover, clients are generally uninterested in the details behind these type of workflows, and in some cases computing the final state of a resource on the client side, as required for a PUT operation, is impractical or impossible.[2]

This paper introduces the concept of *action resources* and explores extensions to Jersey [9] to support them. An action resource is a sub-resource defined for the purpose of exposing workflow-related operations on parent resources. As sub-resources, action resources are identified by URIs that are relative to their parent. For instance, the following are examples of action resources:

```
http://.../orders/1/review
http://.../orders/1/pay
http://.../orders/1/ship
```

for purchase order "1" identified by `http://.../orders/1`.

Action resources provide a simplified hypertext model that can be more easily supported by generic frameworks like Jersey. A set of action resources defines—via their link relationships—a *contract* with clients that has the potential to evolve over time depending on the application's state. For instance, assuming purchase orders are only reviewed once, the `review` action will become unavailable and the `pay` action will become available after an order is reviewed.

The notion of action resources naturally leads to discussions about improved *client* APIs to support them. Given that action resources are identified by URIs, no additional API is really necessary, but the use of client-side proxies and method invocations to trigger these actions seems quite natural [7]. Additionally, the use of client proxies introduces a level of indirection that enables better support for *server evolution*, i.e. the ability of a server's contract to support certain changes without breaking existing clients. Finally, it has been argued [10] that using client proxies is simply more natural for developers and less error prone as fewer

[1]The choice of operation, such as POST, PUT or PATCH, for this type of update is still a matter of debate. See [6].

[2]For instance, when certain parts of the model needed by the workflow are not exposed as resources on the client side.

WS-REST 2010, April 26 2010, Raleigh, NC, USA

URIs need to be constructed.

2. TYPES OF CONTRACTS

A contract established between a server and its clients can be *static* or *dynamic*. In a static contract, knowledge about the server's model is embedded into clients and cannot be updated without re-writing them. In a dynamic contract, clients are capable of discovering knowledge about the contract at runtime and adjust accordingly.

In addition, a dynamic contract can be further subdivided into *contextual* and *non-contextual*. Contextual contracts can be updated in the course of a conversation depending on the application's state; conversely, non-contextual contracts are fixed and independent of the application's state.

HATEOAS is characterized by the use of contextual contracts where the set of actions varies over time. In our purchase ordering system example, this contextual contract will prevent the `ship` action to be carried out before the order is paid, i.e. before the `pay` action is completed.

3. HUMAN VS. BOTS

In order to enable servers to evolve independently, clients and servers should be as decoupled as possible and everything that can change should be learned on the fly. The *Human Web* is based on this type of highly dynamic contracts in which very little is known *a priori*. As very adaptable creatures, humans are able to quickly learn new contracts (e.g. a new login page to access a bank account) and maintain compatibility.

In the *Bot Web*, on the other hand, contracts are necessarily less dynamic and must incorporate some static knowledge as part of the bot's programming: a bot that dynamically learns about some action resources will not be able to *choose* which one to use next if that decision is not part of its programming. It follows that at least a subset of the server's state machine—of which actions are transitions—must be statically known for the bot to accomplish some pre-defined task. However, if action descriptions (including HTTP methods, URI, query parameters, etc.) are mapped at runtime, then they need not be statically known. In fact, several degrees of coupling can be supported as part of the same framework depending on how much information is available statically vs. dynamically.

4. HYPERMEDIA IN JERSEY

Jersey [9] is the reference implementation of the Java API for RESTful Web Services [4]. In this section, we shall describe experimental extensions developed for Jersey to support HATEOAS, including a new client API based on Java dynamic proxies.

These Jersey extensions were influenced by the following (inter-related) requirements:

HATEOAS Support for *actions* and *contextual action sets* as first-class citizens.

Ease of use Annotation-driven model for both client APIs and server APIs. Improved client API based on dynamic generation of Java proxies.

Server Evolution Various degrees of client and server coupling, ranging from static contracts to contextual contracts.

