JAX-RS: Java™ API for RESTful Web Services

Editors Draft
October 3, 2007

Editors:
Marc Hadley
Paul Sandoz

Comments to: users@jsr311.dev.java.net

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 USA

Specification: JAX-RS - Java™ API for RESTful Web Services (“Specification”)
Version: 1.0-ED1
Status: Pre-FCS Public Release
Release: October 2, 2007

NOTICE

TRADEMARKS

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED ”AS IS” AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government’s rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification (“Feedback”). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties’ entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Contents

Chapter 1

Introduction

This specification defines a set of Java APIs for the development of Web services built according to the Representational State Transfer[1] (REST) architectural style. Readers are assumed to be familiar with REST; for more information about the REST architectural style and RESTful Web services, see:

- Architectural Styles and the Design of Network-based Software Architectures[1]
- The REST Wiki[2]
- Representational State Transfer on Wikipedia[3]

1.1 Status

This is an early draft review; this specification is not yet complete. A list of open issues can be found at:

https://jsr311.dev.java.net/servlets/ProjectIssues

Javadocs can be found online at:

https://jsr311.dev.java.net/nonav/releases/0.3/index.html

The reference implementation can be obtained at:

https://jersey.dev.java.net/

The expert group seeks feedback from the community on any aspect of this specification, please send comments to:

users@jsr311.dev.java.net

1.2 Goals

The following are the goals of the API:

POJO-based The API will provide a set of annotations and associated classes/interfaces that may be used with POJOs in order to expose them as Web resources. The specification will define object lifecycle and scope.

HTTP-centric The specification will assume HTTP[4] is the underlying network protocol and will provide a clear mapping between HTTP and URI[5] elements and the corresponding API classes and annotations. The API will provide high level support for common HTTP usage patterns and will be sufficiently flexible to support a variety of HTTP applications including WebDAV[6] and the Atom Publishing Protocol[7].

Format independence The API will be applicable to a wide variety of HTTP entity body content types. It will provide the necessary pluggability to allow additional types to be added by an application in a standard manner.

Container independence Artifacts using the API will be deployable in a variety of Web-tier containers. The specification will define how artifacts are deployed in a Servlet[8] container and as a JAX-WS[9] Provider.

Inclusion in Java EE The specification will define the environment for a Web resource class hosted in a Java EE container and will specify how to use Java EE features and components within a Web resource class.

1.3 Non-Goals

The following are non-goals:

Support for Java versions prior to J2SE 5.0 The API will make extensive use of annotations and will require J2SE 5.0 or later.

Description, registration and discovery The specification will neither define nor require any service description, registration or discovery capability.

Client APIs The specification will not define client-side APIs. Other specifications are expected to provide such functionality.

HTTP Stack The specification will not define a new HTTP stack. HTTP protocol support is provided by a container that hosts artifacts developed using the API.

Data model/format classes The API will not define classes that support manipulation of entity body content, rather it will provide pluggability to allow such classes to be used by artifacts developed using the API.

1.4 Conventions

The keywords ‘MUST’, ‘MUST NOT’, ‘REQUIRED’, ‘SHALL’, ‘SHALL NOT’, ‘SHOULD’, ‘SHOULD NOT’, ‘RECOMMENDED’, ‘MAY’, and ‘OPTIONAL’ in this document are to be interpreted as described in RFC 2119[10].

Java code and sample data fragments are formatted as shown in figure 1.1:

Figure 1.1: Example Java Code

```
1  package com.example.hello;
2
3  public class Hello {
4      public static void main(String args[]) {
5          System.out.println("Hello World");
6      }
7  }
```

URIs of the general form ‘http://example.org/...’ and ‘http://example.com/...’ represent application or context-dependent URIs.

All parts of this specification are normative, with the exception of examples, notes and sections explicitly marked as ‘Non-Normative’. Non-normative notes are formatted as shown below.

Note: *This is a note.*

1.5 Terminology

Resource class A Java class that uses JAX-RS annotations to implement a corresponding Web resource, see chapter 2.

Root resource class A resource class annotated with `@UriTemplate`. Root resource classes provide the roots of the resource class tree and provide access to sub-resources, see chapter 2.

