JAX-RS: Java™ API for RESTful Web Services

Editors Draft
August 27, 2008

Editors:
Marc Hadley
Paul Sandoz

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

Sun Microsystems, Inc.
4150 Network Circle, Santa Clara, CA 95054 USA.
180, Avenue de L'Europe, 38330 Montbonnot Saint Martin, France

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 editors draft; this specification is not yet complete. A list of open issues can be found at:

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

The latest Javadocs can be found online at:

https://jsr311.dev.java.net/nonav/javadoc/index.html

The reference implementation can be obtained from:

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

```
package com.example.hello;

public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

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 3.

Root resource class A *resource class* annotated with `@Path`. Root resource classes provide the roots of the resource class tree and provide access to sub-resources, see chapter 3.

Request method designator A runtime annotation annotated with `@HttpMethod`. Used to identify the HTTP request method to be handled by a *resource method*.

Resource method A method of a *resource class* annotated with a *request method designator* that is used to handle requests on the corresponding resource, see section 3.3.

Sub-resource locator A method of a *resource class* that is used to locate sub-resources of the corresponding resource, see section 3.4.1.

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 3.4.1.

Provider An implementation of a JAX-RS extension interface. Providers extend the capabilities of a JAX-RS runtime and are described in chapter 4.

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)
Stephan Koops (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

During the course of the JSR we received many excellent suggestions on the JSR and Jersey (RI) mailing lists, thanks in particular to Bill Burke (Red Hat) and James Manger (Telstra) for their contributions.

The `GenericEntity` class was inspired by the Google Guice `TypeLiteral` class. Our thanks to Bob Lee and Google for donating this class to JAX-RS.

The following individuals (all Sun Microsystems) have also made invaluable technical contributions: Roberto Chinnici, Dianne Jiao (TCK), Ron Monzillo, Rajiv Mordani, Eduardo Pelegri-Llopart, Jakub Podlesak (RI) and Bill Shannon.

Chapter 2

Applications

A JAX-RS application consists of one or more resources (see chapter 3) and zero or more providers (see chapter 4). This chapter describes aspects of JAX-RS that apply to an application as a whole, subsequent chapters describe particular aspects of a JAX-RS application and requirements on JAX-RS implementations.

2.1 Configuration

The resources and providers that make up a JAX-RS application are configured via an application-supplied subclass of `Application`. An implementation MAY provide alternate mechanisms for locating resource classes and providers (e.g. runtime class scanning) but use of `Application` is the only portable means of configuration.

2.2 Validation

Specific validation requirements are detailed throughout this specification and the JAX-RS Javadocs. Implementations MAY perform additional validation where feasible and SHOULD report any issues arising from such validation to the user.

2.3 Publication

Applications are published in different ways depending on whether the application is run in a Java SE environment or within a container. This section describes the alternate means of publication.

2.3.1 Java SE

In a Java SE environment a configured instance of an endpoint class can be obtained using the `createEndpoint` method of `RuntimeDelegate`. The application supplies an instance of `Application` and the type of endpoint required. An implementation MAY support zero or more endpoint types of any desired type.

How the resulting endpoint class instance is used to publish the application is outside the scope of this specification.

2.3.1.1 JAX-WS

An implementation that supports publication via JAX-WS MUST support `createEndpoint` with an endpoint type of `javax.xml.ws.Provider`. JAX-WS describes how a `Provider` based endpoint can be published in an SE environment.

2.3.2 Servlet

A JAX-RS application is packaged as a Servlet in a `.war` file. The `Application` subclass (see section 2.1), resource classes, and providers are packaged in `WEB-INF/classes`, required libraries are packaged in `WEB-INF/lib`. Included libraries MAY also contain resource classes and providers as desired. See the Servlet specification for full details on packaging of web applications.

When using a JAX-RS aware servlet container, the `servlet-class` element of the `web.xml` descriptor SHOULD name the application-supplied subclass of `Application`.

When using a non-JAX-RS aware servlet container, the `servlet-class` element of the `web.xml` descriptor SHOULD name the JAX-RS implementation-supplied Servlet class. The application-supplied subclass of `Application` is identified using an `init-param` with a `param-name` of `javax.ws.rs-.Application`.

2.3.3 Other Container

An implementation MAY provide facilities to host a JAX-RS application in other types of container, such facilities are outside the scope of this specification.

Chapter 3

Resources

Using JAX-RS a Web resource is implemented as a resource class and requests are handled by resource methods. This chapter describes resource classes and resource methods in detail.

3.1 Resource Classes

A resource class is a Java class that uses JAX-RS annotations to implement a corresponding Web resource. Resource classes are POJOs that have at least one method annotated with `@Path` or a request method designator.

3.1.1 Lifecycle and Environment

By default a new resource class instance is created for each request to that resource. First the constructor (see section 3.1.2) is called, then any requested dependencies are injected (see section 3.2), then the appropriate method (see section 3.3) is invoked and finally the object is made available for garbage collection.

An implementation MAY offer other resource class lifecycles, mechanisms for specifying these are outside the scope of this specification. E.g. an implementation based on an inversion-of-control framework may support all of the lifecycle options provided by that framework.

3.1.2 Constructors

Root resource classes are instantiated by the JAX-RS runtime and MUST have a public constructor for which the JAX-RS runtime can provide all parameter values. Note that a zero argument constructor is permissible under this rule.

A public constructor MAY include parameters annotated with one of the following: `@Context`, `@HeaderParam`, `@CookieParam`, `@MatrixParam`, `@QueryParam` or `@PathParam`. However, depending on the resource class lifecycle and concurrency, per-request information may not make sense in a constructor. If more than one public constructor is suitable then an implementation MUST use the one with the most parameters. Choosing amongst suitable constructors with the same number of parameters is implementation specific, implementations SHOULD generate a warning about such ambiguity.

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

3.2 Fields and Bean Properties

When a resource class is instantiated, the values of fields and bean properties annotated with one the following annotations are set according to the semantics of the annotation:

@MatrixParam Extracts the value of a URI matrix parameter.

@QueryParam Extracts the value of a URI query parameter.

@PathParam Extracts the value of a URI template parameter.

@CookieParam Extracts the value of a cookie.

@HeaderParam Extracts the value of a header.

@Context Injects an instance of a supported resource, see chapters 5 and 6 for more details.

Because injection occurs at object creation time, use of these annotations (with the exception of `@Context`) on resource class fields and bean properties is only supported for the default per-request resource class lifecycle. An implementation SHOULD warn if resource classes with other lifecycles use these annotations on resource class fields or bean properties.

An implementation is only required to set the annotated field and bean property values of instances created by the implementation runtime. Objects returned by sub-resource locators (see section 3.4.1) are expected to be initialized by their creator and field and bean properties are not modified by the implementation runtime.