Rather than presenting all these extensions abstractly, we shall illustrate their use via an example. The *Purchase Ordering System* exemplifies a system in which customers can submit orders and where orders are guided by a workflow that includes states like `REVIEWED`, `PAID` and `SHIPPED`.

The system's model is comprised of 4 entities: `Order`, `Product`, `Customer` and `Address`. These model entities are controlled by 3 resource classes: `OrderResource`, `CustomerResource` and `ProductResource`. Addresses are sub-resources that are also controlled by `CustomerResource`. An order instance refers to a single customer, a single address (of that customer) and one or more products. The XML representation (or view) of a sample order is shown below.[3]

```
<order>
    <id>1</id>
    <customer>http://.../customers/21</customer>
    <shippingAddress>
        http://.../customers/21/address/1
    </shippingAddress>
    <orderItems>
        <product>http://.../products/3345</product>
        <quantity>1</quantity>
    </orderItems>
    <status>RECEIVED</status>
</order>
```

Note the use of URIs to refer to each component of an order. This form of *serialization by reference* is supported in Jersey using JAXB beans and the `@XmlJavaTypeAdapter` annotation.[4]

4.1 Server API

The server API introduces 3 new annotation types: `@Action`, `@ContextualActionSet` and `@HypermediaController`. The `@Action` annotation identifies a sub-resource as a *named action*. The `@ContextualActionSet` is used to support contextual contracts and must annotate a method that returns a set of action names. Finally, `@HypermediaController` marks a resource class as a *hypermedia controller class*: a class with one more methods annotated with `@Action` and at most one method annotated with `@ContextualActionSet`.

The following example illustrates the use of all these annotation types to define the `OrderResource` controller.[5]

```
@Path("/orders/{id}")
@HypermediaController(
    model=Order.class,
    linkType=LinkType.LINK_HEADERS)
```

[3]Additional whitespace was added for clarity and space restrictions.

[4]This annotation can be used to customize marshalling and unmarshalling using `XmlAdapter`'s. An `XmlAdapter` is capable of mapping an object reference in the model to a URI.

[5]Several details about this class are omitted for clarity and space restrictions.

```
public class OrderResource {

    private Order order;

    @GET @Produces("application/xml")
    public Order getOrder(
      @PathParam("id") String id) {
        return order;
    }

    @POST @Action("review") @Path("review")
    public void review(
      @HeaderParam("notes") String notes) {
        ...
        order.setStatus(REVIEWED);
    }

    @POST @Action("pay") @Path("pay")
    public void pay(
      @QueryParam("newCardNumber") String newCardNumber) {
        ...
        order.setStatus(PAID);
    }

    @PUT @Action("ship") @Path("ship")
    @Produces("application/xml")
    @Consumes("application/xml")
    public Order ship(Address newShippingAddress) {
        ...
        order.setStatus(SHIPPED);
        return order;
    }

    @POST @Action("cancel") @Path("cancel")
    public void cancel(
      @QueryParam("notes") String notes) {
        ...
        order.setStatus(CANCELED);
    }
}
```

The `@HypermediaController` annotation above indicates that this resource class is a hypermedia controller for the `Order` class. Each method annotated with `@Action` defines a link relationship and associated action resource. These methods are also annotated with `@Path` to make a sub-resource.[6] In addition, `linkType` selects the way in which URIs corresponding to action resources are serialized: in this case, using link headers [11]. These link headers become part of the order's representation. For instance, an order in the `RECEIVED` state, i.e. an order that can only be reviewed or canceled, will be represented as follows.[7]

```
Link: <http://.../orders/1/review>;rel=review;op=POST
Link: <http://.../orders/1/cancel>;rel=cancel;op=POST
<order>
    <id>1</id>
    <customer>http://.../customers/21</customer>
    <shippingAddress>
        http://.../customers/21/address/1
    </shippingAddress>
    <orderItems>
        <product>http://.../products/3345</product>
        <quantity>1</quantity>
    </orderItems>
    <status>RECEIVED</status>
</order>
```

Link headers, rather than links embedded within the entity, were chosen for expediency. A full-featured framework would support both link headers and links embedded within entities but, for the purposes of this investigation, having links only in headers allowed for simpler, media-type independent, link extraction machinery on the client-side.