Resource method A method of a resource class that is used to handle requests on the corresponding resource, see section 2.4.

Sub-resource locator A method of a resource class that is used to locate sub-resources of the corresponding resource, see section 2.1.1.

Sub-resource method A method of a resource class that is used to handle requests on a sub-resource of the corresponding resource, see section 2.1.1.

1.6 Expert Group Members

This specification is being developed as part of JSR 311 under the Java Community Process. This specification is the result of the collaborative work of the members of the JSR 311 Expert Group. The following are the present and former expert group members:

Jan Algermissen (Individual Member)
Heiko Braun (Red Hat Middleware LLC)

Larry Cable (BEA Systems)
Bill De Hora (Individual Member)
Roy Fielding (Day Software, Inc.)
Harpreet Geekee (Nortel)
Nickolas Grabovas (Individual Member)
Mark Hansen (Individual Member)
John Harby (Individual Member)
Hao He (Individual Member)
Ryan Heaton (Individual Member)
David Hensley (Individual Member)
Changshin Lee (NCsoft Corporation)
Francois Leygues (Alcatel-Lucent)
Jerome Louvel (Individual Member)
Hamid Ben Malek (Fujitsu Limited)
Ryan J. McDonough (Individual Member)
Felix Meschberger (Day Software, Inc.)
David Orchard (BEA Systems)
Dhanji R. Prasanna (Individual Member)
Julian Reschke (Individual Member)
Jan Schulz-Hofen (Individual Member)
Joel Smith (IBM)
Stefan Tilkov (innoQ Deutschland GmbH)

1.7 Acknowledgements

Editors Note 1.1 *TBD.*

Chapter 2

Resource Classes

Using JAX-RS a Web resource is implemented as a resource class; this section describes resource classes in detail.

2.1 URI Templates

A resource class is anchored in URI space using the `@UriTemplate` annotation. The value of the annotation is a relative URI template with the base URI being provided by the deployment context. Root resource classes are anchored directly using a `@UriTemplate` annotation on the class.

A URI template is a string with zero or more embedded parameters that, when values are substituted for all the parameters, conforms to the URI[5] production. A parameter is represented as '{'*name*'}' where *name* is the name of the parameter. E.g.:

Editors Note 2.1 *Add reference to URI Templates ID when available.*

```
1  @UriTemplate("widgets/{id}")
2  public class Widget {
3    ...
4  }
```

In the above example the `Widget` resource class is identified by the relative URI `widgets/`*`xxx`* where *`xxx`* is the value of the `id` parameter.

The `encode` property controls whether the value of the `@UriTemplate` annotation is automatically encoded (the default) or not. When automatic encoding is disabled, care must be taken to ensure that the value of the URI template is valid. E.g. the following two lines are equivalent:

```
1  @UriTemplate("widget list/{id}")
2  @UriTemplate(value="widget%20list/{id}" encode=false)
```

The `limited` property controls whether a trailing template variable matches a single path segment or multiple. Setting the property to false allows a single template variable to match a path and can be used, e.g., when a template represents a path prefix followed by an arbitrary length path.

2.1.1 Sub Resources

Resource class methods can also be annotated with `@UriTemplate`. The effect of the annotation depends on whether the method is also annotated with `@HttpMethod` or not:

Not annotated with `@HttpMethod` Such methods, known as sub-resource locators, are used to further resolve the object that will handle the request. Any returned object is treated as a resource class and used to either handle the request or to further resolve the object that will handle the request, see 2.6 for further details.

Annotated with `@HttpMethod` Such methods, known as sub-resource methods, are treated like a normal resource method (see section 2.4) except the method is only invoked for request URIs that match a URI template created by concatenating the URI template of the resource class with the URI template of the method[1].

The following example illustrates the difference:

```
1  @UriTemplate("widgets")
2  public class WidgetList {
3    @HttpMethod
4    @UriTemplate("offers")
5    WidgetList getDiscounted() {...}
6
7    @UriTemplate("{id}")
8    Widget findWidget(@UriParam("id") String id) {
9      return lookupWidget(id);
10   }
11 }
```

In the above a `GET` request for the `widgets/offers` resource is handled directly by the `getDiscounted` sub-resource method of `WidgetList` whereas a `GET` request for `widgets/`*`xxx`* is handled by whatever object is returned by the `findWidget` sub-resource locator.