Valid parameter types for each of the above annotations are listed in the corresponding Javadoc, however in general (excluding `@Context`) the following types are supported:

1. Primitive types
2. Types that have a constructor that accepts a single `String` argument
3. Types that have a static method named `valueOf` with a single `String` argument
4. `List<T>`, `Set<T>`, or `SortedSet<T>`, where *T* satisfies 2 or 3 above.

The `DefaultValue` annotation may be used to supply a default value for some of the above, see the Javadoc for `DefaultValue` for usage details and rules for generating a value in the absence of this annotation and the requested data. The `Encoded` annotation may be used to disable automatic URI decoding for `@MatrixParam`, `@QueryParam`, and `@PathParam` annotated fields and properties.

A `WebApplicationException` thrown during construction of field or property values using 2 or 3 above is processed directly as described in section 3.3.4. Other exceptions thrown during construction of field or property values using 2 or 3 above are treated as client errors: if the field or property is annotated with `@MatrixParam`, `@QueryParam` or `@PathParam` then an implementation MUST generate a `WebApplicationException` that wraps the thrown exception with a not found response (404 status) and no entity; if the field or property is annotated with `@HeaderParam` or `@CookieParam` then an implementation MUST generate a `WebApplicationException` that wraps the thrown exception with a client error response (400 status) and no entity. The `WebApplicationException` MUST be then be processed as described in section 3.3.4.

3.3 Resource Methods

Resource methods are methods of a resource class annotated with a request method designator. They are used to handle requests and MUST conform to certain restrictions described in this section.

A request method designator is a runtime annotation that is annotated with the `@HttpMethod` annotation. JAX-RS defines a set of request method designators for the common HTTP methods: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@HEAD`. Users may define their own custom request method designators including alternate designators for the common HTTP methods.

3.3.1 Visibility

Only `public` methods may be exposed as resource methods. An implementation SHOULD warn users if a non-`public` method carries a method designator or `@Path` annotation.

3.3.2 Parameters

When a resource method is invoked, parameters annotated with `@FormParam` or one of the annotations listed in section 3.2 are mapped from the request according to the semantics of the annotation. Similar to fields and bean properties:

- The `DefaultValue` annotation may be used to supply a default value for parameters
- The `Encoded` annotation may be used to disable automatic URI decoding of parameter values
- Exceptions thrown during construction of parameter values are treated the same as exceptions thrown during construction of field or bean property values, see section 3.2.

3.3.2.1 Entity Parameters

The value of a non-annotated parameter, called the entity parameter, is mapped from the request entity body. Conversion between an entity body and a Java type is the responsibility of an entity provider, see section 4.2.

Resource methods MUST NOT have more than one parameter that is not annotated with one of the above-listed annotations.

3.3.3 Return Type

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

`void` Results in an empty entity body with a 204 status code.

`Response` Results in an entity body mapped from the entity property of the `Response` with the status code specified by the status property of the `Response`. A `null` return value results in a 204 status code. If the status property of the `Response` is not set: a 200 status code is used for a non-`null` entity property and a 204 status code is used if the entity property is `null`.

GenericEntity Results in an entity body mapped from the Entity property of the GenericEntity. If the return value is not null a 200 status code is used, a null return value results in a 204 status code.

Other Results in an entity body mapped from the class of the returned instance. If the return value is not null a 200 status code is used, a null return value results in a 204 status code.

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

Conversion between a Java object and an entity body is the responsibility of an entity provider, see section 4.2. The return type of a resource method and the type of the returned instance are used to determine the raw type and generic type supplied to the isWritable method of MessageBodyWriteras follows:

Return Type	Returned Instance[1]	Raw Type	Generic Type
GenericEntity	GenericEntity or subclass	RawType property	Type property
Response	GenericEntity or subclass	RawType property	Type property
Response	Object or subclass	Class of instance	Class of instance
Other	Return type or subclass	Class of instance	Generic type of return type

Table 3.1: Determining raw and generic types of return values

To illustrate the above consider a method that always returns an instance of ArrayList<String> either directly or wrapped in some combination of Response and GenericEntity. The resulting raw and generic types are shown below.

Return Type	Returned Instance	Raw Type	Generic Type
GenericEntity	GenericEntity<List<String>>	ArrayList<?>	List<String>
Response	GenericEntity<List<String>>	ArrayList<?>	List<String>
Response	ArrayList<String>	ArrayList<?>	ArrayList<?>
List<String>	ArrayList<String>	ArrayList<?>	List<String>

Table 3.2: Example raw and generic types of return values

3.3.4 Exceptions

A resource method, sub-resource method or sub-resource locator may throw any checked or unchecked exception. An implementation MUST catch all exceptions and process them as follows:

1. Instances of WebApplicationException MUST be mapped to a response as follows. If the response property of the exception does not contain an entity and an exception mapping provider (see section 4.4) is available for WebApplicationException an implementation MUST use the provider to create a new Response instance, otherwise the response property is used directly. The resulting Response instance is then processed according to section 3.3.3.

2. If an exception mapping provider (see section 4.4) is available for the exception or one of its superclasses, an implementation MUST use the provider whose generic type is the nearest superclass of

[1] Or Entity property of returned instance if return type is Response or a subclass thereof.

the exception to create a `Response` instance that is then processed according to section 3.3.3. If the exception mapping provider throws an exception while creating a `Response` then return a server error (status code 500) response to the client.

3. Unchecked exceptions and errors MUST be re-thrown and allowed to propagate to the underlying container.
4. Checked exceptions and throwables that cannot be thrown directly MUST be wrapped in a container-specific exception that is then thrown and allowed to propagate to the underlying container. Servlet-based implementations MUST use `ServletException` as the wrapper. JAX-WS `Provider`-based implementations MUST use `WebServiceException` as the wrapper.

Note: *Items 3 and 4 allow existing container facilities (e.g. a Servlet filter or error pages) to be used to handle the error if desired.*

3.3.5 HEAD and OPTIONS

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

1. Call a method annotated with a request method designator for `HEAD` or, if none present,
2. Call a method annotated with a request method designator for `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 a request method designator for `OPTIONS` or, if none present,
2. Generate an automatic response using the metadata provided by the JAX-RS annotations on the matching class and its methods.

3.4 URI Templates

A root resource class is anchored in URI space using the `@Path` annotation. The value of the annotation is a relative URI path template whose base URI is provided by the deployment context.

A URI path template is a string with zero or more embedded parameters that, when values are substituted for all the parameters, is a valid URI[5] path. The Javadoc for the `@Path` annotation describes their syntax. E.g.:

```
@Path("widgets/{id}")
public class Widget {
  ...
}
```

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

Note: *Because '{'and '}' are not part of either the reserved or unreserved productions of URI[5] they will not appear in a valid URI.*

The value of the annotation is automatically encoded, e.g. the following two lines are equivalent:

```
@Path("widget list/{id}")
@Path("widget%20list/{id}")
```

Template parameters can optionally specify the regular expression used to match their values. The default value matches any text and terminates at the end of a path segment but other values can be used to alter this behavior, e.g.:

```
@Path("widgets/{path:.+}")
public class Widget {
  ...
}
```

In the above example the `Widget` resource class will be matched for any request whose path starts with `widgets` and contains at least one more path segment; the value of the `path` parameter will be the request path following `widgets`. E.g. given the request path `widgets/small/a` the value of `path` would be `small/a`.

3.4.1 Sub Resources

Methods of a resource class that are annotated with `@Path` are either sub-resource methods or sub-resource locators. Sub-resource methods handle a HTTP request directly whilst sub-resource locators return an object that will handle a HTTP request. The presence or absence of a request method designator (e.g. @GET) differentiates between the two:

Present Such methods, known as *sub-resource methods*, are treated like a normal resource method (see section 3.3) 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[2].

Absent Such methods, known as *sub-resource locators*, are used to dynamically resolve the object that will handle the request. Any returned object is treated as a resource class instance and used to either handle the request or to further resolve the object that will handle the request, see 3.7 for further details. An implementation MUST dynamically determine the class of object returned rather than relying on the static sub-resource locator return type since the returned instance may be a subclass of the declared type with potentially different annotations, see section 3.6 for rules on annotation inheritance. Sub-resource locators may have all the same parameters as a normal resource method (see section 3.3) except that they MUST NOT have an entity parameter.

The following example illustrates the difference:

```
@Path("widgets")
public class WidgetsResource {
  @GET
```

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

```
  @Path("offers")
  public WidgetList getDiscounted() {...}

  @Path("{id}")
  public WidgetResource findWidget(@PathParam("id") String id) {
    return new WidgetResource(id);
  }
}

public class WidgetResource {
  public WidgetResource(String id) {...}

  @GET
  public Widget getDetails() {...}
}
```

In the above a `GET` request for the `widgets/offers` resource is handled directly by the `getDiscounted` sub-resource method of the resource class `WidgetsResource` whereas a `GET` request for `widgets/`*`xxx`* is handled by the `getDetails` method of the `WidgetResource` resource class.

Note: *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.*

3.5 Declaring Media Type Capabilities

Application classes can declare the supported request and response media types using the `@Consumes` and `@Produces` annotations respectively. These annotations MAY be applied to a resource method, a resource class, or to an entity provider (see section 4.2.3). Use of these annotations on a resource method overrides any on the resource class or on an entity provider for a method argument or return type. In the absence of either of these annotations, support for any media type (“*/*”) is assumed.

The following example illustrates the use of these annotations:

```
@Path("widgets")
@Produces("application/widgets+xml")
public class WidgetsResource {

  @GET
  public Widgets getAsXML() {...}

  @GET
  @Produces("text/html")
  public String getAsHtml() {...}

  @POST
  @Consumes("application/widgets+xml")
  public void addWidget(Widget widget) {...}
}

@Provider
@Produces("application/widgets+xml")
```

```
public class WidgetsProvider implements MessageBodyWriter<Widgets> {...}

@Provider
@Consumes("application/widgets+xml")
public class WidgetProvider implements MessageBodyReader<Widget> {...}
```

In the above:

- The `getAsXML` resource method will be called for `GET` requests that specify a response media type of `application/widgets+xml`. It returns a `Widgets` instance that will be mapped to that format using the `WidgetsProvider` class (see section 4.2 for more information on `MessageBodyWriter`).
- The `getAsHtml` resource method will be called for `GET` requests that specify a response media type of `text/html`. It returns a `String` containing `text/html` that will be written using the default implementation of `MessageBodyWriter<String>`.
- The `addWidget` resource method will be called for `POST` requests that contain an entity of the media type `application/widgets+xml`. The value of the `widget` parameter will be mapped from the request entity using the `WidgetProvider` class (see section 4.2 for more information on `Message-BodyReader`).

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

3.6 Annotation Inheritance

JAX-RS annotations MAY be used on the methods of a super-class or an implemented interface. Such annotations are inherited by a corresponding sub-class or implementation class method provided that method does not have any of its own JAX-RS annotations. Annotations on a super-class take precedence over those on an implemented interface. If a subclass or implementation method has any JAX-RS annotations then *all* of the annotations on the super class or interface method are ignored. E.g.:

```
public interface ReadOnlyAtomFeed {
  @GET @Produces("application/atom+xml")
  Feed getFeed();
}

@Path("feed")
public class ActivityLog implements ReadOnlyAtomFeed {
  public Feed getFeed() {...}
}
```

In the above, `ActivityLog.getFeed` inherits the `@GET` and `@Produces` annotations from the interface. Conversely:

```
@Path("feed")
public class ActivityLog implements ReadOnlyAtomFeed {
  @Produces("application/atom+xml")
  public Feed getFeed() {...}
}
```

In the above, the `@GET` annotation on `ReadOnlyAtomFeed.getFeed` is not inherited by `Activity-Log-.getFeed` and it would require its own request method designator since it redefines the `@Produces` annotation.

3.7 Matching Requests to Resource Methods

This section describes how a request is matched to a resource class and method. Implementations are not required to use the algorithm as written but MUST produce results equivalent to those produced by the algorithm.

3.7.1 Request Preprocessing

Prior to matching, request URIs are normalized[3] by following the rules for case, path segment, and percent encoding normalization described in section 6.2.2 of RFC 3986[5]. The normalized request URI MUST be reflected in the URIs obtained from an injected `UriInfo`.

3.7.2 Request Matching

A request is matched to the corresponding resource method or sub-resource method by comparing the normalized request URI (see section 3.7.1), the media type of any request entity, and the requested response entity format to the metadata annotations on the resource classes and their methods. If no matching resource method or sub-resource method can be found then an appropriate error response is returned. Matching of requests to resource methods proceeds in three stages as follows:

1. Identify the root resource class:
   (a) Set $U =$ request URI path, $C = \{$root resource classes$\}, E = \{\}$
   (b) For each class in C add a regular expression (computed using the function $R(A)$ described in section 3.7.3) to E as follows:
      - Add $R(T_{\text{class}})$ where T_{class} is the URI path template specified for the class.
   (c) Filter E by matching each member against U as follows:
      - Remove members that do not match U.
      - Remove members for which the final regular expression capturing group (henceforth simply referred to as a capturing group) value is neither empty nor '/' and the class associated with $R(T_{\text{class}})$ had no sub-resource methods or locators.
   (d) If E is empty then no matching resource can be found, the algorithm terminates and an implementation MUST generate a `WebApplicationException` with a not found response (HTTP 404 status) and no entity. The exception MUST be processed as described in section 3.3.4.
   (e) Sort E using the number of literal characters[4] in each member as the primary key (descending order), the number of capturing groups as a secondary key (descending order) and the number of capturing groups with non-default regular expressions (i.e. not '([ˆ/]+?)') as the tertiary key (descending order).

[3]Note: some containers might perform this functionality prior to passing the request to an implementation.
[4]Here, literal characters means those not resulting from template variable substitution.

(f) Set R_{match} to be the first member of E, set U to be the value of the final capturing group of R_{match} when matched against U, and instantiate an object O of the associated class.

2. Obtain the object that will handle the request and a set of candidate methods:

(a) If U is null or '/', set

$$M = \{\text{resource methods of } O \text{ (excluding sub resource methods)}\}$$

and go to step 3

(b) Set $C = \text{class of } O, E = \{\}$

(c) For class C add regular expressions to E for each sub-resource method and locator as follows:

i. For each sub-resource method, add $R(T_{\text{method}})$ where T_{method} is the URI path template of the sub-resource method.

ii. For each sub-resource locator, add $R(T_{\text{locator}})$ where T_{locator} is the URI path template of the sub-resource locator.

(d) Filter E by matching each member against U as follows:

- Remove members that do not match U.
- Remove members derived from T_{method} (those added in step 2(c)i) for which the final capturing group value is neither empty nor '/'.

(e) If E is empty then no matching resource can be found, the algorithm terminates and an implementation MUST generate a `WebApplicationException` with a not found response (HTTP 404 status) and no entity. The exception MUST be processed as described in section 3.3.4.

(f) Sort E using the number of literal characters in each member as the primary key (descending order), the number of capturing groups as a secondary key (descending order), the number of capturing groups with non-default regular expressions (i.e. not '([^/]+?)') as the tertiary key (descending order), and the source of each member as quaternary key sorting those derived from T_{method} ahead of those derived from T_{locator}.

(g) Set R_{match} to be the first member of E

(h) If R_{match} was derived from T_{method}, then set

$$M = \{\text{subresource methods of } O \text{ where } R(T_{\text{method}}) = R_{\text{match}}\}$$

and go to step 3.

(i) Set U to be the value of the final capturing group of $R(T_{\text{match}})$ when matched against U, invoke the sub-resource locator method of O and set O to the value returned from that method.

(j) Go to step 2a.

3. Identify the method that will handle the request:

(a) Filter M by removing members that do not meet the following criteria:

- The request method is supported. If no methods support the request method an implementation MUST generate a `WebApplicationException` with a method not allowed response (HTTP 405 status) and no entity. The exception MUST be processed as described in section 3.3.4. Note the additional support for `HEAD` and `OPTIONS` described in section 3.3.5.

- The media type of the request entity body (if any) is a supported input data format (see section 3.5). If no methods support the media type of the request entity body an implementation MUST generate a `WebApplicationException` with an unsupported media type response (HTTP 415 status) and no entity. The exception MUST be processed as described in section 3.3.4.
- At least one of the acceptable response entity body media types is a supported output data format (see section 3.5). If no methods support one of the acceptable response entity body media types an implementation MUST generate a `WebApplicationException` with a not acceptable response (HTTP 406 status) and no entity. The exception MUST be processed as described in section 3.3.4.

(b) Sort M in descending order as follows:

- The primary key is the media type of input data. Methods whose `@Consumes` value is the best match for the media type of the request are sorted first.
- The secondary key is the `@Produces` value. Methods whose value of `@Produces` best matches the value of the request accept header are sorted first.

Determining the best matching media types follows the general rule: n/m $>$ n/* $>$ */*, i.e. a method that explicitly consumes the request media type or produces one of the requested media types is sorted before a method that consumes or produces */*. Quality parameter values in the accept header are also considered such that methods that produce media types with a higher acceptable q-value are sorted ahead of those with a lower acceptable q-value (i.e. n/m;q=1.0 $>$ n/m;q=0.7) - see section 14.1 of [4] for more details.

(c) The request is dispatched to the first Java method in the set[5].

3.7.3 Converting URI Templates to Regular Expressions

The function $R(A)$ converts a URI path template annotation A into a regular expression as follows:

1. URI encode the template, ignoring URI template variable specifications.
2. Escape any regular expression characters in the URI template, again ignoring URI template variable specifications.
3. Replace each URI template variable with a capturing group containing the specified regular expression or '([^/]+?)' if no regular expression is specified.
4. If the resulting string ends with '/' then remove the final character.
5. Append '(/.*)?' to the result.

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 `@PathParam` or `UriInfo.getTemplateParameters`.

3.8 Determining the MediaType of Responses

In many cases it is not possible to statically determine the media type of a response. The following algorithm is used to determine the response media type, M_{selected}, at run time:

[5] Step 3a ensures the set contains at least one member.

1. If the method returns an instance of `Response` whose metadata includes the response media type ($M_{\text{specified}}$) then set $M_{\text{selected}} = M_{\text{specified}}$, finish.
2. Gather the set of producible media types P:
   - If the method is annotated with `@Produces`, set $P = \{V(\text{method})\}$ where $V(t)$ represents the values of `@Produces` on the specified target t.
   - Else if the class is annotated with `@Produces`, set $P = \{V(\text{class})\}$.
   - Else set $P = \{V(\text{writers})\}$ where 'writers' is the set of `MessageBodyWriter` that support the class of the returned entity object.
3. If $P = \{\}$, set $P = \{\text{'*/*'}\}$
4. Obtain the acceptable media types A. If $A = \{\}$, set $A = \{\text{'*/*'}\}$
5. Set $M = \{\}$. For each member of A, a:
   - For each member of P, p:
     - If a is compatible with p, add $S(a, p)$ to M, where the function S returns the most specific media type of the pair with the q-value of a.
6. If $M = \{\}$ then generate a `WebApplicationException` with a not acceptable response (HTTP 406 status) and no entity. The exception MUST be processed as described in section 3.3.4. Finish.
7. Sort M in descending order, with a primary key of specificity (n/m $>$ n/* $>$ */*) and secondary key of q-value.
8. For each member of M, m:
   - If m is a concrete type, set $M_{\text{selected}} = m$, finish.
9. If M contains '*/*' or 'application/*', set $M_{\text{selected}} =$ 'application/octet-stream', finish.
10. Generate a `WebApplicationException` with a not acceptable response (HTTP 406 status) and no entity. The exception MUST be processed as described in section 3.3.4. Finish.

Note that the above renders a response with a default media type of 'application/octet-stream' when a concrete type cannot be determined. It is RECOMMENDED that `MessageBodyWriter` implementations specify at least one concrete type via `@Produces`.

Chapter 4

Providers

The JAX-RS runtime is extended using application-supplied provider classes. A provider is annotated with `@Provider` and implements one or more interfaces defined by JAX-RS.

4.1 Lifecycle and Environment