Without a method annotated with `@ContextualActionSet`, all actions are available at all times regardless of the state of an order. The following method can be provided to define a contextual contract for this resource.

```
@ContextualActionSet
public Set<String> getContextualActionSet() {
    Set<String> result = new HashSet<String>();
    switch (order.getStatus()) {
        case RECEIVED:
            result.add("review");
            result.add("cancel");
            break;
        case REVIEWED:
            result.add("pay");
            result.add("cancel");
            break;
        case PAID:
            result.add("ship");
            break;
        case CANCELED:
        case SHIPPED:
            break;
    }
    return result;
}
```

This method returns a set of action names based on the order's internal state; the values returned in this set correspond to the `@Action` annotations in the controller. For example, this contextual contract prevents shipping an order that has not been paid by only including the `ship` action in the `PAID` state.

Alternate declarative approaches for contextualizing action sets were also investigated but the above approach was chosen for this investigation due to its relative simplicity and expediency.

4.2 Client API

Although action resources can be accessed using traditional APIs for REST, including Jersey's client API [9], the use of client-side proxies and method invocations to trigger these actions seems quite natural. As we shall see, the use of client proxies also introduces a level of indirection that enables better support for server evolution, permitting the definition of contracts with various degrees of coupling.

Client proxies are created based on *hypermedia controller interfaces*. Hypermedia controller interfaces are Java interfaces annotated by `@HypermediaController` that, akin to the server side API, specify the name of a model class and the type of serialization to use for action resource URIs.

[6]There does not appear to be a need to use `@Action` and `@Path` simultaneously, but without the latter some resource methods may become ambiguous. In the future, we hope to eliminate the use of `@Path` when `@Action` is present.

[7]Excluding all other HTTP headers for clarity.

The client-side model class should be based on the representation returned by the server; in particular, in our example the client-side model for an `Order` uses instances of `URI` to link an order to a customer, an address and a list of products.

```
@HypermediaController(
    model=Order.class,
    linkType=LinkType.LINK_HEADERS)
public interface OrderController {

    public Order getModel();

    @Action("review")
    public void review(@Name("notes") String notes);

    @POST @Action("pay")
    public void pay(@QueryParam("newCardNumber")
        String newCardNumber);

    @Action("ship")
    public Order ship(Address newShippingAddress);

    @Action("cancel")
    public void cancel(@Name("notes") String notes);

}
```

The `@Action` annotation associates an interface method with a link relation and hence an action resource on the server. Thus, invoking a method on the generated proxy results in an interaction with the corresponding action resource. The way in which the method invocation is mapped to an HTTP request depends on the additional annotations specified in the interface. For instance, the `pay` action in the example above indicates that it must use a `POST` and that the `String` parameter `newCardNumber` must be passed as a query parameter. This is an example of a *static* contract in which the client has built-in knowledge of the way in which an action is defined by the server. RESTeasy [12] provides a client API that follows this model.

In contrast, the `review` action only provides a name for its parameter using the `@Name` annotation. This is an example of a *dynamic* contract in which the client is only coupled to the `review` link relation and the knowledge that this relation requires `notes` to be supplied. The exact interaction with the `review` action must therefore be discovered dynamically and the `notes` parameter mapped accordingly. The Jersey client runtime uses WADL fragments, dynamically generated by the server, that describe action resources to map these method calls into HTTP requests. The following shows the WADL description of the `review` action resource:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <doc xmlns:jersey="http://jersey.dev.java.net/"
    jersey:generatedBy="Jersey: ..."/>
  <resources base="http://localhost:9998/">
    <resource path="orders/1/review">
      <method name="POST" id="review">
        <request>
          <param type="xs:string" style="header"
            name="notes"/>
        </request>
      </method>
    </resource>
  </resources>
</application>
```

Essentially the WADL is used as a form to configure the request made by the client runtime. The WADL defines the appropriate HTTP method to use (`POST` in this case) and the value of the `@Name` annotation is matched to the corresponding parameter in the WADL to identify where in the request to serialize the value of the `notes` parameter.

The following sample shows how to use `OrderController` to generate a proxy to review, pay and ship an order. For the sake of the example, we assume the customer that submitted the order has been suspended and needs to be activated before the order is reviewed. For that purpose, the client code retrieves the customer URI from the order's model and obtains an instance of `CustomerController`.[8]