Note that a set of sub-resource methods annotated with the same URI template value are functionally equivalent to a similarly annotated sub-resource locator that returns an instance of a resource class with the same set of resource methods.

2.2 Lifecycle

A new resource class instance is created for each request to that resource. First the constructor (see section 2.3) is called, then the appropriate method (see section 2.4) is invoked and finally the object is made available for garbage collection.

2.3 Constructors

Root resource classes are instantiated by the JAX-RS runtime and MUST have a constructor with one of the following annotations on every parameter: `@HttpContext`, `@HeaderParam`, `@MatrixParam`, `@QueryParam` or `@UriParam`. Note that a zero argument constructor is permissible under this rule. Section 2.4.1

[1] If the resource class URI template does not end with a '/' character then one is added during the concatenation.

defines the parameter types permitted for each annotation. If more than one constructor that matches the above pattern is available then an implementation MUST use the one with the most parameters. Choosing amongst constructors with the same number of parameters is implementation specific.

Non-root resource classes are instantiated by an application and do not require the above-described constructor.

2.4 Resource Methods

Resource methods are resource class methods annotated with `@HttpMethod`. They are used to handle requests and MUST conform to certain restrictions described in this section.

The `@HttpMethod` annotation has an optional value that corresponds to the name of a request method. In the absence of a value, the request method is inferred from the resource method name: they match if the resource method name starts with the request method name converted to lower case.

2.4.1 Parameters

When the method is invoked, annotated parameter values are mapped from the request according to the semantics of the annotation. The following describes the permitted types for an annotated parameter.

`@MatrixParam`, `@QueryParam` or `@UriParam` The class of the annotated parameter MUST have a constructor that accepts a single `String` argument, or a static method named `valueOf` that accepts a single `String` argument. By default, parameter values are automatically decoded; automatic decoding can be disabled using the `@Encoded` annotation.

`@HttpContext` The class of the annotated parameter MUST be `UriInfo`, `PreconditionEvaluator` or `HttpHeaders`. See chapter 4 for additional information on these types.

`@HeaderParam` The class of the annotated parameter MUST have a constructor that accepts a single `String` argument, or a static method named `valueOf` that accepts a single `String` argument. Other types may be supported using a `HeaderProvider` as described in section 3.2.

The value of an non-annotated parameter is mapped from the request entity body. Resource methods MUST NOT have more than one parameter that is not annotated with one of `@HttpContext`, `@HeaderParam`, `@MatrixParam`, `@QueryParam` or `@UriParam`. Conversion between an entity body and a Java type is the responsibility of an `EntityProvider`, see section 3.1.

2.4.2 Return Type

Resource methods MAY return `void`, `Response` or another Java type, these return types are mapped to a response entity body as follows:

`void` Results in an empty entity body.

`instanceof Response` Results in an entity body mapped from the `Entity` property of the `Response`.

Other Results in an entity body mapped from the return type.

Conversion between a Java types and an entity body is the responsibility of an `EntityProvider`, see section 3.1.

Methods that need to provide additional metadata with a response should return an instance of `Response`, the `Response.Builder` class provides a convenient way to create a `Response` instance using a builder pattern.

2.4.3 Exceptions

An implementation MUST catch `WebApplicationException` and map it to a response. If the `response` property of the exception is not `null` then it MUST be used to create the response. If the `response` property of the exception is `null` an implementation MUST generate a server error response.

An implementation MUST allow other runtime exceptions to propagate to the underlying container. This allows existing container facilities (e.g. a Servlet filter) to be used to handle the error if desired.

Editors Note 2.2 *What to do about checked exceptions ? If we allow them on resource methods then do we need some standard runtime exception that can be used to wrap the checked exception so it can be propagated to the container in a standard way ?*

2.4.4 HEAD and OPTIONS

`HEAD` and `OPTIONS` requests receive additional support. On receipt of `HEAD` request an implementation MUST either:

1. Call a method annotated with `@HttpMethod` that supports `HEAD` or, if none present,
2. Call a method annotated with `@HttpMethod` that supports `GET` and discard any returned entity.