By default a single instance of each provider class is instantiated for each JAX-RS application. First the constructor (see section 4.1.1) is called, then any requested dependencies are injected (see chapter 5), then the appropriate provider methods may be called multiple times (simultaneously), and finally the object is made available for garbage collection. Section 5.2.5 describes how a provider obtains access to other providers via dependency injection.

An implementation MAY offer other provider lifecycles, mechanisms for specifying these are outside the scope of this specification. E.g. an implementation based on an inversion-of-control framework may support all of the lifecycle options provided by that framework.

4.1.1 Constructors

Provider classes are instantiated by the JAX-RS runtime and MUST have a public constructor for which the JAX-RS runtime can provide all parameter values. Note that a zero argument constructor is permissible under this rule.

A public constructor MAY include parameters annotated with `@Context`- chapter 5 defines the parameter types permitted for this annotation. Since providers may be created outside the scope of a particular request, only deployment-specific properties may be available from injected interfaces at construction time - request-specific properties are available when a provider method is called. If more than one public constructor can be used then an implementation MUST use the one with the most parameters. Choosing amongst constructors with the same number of parameters is implementation specific, implementations SHOULD generate a warning about such ambiguity.

4.2 Entity Providers

Entity providers supply mapping services between representations and their associated Java types. Entity providers come in two flavors: `MessageBodyReader` and `MessageBodyWriter` described below. In the absence of a suitable entity provider, JAX-RS implementations are REQUIRED to use to the JavaBeans Activation Framework[11] to try to obtain a suitable data handler to perform the mapping instead.

4.2.1 Message Body Reader

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

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 3.7 describes how the Java method is chosen.
2. Select the set of `MessageBodyReader` classes that support the media type of the request, see section 4.2.3.
3. Iterate through the selected `MessageBodyReader` classes and, utilizing the `isReadable` method of each, choose a `MessageBodyReader` provider that supports the desired Java type.
4. If step 3 locates a suitable `MessageBodyReader` then use its `readFrom` method to map the entity body to the desired Java type.
5. Else if a suitable data handler can be found using the JavaBeans Activation Framework[11] then use it to map the entity body to the desired Java type.
6. Else generate a `WebApplicationException` that contains an unsupported media type response (HTTP 415 status) and no entity. The exception MUST be processed as described in section 3.3.4.

A `MessageBodyReader.readFrom` method MAY throw `WebApplicationException`. If thrown, the resource method is not invoked and the exception is treated as if it originated from a resource method, see section 3.3.4.

4.2.2 Message Body Writer

The `MessageBodyWriter` interface defines the contract between the JAX-RS runtime and components that provide mapping services from a Java type to a representation. A class wishing to provide such a service implements the `MessageBodyWriter` interface and is annotated with `@Provider`.

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. Determine the media type of the response, see section 3.8.

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

3. Select the set of `MessageBodyWriter` providers that support (see section 4.2.3) the object and media type of the response entity body.

4. Sort the selected `MessageBodyWriter` providers as described in section 4.2.3.

5. Iterate through the sorted `MessageBodyWriter` providers and, utilizing the `isWriteable` method of each, choose an `MessageBodyWriter` that supports the object that will be mapped to the entity body.

6. If step 5 locates a suitable `MessageBodyWriter` then use its `writeTo` method to map the object to the entity body.

7. Else if a suitable data handler can be found using the JavaBeans Activation Framework[11] then use it to map the object to the entity body.

8. Else generate a `WebApplicationException` with an internal server error response (HTTP 500 status) and no entity. The exception MUST be processed as described in section 3.3.4.

A `MessageBodyWriter.write` method MAY throw `WebApplicationException`. If thrown before the response is committed, the exception is treated as if it originated from a resource method, see section 3.3.4. To avoid an infinite loop, implementations SHOULD NOT attempt to map exceptions thrown during serialization of an response previously mapped from an exception and SHOULD instead simply return a server error (status code 500) response.

4.2.3 Declaring Media Type Capabilities

Message body readers and writers MAY restrict the media types they support using the `@Consumes` and `@Produces` annotations respectively. The absence of these annotations is equivalent to their inclusion with media type ("*/*"), i.e. absence implies that any media type is supported. An implementation MUST NOT use an entity provider for a media type that is not supported by that provider.

When choosing an entity provider 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 */*.

4.2.4 Standard Entity Providers

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

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

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

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

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

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

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

javax.xml.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).

StreamingOutput All media types (*/*), MessageBodyWriter only.

The implementation-supplied entity provider(s) for javax.xml.bind.JAXBElement and application-supplied JAXB classes MUST use JAXBContext instances provided by application-supplied context resolvers, see section 4.3. If an application does not supply a JAXBContext for a particular type, the implementation-supplied entity provider MUST use its own default context instead.

When writing responses, implementations SHOULD respect application-supplied character set metadata and SHOULD use UTF-8 if a character set is not specified by the application or if the application specifies a character set that is unsupported.

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

4.2.5 Transfer Encoding

Transfer encoding for inbound data is handled by a component of the container or the JAX-RS runtime. MessageBodyReader providers always operate on the decoded HTTP entity body rather than directly on the HTTP message body.

A JAX-RS runtime or container MAY transfer encode outbound data or this MAY be done by application code.

4.2.6 Content Encoding

Content encoding is the responsibility of the application. Application-supplied entity providers MAY perform such encoding and manipulate the HTTP headers accordingly.

4.3 Context Providers

Context providers supply context to resource classes and other providers. A context provider class implements the ContextResolver<T> interface and is annotated with @Provider. E.g. an application wishing to provide a customized JAXBContext to the default JAXB entity providers would supply a class implementing ContextResolver<JAXBContext>.

Context providers MAY return null from the getContext method if they do not wish to provide their context for a particular Java type. E.g. a JAXB context provider may wish to only provide the context for certain JAXB classes. Context providers MAY also manage multiple contexts of the same type keyed to different Java types.

4.3.1 Declaring Media Type Capabilities

Context provider implementations MAY restrict the media types they support using the `@Produces` annotation. The absence of this annotation is equivalent to its inclusion with media type (“*/*”), i.e. absence implies that any media type is supported.

When choosing a context provider 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 type is sorted before a provider that lists */*.

4.4 Exception Mapping Providers

When a resource class or provider method throws an exception, the JAX-RS runtime will attempt to map the exception to a suitable HTTP response - see section 3.3.4. An application can supply exception mapping providers to customize this mapping.

Exception mapping providers map a checked or runtime exception to an instance of `Response`. An exception mapping provider implements the `ExceptionMapper<T>` interface and is annotated with `@Provider`. When a resource method throws an exception for which there is an exception mapping provider, the matching provider is used to obtain a `Response` instance. The resulting `Response` is processed as if the method throwing the exception had instead returned the `Response`, see section 3.3.3.

When choosing an exception mapping provider to map an exception, an implementation MUST use the provider whose generic type is the nearest superclass of the exception.

Chapter 5

Context

JAX-RS provides facilities for obtaining and processing information about the application deployment context and the context of individual requests. Such information is available to both root resource classes (see chapter 3) and providers (see chapter 4). This chapter describes these facilities.

5.1 Concurrency

Context is specific to a particular request but instances of certain JAX-RS components (providers and resource classes with a lifecycle other than per-request) may need to support multiple concurrent requests. When injecting an instance of one of the types listed in section 5.2, the instance supplied MUST be capable of selecting the correct context for a particular request. Use of a thread-local proxy is a common way to achieve this.

5.2 Context Types

This section describes the types of context available to resource classes and providers.

5.2.1 URIs and URI Templates

An instance of `UriInfo` can be injected into a class field or method parameter using the `@Context` 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:

```
@GET
@Produces{"text/plain"}
public String listQueryParamNames(@Context UriInfo info) {
  StringBuilder buf = new StringBuilder();
  for (String param: info.getQueryParameters().keySet()) {
    buf.append(param);
    buf.append("\n");
  }
  return buf.toString();
}
```

Note that the methods of `UriInfo` provide access to request URI information following the pre-processing described in section 3.7.1.

5.2.2 Headers

An instance of `HttpHeaders` can be injected into a class field or method parameter using the `@Context` 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:

```
@GET
@Produces{"text/plain"}
public String listHeaderNames(@Context HttpHeaders headers) {
  StringBuilder buf = new StringBuilder();
  for (String header: headers.getRequestHeaders().keySet()) {
    buf.append(header);
    buf.append("\n");
  }
  return buf.toString();
}
```

Note that the methods of `HttpHeaders` provide access to request information following the pre-processing described in section 3.7.1.

Response headers may be provided using the `Response` class, see 3.3.3 for more details.

5.2.3 Content Negotiation and Preconditions

JAX-RS simplifies support for content negotiation and preconditions using the `Request` interface. An instance of `Request` can be injected into a class field or method parameter using the `@Context` annotation. The methods of `Request` allow a caller to determine the best matching representation variant and to evaluate whether the current state of the resource matches any preconditions in the request. Precondition support methods return a `ResponseBuilder` that can be returned to the client to inform it that the request preconditions were not met. E.g. the following checks if the current entity tag matches any preconditions in the request before updating the resource:

```
@PUT
public Response updateFoo(@Context Request request, Foo foo) {
    EntityTag tag = getCurrentTag();
    ResponseBuilder responseBuilder = request.evaluatePreconditions(tag);
    if (responseBuilder != null)
      return responseBuilder.build();
    else
      return doUpdate(foo);
}
```

The application could also set the content location, expiry date and cache control information into the returned `ResponseBuilder` before building the response.

5.2.4 Security Context

The `SecurityContext` interface provides access to information about the security context of the current request. An instance of `SecurityContext` can be injected into a class field or method parameter using the `@Context` annotation. The methods of `SecurityContext` provide access to the current user principle, information about roles assumed by the requester, whether the request arrived over a secure channel and the authentication scheme used.

5.2.5 Providers

The `Providers` interface allows for lookup of provider instances based on a set of search criteria. An instance of `Providers` can be injected into a class field or method parameter using the `@Context` annotation.

This interface is expected to be primarily of interest to provider authors wishing to use other providers functionality.

Chapter 6

Environment

The container-managed resources available to a JAX-RS root resource class or provider depend on the environment in which it is deployed. Section 5.2 describes the types of context available regardless of container. The following sections describe the additional container-managed resources available to a JAX-RS root resource class or provider deployed in a variety of environments.

6.1 Servlet Container

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

An injected `HttpServletRequest` allows a resource method to stream the contents of a request entity. If the resource 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 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.

6.2 Java EE Container (Non-normative)

This section describes the additional features anticipated to be available to a JAX-RS application hosted in a Java EE 6 container. It is planned that JAX-RS will be finalized prior to Java EE 6 so the contents of this section are preliminary and subject to change. Nothing in this section should be considered a conformance requirement.

JAX-RS root resource classes and providers are supplied with the same resource injection capabilities as are provided for a Servlet instance running in a Java EE Web container. In particular the following annotations may be used according to their individual semantics: `@Resource`, `@Resources`, `@EJB`, `@EJBs`, `@WebServiceRef`, `@WebServiceRefs`, `@PersistenceContext`, `@PersistenceContexts`, `@PersistenceUnit` and `@PersistenceUnits`.

JAX-RS root resource classes and providers may also make use of the following JSR 250 lifecycle management and security annotations: `@PostConstruct`, `@PreDestroy`, `@RunAs`, `@RolesAllowed`, `@Permit-`

`All`, `@DenyAll` and `@DeclareRoles`.

6.3 Other

Other container technologies MAY specify their own set of injectable resources but MUST, at a minimum, support access to the types of context listed in section 5.2.

Chapter 7

Runtime Delegate

`RuntimeDelegate` is an abstract factory class that provides various methods for the creation of objects that implement JAX-RS APIs. These methods are designed for use by other JAX-RS API classes and are not intended to be called directly by applications. `RuntimeDelegate` allows the standard JAX-RS API classes to use different JAX-RS implementations without any code changes.

An implementation of JAX-RS MUST provide a concrete subclass of `RuntimeDelegate`. Using the supplied `RuntimeDelegate` this can be provided to JAX-RS in one of two ways:

1. An instance of `RuntimeDelegate` can be instantiated and injected using its static method `setInstance`. In this case the implementation is responsible for creating the instance; this option is intended for use with implementations based on IoC frameworks.
2. The class to be used can be configured, see section 7.1. In this case JAX-RS is responsible for instantiating an instance of the class and the configured class MUST have a public constructor which takes no arguments.

Note that an implementation MAY supply an alternate implementation of the `RuntimeDelegate` API class (provided it passes the TCK signature test and behaves according to the specification) that supports alternate means of locating a concrete subclass.

A JAX-RS implementation may rely on a particular implementation of `RuntimeDelegate` being used – applications SHOULD NOT override the supplied `RuntimeDelegate` instance with an application-supplied alternative and doing so may cause unexpected problems.

7.1 Configuration

If not supplied by injection, the supplied `RuntimeDelegate` API class obtains the concrete implementation class using the following algorithm. The steps listed below are performed in sequence and, at each step, at most one candidate implementation class name will be produced. The implementation will then attempt to load the class with the given class name using the current context class loader or, missing one, the java.lang.Class.forName(String) method. As soon as a step results in an implementation class being successfully loaded, the algorithm terminates.

1. If a resource with the name of `META-INF/services/javax.ws.rs.ext.RuntimeDelegate` exists, then its first line, if present, is used as the UTF-8 encoded name of the implementation class.

2. If the `${java.home}/lib/jaxrs.properties` file exists and it is readable by the `java.util.Properties.load(InputStream)` method and it contains an entry whose key is `javax.ws.rs.ext.RuntimeDelegate`, then the value of that entry is used as the name of the implementation class.

3. If a system property with the name `javax.ws.rs.ext.RuntimeDelegate` is defined, then its value is used as the name of the implementation class.

4. Finally, a default implementation class name is used.

Appendix A

Summary of Annotations

Annotation	Target	Description
`Consumes`	Type or method	Specifies a list of media types that can be consumed.
`Produces`	Type or method	Specifies a list of media types that can be produced.
`GET`	Method	Specifies that the annotated method handles HTTP GET requests.
`POST`	Method	Specifies that the annotated method handles HTTP POST requests.
`PUT`	Method	Specifies that the annotated method handles HTTP PUT requests.
`DELETE`	Method	Specifies that the annotated method handles HTTP DELETE requests.
`HEAD`	Method	Specifies that the annotated method handles HTTP HEAD requests. Note that HEAD may be automatically handled, see section 3.3.5.
`Path`	Type or method	Specifies a relative path for a resource. When used on a class this annotation identifies that class as a root resource. When used on a method this annotation identifies a sub-resource method or locator.
`PathParam`	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from the request URI path. The value of the annotation identifies the name of a URI template parameter.