```
// Instantiate Jersey's Client class
Client client = new Client();

// Create proxy for order and retrieve model
OrderController orderCtrl = client.proxy(
    "http://.../orders/1", OrderController.class);

// Create proxy for customer in order 1
CustomerController customerCtrl = client.proxy(
    orderCtrl.getModel().getCustomer(),
    CustomerController.class);

// Activate customer in order 1
customerCtrl.activate();

// Review and pay order
orderCtrl.review("approve");
orderCtrl.pay("123456789");

// Ship order
Address newAddress = getNewAddress();
orderCtrl.ship(newAddress);
```

The client runtime will automatically update the action set throughout a conversation: for example, even though the `review` action does not produce a result, the HTTP response to that action still includes a list of link headers defining the contextual action set, which in this case will consist of the `pay` and `cancel` actions but not the `ship` action. An attempt to interact with any action not in the context will result in a client-side exception.

4.3 Server Evolution

Section 4 listed server evolution and the ability to support degrees of coupling as a requirement for our solution. In the last section, we have seen how the use of client proxies based on *partially* annotated interfaces facilitates server evolution.

An interface method annotated with `@Action` and `@Name` represents a *loose* contract with a server. Changes to action resource URIs, HTTP methods and parameter types

[8] `CustomerController` is a hypermedia controller interface akin to `OrderController` which is omitted since it does not highlight any additional feature.

on the server will not require a client re-spin. Naturally, as in all client-server architectures, it is always possible to break backward compatibility, but the ability to support more dynamic contracts—usually at the cost of additional processing time—is still an area of investigation. We see our contribution as a small step in this direction, showing the potential of using dynamic meta-data (WADL in our case) for the definition of these type of contracts.

5. RELATED WORK

In this section we provide a short overview of other REST frameworks that inspired our work. RESTeasy [12] provides a framework that supports client proxies generated from annotated interfaces. RESTfulie [8] is, to the best of our knowledge, the first public framework with built-in support for hypermedia.

5.1 RESTeasy Client Framework

The RESTeasy Client Framework follows a similar approach in the use of client-side annotations on Java interfaces for the creation of dynamic proxies. It differs from our approach in that it neither supports hypermedia nor the ability to map proxy method calls using dynamic information. That is, in the RESTeasy Client Framework, proxy method calls are mapped to HTTP requests exclusively using static annotations. These client-side annotations establish a tight coupling that makes server evolution difficult.

Despite these shortcomings, we believe their approach is a good match for certain types of applications, especially those in which servers and clients are controlled by the same organization. In addition, the use of client proxies simplifies programming and improves developer productivity. For these reasons, we have followed a similar programming model while at the same time provided support for hypermedia and dynamic contracts.

5.2 RESTfulie Hypermedia Support

This article [7] by Guillerme S. describes how to implement hypermedia aware resources using RESTfulie. In RESTfulie, action URIs are made part of a resource representation (more specifically, the entity) via the use of Atom links [14]. Even though we foresee supporting other forms of URI serialization (as indicated by the use of `linkType` in `@HypermediaController`), we believe link headers to be the least intrusive and most likely to be adopted when migrating existing applications into hypermedia-aware ones.[9]

Rather than providing an explicit binding between actions and HTTP methods, RESTfulie provides a pre-defined table that maps `rel` elements to HTTP methods. For example, an `update` action is mapped to `PUT` and a `destroy` action is mapped to `DELETE`. We believe this implicit mapping to be unnecessarily confusing and not easily extensible. Instead, as we have done in this paper, we prefer to make *action resources* first class and provide developers tools to define explicit mappings via the use of `@Action` annotations.

In RESTfulie, knowledge about action resources is discovered dynamically and, as a result, Java reflection is the only mechanism available to interact with hypermedia-aware resources.[10] So instead of writing `order.cancel()`, a developer needs to write:

```
resource(order).getTransition("pay").execute()
```

As explained in Section 3, this decision is impractical given that certain static knowledge is required in order to program bots. We believe that Java interfaces annotated with `@Action` and `@Name` provide the right amount of static information to enable bot programming and server evolution whilst not sacrificing the ease of use offered by client proxies.

6. CONCLUSIONS

In this paper we have introduced the notion of an action resource and, with it, described some experimental extensions for Jersey to support HATEOAS. Our analysis lead us into the exploration of innovative client APIs that enable developers to define client-server contracts with different degrees of coupling and improve the ability of servers to evolve independently. In the process, we also argued that there is a minimum amount of static information that is needed to enable programmable clients (bots) and showed how that information can be captured using client interfaces.

All the source code shown in the previous sections is part of a complete and runnable sample [15] available in the Jersey subversion repository. The solution proposed herein is experimental and is likely to evolve in unforeseen directions once developers start exploring HATEOAS in real-world systems.

We are currently evaluating support for entity-based action resources URIs. Support for Atom link elements (as has been proposed by other authors) is a likely avenue for exploration since their use, like that of Link headers, permits support of many different XML-based media types without requiring media-type specific machinery. Different ways in which contextual action sets are defined are being explored, as well as simplifications to the client APIs for the instantiation of dynamic proxies, especially those created from other proxies.

The authors would like to thank Martin Matula for his suggestions on how to improve the sample and Gerard Davison for reviewing our work and providing insightful comments.

7. REFERENCES

[1] B. Burke. *RESTful Java with JAX-RS*. O'Reilly, 2009.
[2] R. Chinnici and B. Shannon. Java Platform, Enterprise Edition (JavaEE) Specification, v6. JSR, JCP, November 2009. See http://jcp.org/en/jsr/detail?id=316.
[3] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Ph.d dissertation, University of California, Irvine, 2000. See http://roy.gbiv.com/pubs/dissertation/top.htm.

[9]For example, existing applications that use XML schema validation on the entities.

[10]The exact manner in which the RESTfulie runtime maps parameters in an HTTP request is unclear to us at the time of writing.

[4] M. Hadley and P. Sandoz. JAXRS: Java API for RESTful Web Services. JSR, JCP, September 2009. See http://jcp.org/en/jsr/detail?id=311.

[5] T. Bray. RESTful Casuistry. Blog, March 2009. See http://www.tbray.org/ongoing/When/200x/2009/03/20/Rest-Casuistry.

[6] R. Fielding. It is okay to use POST. Blog, March 2009. See http://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post.

[7] G. Silveira. Quit pretending, use the web for real: restfulie. Blog, November 2009. See http://guilhermesilveira.wordpress.com/2009/11/03/quit-pretending-use-the-web-for-real-restfulie.

[8] RESTfulie. See http://freshmeat.net/projects/restfulie.

[9] JAX-RS reference implementation for building RESTful web services. See https://jersey.dev.java.net.

[10] Why HATEOAS? Blog, April 2009. See http://blogs.sun.com/craigmcc/entry/why_hateoas.

[11] Web Linking (draft). Mark Nottingham. http://tools.ietf.org/html/draft-nottingham-http-link-header-06.

[12] RESTeasy Client Framework. See http://www.jboss.org/file-access/default/members/resteasy/freezone/docs/1.2.GA/userguide/html_single/index.html#RESTEasy_Client_Framework.

[13] Web Application Description Language (WADL). Marc Hadley. See https://wadl.dev.java.net.

[14] Atom Syndication Format. See http://www.w3.org/2005/Atom.

[15] Jersey Hypermedia Sample. See http://tinyurl.com/jersey-hypermedia.