Note that option 2 may result in reduced performance where entity creation is significant.

On receipt of an `OPTIONS` request an implementation MUST either:

1. Call a method annotated with `@HttpMethod` that supports `OPTIONS` or, if none present,
2. Generate an automatic response from the declared metadata of the matching class.

2.5 Declaring Media Type Capabilities

Application classes can declare the supported request and response media types using the `@ProduceMime` and `@ConsumeMime` annotations. These annotations MAY be applied to a resource class method, a resource class, or to an `EntityProvider` (see section 3.1.1). Declarations on a resource class method override any on the resource class; declarations on an `EntityProvider` for a method argument or return type override those on a resource class or resource method. In the absence of either of these annotations, support for any media type ("*/*") is assumed.

The following example illustrates the `@ProduceMime` annotation:

```
@UriTemplate("widgets")
@ProduceMime("application/widgets+xml")
public class WidgetList {

  @HttpMethod
  String getAll() {...}

  @HttpMethod
  @UriTemplate("{id}")
  Widget getWidget(@UriParam("id") String id) {...}

  @HttpMethod
  @UriTemplate("{id}/description")
  @ProduceMime("text/html")
  String getDescription(@UriParam("id") String id) {...}
}

@Provider
@ProduceMime({"application/widgets+xml", "application/json"})
public class WidgetProvider implements EntityProvider<Widget> {...}
```

In the above, the `getAll` resource method returns a `String` in the `application/widgets+xml` format, the `getDescription` sub-resource method returns a `String` as `text/html` and the `getWidget` sub-resource method return a `Widget` instance that can be mapped to either `application/widgets+xml` or `application/json` using the `WidgetProvider` class (see section 3.1 for more information on `EntityProvider`).

An implementation MUST NOT invoke a method whose effective value of `@ProduceMime` does not match the request `Accept` header. An implementation MUST NOT invoke a method whose effective value of `@ConsumeMime` does not match the request `Content-Type` header.

2.6 Matching Requests to Resource Methods

Matching of requests to resource methods proceeds in two stages:

1. Obtain the object that will handle the request.
   (a) Set `uri` to the request URI
   (b) For each resource class compute a regular expression from its URI template using the process described in section 2.6.1. If the resource class has sub-resources (see section 2.1.1) then append '(/.*)?' to the resulting regular expression, if not then append '(/)?'.
   (c) Filter the set of resource classes by rejecting those whose regular expression does not match `uri`. If the set is empty then no matching resource can be found, the algorithm terminates and an implementation MUST generate a not found response (HTTP 404 status).
   (d) Sort the set of matching resource classes using the number of characters in the regular expression not resulting from template variables as the primary key and the number of matching groups as a secondary key.
   (e) Select the first matching class, instantiate an object of that class and set `uri` to the value of the final matching group.

(f) If `uri` is empty or is '/' go to step 2.

(g) For each of the object's sub-resource methods (see section 2.1.1) compute a regular expression for the URI template using the process described in section 2.6.1 and then appending '(/)?'. If `uri` matches any of the regular expressions go to step 2.

(h) For each of the object's sub-resource locators (see section 2.1.1) compute a regular expression for the URI template using the process described in section 2.6.1 and then appending '(/.*)?'.

(i) Filter the set of sub-resource locators by rejecting those whose regular expression does not match `uri`. If the set is empty then no matching resource can be found, the algorithm terminates and an implementation MUST generate a not found response (HTTP 404 status).

(j) Sort the set of matching sub-resource locators using the number of matching groups as the primary key and the number of characters in the regular expression as a secondary key.

(k) Set `uri` to the value of the final matching group and invoke the first matching method to obtain the next matching resource object. Repeat from step 1f using the new object.