`QueryParam`	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a URI query parameter. The value of the annotation identifies the name of a query parameter.
`FormParam`	Parameter, field or method	Specifies that the value of a method parameter is to be extracted from a form parameter in a request entity body. The value of the annotation identifies the name of a form parameter. Note that whilst the annotation target allows use on fields and methods, the specification only requires support for use on resource method parameters.

Annotation	Target	Description
`MatrixParam`	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a URI matrix parameter. The value of the annotation identifies the name of a matrix parameter.
`CookieParam`	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP cookie. The value of the annotation identifies the name of a the cookie.
`HeaderParam`	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP header. The value of the annotation identifies the name of a HTTP header.
`Encoded`	Type, constructor, method, field or parameter	Disables automatic URI decoding for path, query, form and matrix parameters.
`DefaultValue`	Parameter, field or method	Specifies a default value for a field, property or method parameter annotated with `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@FormParam` or `@HeaderParam`. The specified value will be used if the corresponding query or matrix parameter is not present in the request URI, if the corresponding form parameter is not in the request entity body, or if the corresponding HTTP header is not included in the request.
`Context`	Field, method or parameter	Identifies an injection target for one of the types listed in section 5.2 or the applicable section of chapter 6.
`HttpMethod`	Annotation	Specifies the HTTP method for a request method designator annotation.
`Provider`	Type	Specifies that the annotated class implements a JAX-RS extension interface.

1

Appendix B

HTTP Header Support

The following table lists HTTP headers that are directly supported, either automatically by a JAX-RS implementation runtime or by an application using the JAX-RS API. Any request header may be obtained using `HttpHeaders`, see section 5.2.2; response headers not listed here may set using the `ResponseBuilder-.header` method.

Header	Description
`Accept`	Used by runtime when selecting a resource method, compared to value of `@Produces` annotation, see section 3.5.
`Accept-Charset`	Processed by runtime if application uses `Request.selectVariant` method, see section 5.2.3.
`Accept-Encoding`	Processed by runtime if application uses `Request.selectVariant` method, see section 5.2.3.
`Accept-Language`	Processed by runtime if application uses `Request.selectVariant` method, see section 5.2.3.
`Allow`	Included in automatically generated 405 error responses (see section 3.7.2) and automatically generated responses to OPTIONS requests (see section 3.3.5).
`Authorization`	Depends on container, information available via `SecurityContext`, see section 5.2.4.
`Cache-Control`	See `CacheControl` class and `ResponseBuilder.cacheControl` method.
`Content-Encoding`	Response header set by application using `Response.ok` or `ResponseBuilder.variant`.
`Content-Language`	Response header set by application using `Response.ok`, `ResponseBuilder.language`, or `ResponseBuilder.variant`.
`Content-Length`	Processed automatically for requests, set automatically in responses if value is provided by the `MessageBodyWriter` used to serialize the response entity.
`Content-Type`	Request header used by runtime when selecting a resource method, compared to value of `@Consumes` annotation, see section 3.5. Response header either set by application using `Response.ok`, `ResponseBuilder.type`, or `ResponseBuilder.variant`, or set automatically by runtime (see section 3.8).
`Cookie`	See `Cookie` class and `HttpHeaders.getCookies` method.

Header	Description
`Date`	Included in responses automatically as per HTTP/1.1.
`ETag`	See `EntityTag` class, `Response.notModified` method and `ResponseBuilder.tag` method.
`Expect`	Depends on underlying container.
`Expires`	Set by application using the `ResponseBuilder.expires` method.
`If-Match`	Processed by runtime if application uses corresponding `Request.evaluatePreconditions` method, see section 5.2.3.
`If-Modified-Since`	Processed by runtime if application uses corresponding `Request.evaluatePreconditions` method, see section 5.2.3.
`If-None-Match`	Processed by runtime if application uses corresponding `Request.evaluatePreconditions` method, see section 5.2.3.
`If-Unmodified-Since`	Processed by runtime if application uses corresponding `Request.evaluatePreconditions` method, see section 5.2.3.
`Last-Modified`	Set by application using the `ResponseBuilder.lastModified` method.
`Location`	Set by application using the applicable `Response` method or directly using the `ResponseBuilder.location` method.
`Set-Cookie`	See `NewCookie` class and `ResponseBuilder.cookie` method.
`Transfer-Encoding`	See section 4.2.5.
`Vary`	Set by application using `Response.notAcceptable` method or `ResponseBuilder.variants` method.
`WWW-Authenticate`	Depends on container.

1

Appendix C

Change Log

C.1 Changes Since Proposed Final Draft

- Section 3.7.2: Additional sort criteria so that templates with explicit regexs are sorted ahead of those with the default.
- Sections 3.7.2, 3.8, 4.2.3 and 4.3.1: Q-values not used in `@Consumes` or `@Produces`.
- Chapter 7: Clarify that an implementation can supply an alternate `RuntimeDelegate` API class.

C.2 Changes Since Public Review Draft

- Chapter 2: Renamed ApplicationConfig class to Application.
- Chapter 3: UriBuilder reworked to always encode components.
- Sections 3.1.2 and 4.1.1: Added requirement to warn when choice of constructor is ambiguous.
- Section 3.2: `FormParam` no longer required to be supported on fields or properties.
- Section 3.3.3: Added text describing how to determine raw and generic types from method return type and returned instance.
- Section 3.4: Template parameters can specify the regular expression that forms their capturing group.
- Section 3.7.1: Make pre-processed URIs available rather than original request URI. Added URI normalization.
- Section 3.7.1: Removed URI-based content negotiation.
- Section 3.7.2: Reorganized the request matching algorithm to remove redundancy and improve readability, no functional change.
- Section 3.7.3: Changes to regular expressions to eliminate edge cases.
- Section 4.2: Added requirement to use JavaBean Activation Framework when no entity provider can be found.

- Section 4.2.4: Require standard JAXB entity providers to use application-supplied JAXB contexts in preference to their own.
- Section 4.3: Added support for specifying media type capabilities of context providers.
- Section 5.2: Removed `ContextResolver` from list of injectable resources.
- Section 5.2.5: Changed name to Providers, removed entity provider-specific text to reflect more generic capabilities.
- Chapter B: New appendix describing where particular HTTP headers are supported.

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.

[11] Bill Shannon. JavaBeans Activation Framework. JSR, JCP, May 2006. See http://jcp.org/en/jsr/detail?id=925.