2. Identify the method that will handle the request.

(a) Find the set of resource methods that meet the following criteria:

- If `uri` is not empty or equal to '/', the method must be annotated with a URI template that, when transformed into a regular expression using the process described in section 2.6.1, matches `uri`.
- The request method is supported. If no methods support the request method an implementation MUST generate a method not allowed response (HTTP 405 status). Note the additional support for `HEAD` and `OPTIONS` described in section 2.4.4.
- The media type of the request entity body (if any) is a supported input data format (see section 2.5). If no methods support the media type of the request entity body an implementation MUST generate an unsupported media type response (HTTP 415 status).
- At least one of the acceptable response entity body media types is a supported output data format (see section 2.5). If no methods support one of the acceptable response entity body media types an implementation MUST generate a not acceptable response (HTTP 406 status).

(b) Sort the matching set of resource methods using the media type of input data as the primary key and the media type of output data as the secondary key.

Sorting of media types follows the general rule: x/y $<$ x/* $<$ */*, i.e. a method that explicitly lists one of the requested media types is sorted before a method that lists */*. Quality parameter values are also used such that x/y;q=1.0 $<$ x/y;q=0.7.

(c) If the set of matching resource methods is non-empty then the request is dispatched to the first Java method in the set; otherwise no matching resource method can be found and the algorithm terminates.

2.6.1 Converting URI Templates to Regular Expressions

A URI template is converted into a regular expression by:

1. If required, i.e. `@UriTemplate.encode=true`, URI encoding any invalid characters in the template, ignoring URI template variable specifications.

2. Escaping any regular expression characters in the URI template, again ignoring URI template variable specifications.
3. Substituting '(.*?)' for each occurrence of a URI template variable ({\([w-\._]+?\)}) within the URI template.

Note that the above renders the name of template variables irrelevant for template matching purposes. However, implementations will need to retain template variable names in order to facilitate the extraction of template variable values via `@UriParam` or `UriInfo.getURIParameters`.

Chapter 3

Contracts and Providers

The JAX-RS runtime is extended using application-supplied provider classes. A provider is annotated with `@Provider` and implements a contract defined by an interface annotated with `@Contract`. JAX-RSdefines two contracts: `EntityProvider` and `HeaderProvider` these are described below.

3.1 Entity Providers

The `EntityProvider` interface defines the contract between the JAX-RS runtime and components that provide mapping services between Java types and their associated representations. A class wishing to provide such a service implements the `EntityProvider` interface and is annotated with `@Provider`.

3.1.1 Declaring Media Type Capabilities

An `EntityProvider` MAY restrict the media types it supports using the `@ProduceMime` and `@ConsumeMime` annotations. In the absence of either of these annotations, support for any media type ("*/*") is assumed. An implementation MUST NOT use an `EntityProvider` to map from a representation whose media type is not declared in a `@ConsumeMime` annotation nor to a representation whose media type is not declared in a `@ProduceMime` annotation.

When choosing an `EntityProvider` an implementation sorts the available providers according to the media types they declare support for. Sorting of media types follows the general rule: x/y $<$ x/* $<$ */*, i.e. a provider that explicitly lists a media types is sorted before a provider that lists */*. Quality parameter values are also used such that x/y;q=1.0 $<$ x/y;q=0.7.

3.1.2 Standard Entity Providers

An implementation MUST include pre-packaged `EntityProvider` implementations for the following Java and media type combinations:

`byte[]` All media types (`*/*`).

`java.lang.String` All text media types (`text/*`).

`java.io.InputStream` All media types (`*/*`).

`java.io.File` All media types (`*/*`).

`javax.activation.DataSource` All media types (`*/*`).

`javax.transform.Source` XML types (`text/xml`, `application/xml` and `application/*+xml`).

`javax.xml.bind.JAXBElement` and application-supplied JAXB classes XML media types (`text/xml`, `application/xml` and `application/*+xml`).

`MultivaluedMap<String,String>` Form content (`application/x-www-form-urlencoded`).

An implementation MUST support application-provided `EntityProvider` implementations and MUST use those in preference to its own pre-packaged `EntityProvider` implementations when either could handle the same request.

3.1.3 Consuming a Request Entity Body

The following describes the logical[1] steps taken by a JAX-RS implementation when mapping a request entity body to a Java method parameter:

1. Identify the Java type of the parameter whose value will be mapped from the entity body. Section 2.6 describes how the Java method is chosen.
2. Select the set of `EntityProvider` classes that support the media type of the request, see section 3.1.1.
3. Iterate through the selected `EntityProvider` classes and, utilizing the `supports` method of each, choose an `EntityProvider` that supports the desired Java type.
4. Use the `readFrom` method of the chosen `EntityProvider` to map the entity body to the desired Java type.

3.1.4 Producing a Response Entity Body

The following describes the logical steps taken by a JAX-RS implementation when mapping a return value to a response entity body:

1. Obtain the object that will be mapped to the response entity body. For a return type of `Response` or subclasses the object is the value of the `Entity` property, for other return types it is the returned object.
2. Obtain the effective value of `@ProduceMime` (see section 2.5) and intersect that with the requested response formats to obtain set of permissible media types for the response entity body. Note that section 2.6 ensures that this set will not be empty.
3. Select the set of `EntityProvider` classes that support (see section 3.1.1) one or more of the permissible media types for the response entity body.
4. Sort the selected `EntityProvider` classes as described in section 3.1.1.

[1] Implementations are free to optimize their processing provided the results are equivalent to those that would be obtained if these steps are followed.

5. Iterate through the sorted `EntityProvider` classes and, utilizing the `supports` method of each, choose an `EntityProvider` that supports the object that will be mapped to the entity body.
6. Use the `writeTo` method of the chosen `EntityProvider` to map the object to the entity body.

3.1.5 Transfer Encoding

Transfer encodings are handled by a component of the container or the JAX-RS runtime. Entity providers and application methods always operate on the HTTP entity body rather than directly on the HTTP message body.

3.2 Header Providers

The `HeaderProvider` interface defines the contract between the JAX-RS runtime and components that provide mapping services between Java types and their associated headers. A class wishing to provide such a service implements the `HeaderProvider` interface and is annotated with `@Provider`.

Both `Response.Builder` and `EntityProvider` allow arbitrary objects to be set as the value of headers, an implementation MUST first try to use a `HeaderProvider` for the class of the value object and then, if none exists, use the object's `toString` method to serialize the value instead.

3.2.1 Standard Header Providers

An implementation MUST include pre-packaged `HeaderProvider` implementations for the following JAX-RS types: `MediaType`, `EntityTag` and `NewCookie`.

Chapter 4

Context

JAX-RS provides facilities for obtaining and processing information about the deployment context of a resource class and the context of individual requests. This chapter describes these facilities.

4.1 URIs and URI Templates

An instance of `UriInfo` can be injected into a class field or method parameter using the `@HttpContext` annotation. `UriInfo` provides both static and dynamic, per-request information about the components of a request URI. E.g. the following would return the names of any query parameters in a request:

```
1  @HttpMethod(GET)
2  @ProduceMime{"text/plain"}
3  public String listQueryParamNames(@HttpContext UriInfo info) {
4    StringBuilder buf = new StringBuilder();
5    for (String param: info.getQueryParameters().keySet()) {
6      buf.append(param);
7      buf.append("\n");
8    }
9    return buf.toString();
10 }
```

4.2 Headers

An instance of `HttpHeaders` can be injected into a class field or method parameter using the `@HttpContext` annotation. `HttpHeaders` provides access to request header information either in map form or via strongly typed convenience methods. E.g. the following would return the names of all the headers in a request:

```
1  @HttpMethod(GET)
2  @ProduceMime{"text/plain"}
3  public String listHeaderNames(@HttpContext HttpHeaders headers) {
4    StringBuilder buf = new StringBuilder();
5    for (String header: headers.getRequestHeaders().keySet()) {
6      buf.append(header);
7      buf.append("\n");
```

```
    }
    return buf.toString();
}
```

Note that response headers may be provided using the `Response` interface, see 2.4.2 for more details.

4.3 Preconditions

JAX-RS simplifies support for preconditions using the `PreconditionEvaluator` interface. An instance of `PreconditionEvaluator` can be injected into a class field or method parameter using the `@HttpContext` annotation. The methods of `PreconditionEvaluator` allow a resource method to evaluate whether the current state of the resource matches any preconditions in the request and if not they return a `Response` that can be returned to the client to inform it that the request preconditions were not met. E.g. the following checks that the current entity tag matches any preconditions in the request before updating the resource:

```
@HttpMethod(PUT)
public Response updateFoo(@HttpContext PreconditionEvaluator pre, Foo foo) {
    EntityTag tag = getCurrentTag();
    Response response = pre.evaluate(tag);
    if (response != null)
      return response;
    else
      return doUpdate(foo);
}
```

4.4 Injection Scope

When the `@HttpContext` annotation is applied to a resource class field, an implementation is only required to inject the applicable context into those resource class instances created by the implementation runtime. Objects returned by sub-resource locators are expected to be initialized by their creator and are not subject to resource injection by the implementation runtime.

Chapter 5

Environment

The container-managed resources available to a JAX-RS resource class depend on the environment in which the JAX-RS resource class is deployed. As described in chapter 4, all resource classes can access the `UriInfo`, `HttpHeaders` and `PreconditionEvaluator` contexts regardless of container. The following sections describe the additional container-managed resources available to a JAX-RS resource class deployed in a variety of environments.

5.1 Servlet Container

The `javax.annotation.Resource` annotation can be used to indicate a dependency on a Servlet-defined resource. An implementation MUST support injection of the following types: `ServletConfig`, `ServletContext`, `HttpServletRequest` and `HttpServletResponse`.

An injected `HttpServletRequest` allows a resource class method to stream the contents of a request entity. If the resource class method has a parameter whose value is derived from the request entity then the stream will have already been consumed and an attempt to access it MAY result in an exception.

An injected `HttpServletResponse` allows a resource class method to commit the HTTP response prior to returning. An implementation MUST check the committed status and only process the return value if the response is not yet committed.

5.2 Java EE Container

Editors Note 5.1 *TBD. We anticipate offering the same resource injection capabilities as are provided for a Servlet instance running in a Java EE Web container. In particular we anticipate supporting dependency injection using the following annotations: @Resource, @Resources, @EJB, @EJBs, @WebServiceRef, @WebServiceRefs, @PersistenceContext, @PersistenceContexts, @PersistenceUnit and @PersistenceUnits. We also anticipate supporting the following JSR 250 lifecycle management and security annotations: @PostConstruct, @PreDestroy, @RunAs, @RolesAllowed, @PermitAll, @DenyAll and @DeclareRoles.*

5.3 Other

Other container technologies MAY specify their own set of injectable resources but MUST, at a minimum, support access to the `UriInfo`, `HttpHeaders` and `PreconditionEvaluator` as described in chapter 4.

Bibliography

[1] 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.

[2] REST Wiki. Web site. See http://rest.blueoxen.net/cgi-bin/wiki.pl.

[3] Representational State Transfer. Web site, Wikipedia. See http://en.wikipedia.org/wiki/Representational_State_Transfer.

[4] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. RFC, IETF, January 1997. See http://www.ietf.org/rfc/rfc2616.txt.

[5] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. RFC, IETF, January 2005. See http://www.ietf.org/rfc/rfc3986.txt.

[6] L. Dusseault. RFC 4918: HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC, IETF, June 2007. See http://www.ietf.org/rfc/rfc4918.txt.

[7] J.C. Gregorio and B. de hOra. The Atom Publishing Protocol. Internet Draft, IETF, March 2007. See http://bitworking.org/projects/atom/draft-ietf-atompub-protocol-14.html.

[8] G. Murray. Java Servlet Specification Version 2.5. JSR, JCP, October 2006. See http://java.sun.com/products/servlet.

[9] R. Chinnici, M. Hadley, and R. Mordani. Java API for XML Web Services. JSR, JCP, August 2005. See http://jcp.org/en/jsr/detail?id=224.

[10] S. Bradner. RFC 2119: Keywords for use in RFCs to Indicate Requirement Levels. RFC, IETF, March 1997. See http://www.ietf.org/rfc/rfc2119.txt.