# JAX-RS: Java™ API for RESTful Web Services

*Editors Draft*
*March 20, 2008*

Editors:
Marc Hadley
Paul Sandoz

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

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

# 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

Javadocs can be found online at:

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

```
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 3.2.

**Sub-resource locator** A method of a *resource class* that is used to locate sub-resources of the corresponding resource, see section 3.3.1.

**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 3.3.1.

## 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

# 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 `ApplicationConfig`. An implementation MAY provide alternate mechanisms for locating resource classes and providers (e.g. runtime class scanning) but use of `ApplicationConfig` is the only portable means of configuration.

## 2.2 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.

### 2.2.1 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 `ApplicationConfig` 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.

#### 2.2.1.1 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.

### 2.2.2 Servlet

A JAX-RS application is packaged as a Servlet in a `.war` file. The `ApplicationConfig` 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 `ApplicationConfig`.

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 `ApplicationConfig` is identified using an `init-param` with a `param-name` of `javax.ws.rs.ApplicationConfig`.

### 2.2.3 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 chapter 5), then the appropriate method (see section 3.2) 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`- section 3.2.2 defines the parameter types permitted for each annotation. 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 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.

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

## 3.2 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.

### 3.2.1 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.

### 3.2.2 Parameters

When a resource 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` or `@QueryParam`** Either a primitive type, a class wth a constructor that accepts a single `String` argument, or a class with 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. If more than one query or matrix parameter is present in the request URI then the first value (by lexical order) is selected.

**`@PathParam`** Either `PathSegment`, a primitive type, a class wth a constructor that accepts a single `String` argument, or a class with 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.

**`@Context`** The class of the annotated parameter MUST be one of the types defined in chapter 5 or 6.

**`@CookieParam`** Either `Cookie`, a primitive type, a class wth a constructor that accepts a single `String` argument, or a class with a static method named `valueOf` that accepts a single `String` argument.

**`@HeaderParam`** Either a primitive type, a class wth a constructor that accepts a single `String` argument, or a class with a static method named `valueOf` that accepts a single `String` argument.

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 the above-listed annotations. Conversion between an entity body and a Java type is the responsibility of an entity provider, see section 4.3.

### 3.2.3 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 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.

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

Conversion between a Java types and an entity body is the responsibility of an entity provider, see section 4.3.

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.

### 3.2.4 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 3.1** *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 ?*

### 3.2.5 HEAD and OPTIONS

HEAD and OPTIONS requests receive additional automated support. On receipt of aHEAD 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 from the declared metadata of the matching class.

## 3.3 URI Templates

A 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. Root resource classes are anchored directly using a @Path annotation on the class.

**Editors Note 3.2** *Add reference to URI Templates ID when available.*

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. A template parameter is represented as '{'*name*'}' where *name* is the name of the parameter. 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 `encode` property of `@Path` controls whether the value of the annotation is automatically encoded (the default) or not. E.g. the following two lines are equivalent:

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

When automatic encoding is disabled, care must be taken to ensure that the value of the URI template is valid.

The `limited` property of `@Path` 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 a path consisting of arbitrarily many path segments. E.g.:

```
@Path(value="widgets/{path}", limited=false)
public class Widget {
  ...
}
```

In the above example the `Widget` resource class can be used for any request whose path starts with the `widgets`; 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`.

### 3.3.1 Sub Resources

Methods of a resource class that are annotated with `@Path` are either sub-resource methods or sub-resource locators. The differentiator is the presence or absence of request method designator:

**Present** Such methods, known as *sub-resource methods*, are treated like a normal resource method (see section 3.2) 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].

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

**Absent** 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 3.6 for further details.

The following example illustrates the difference:

```
@Path("widgets")
public class WidgetsResource {
  @GET
  @Path("offers")
  public WidgetList getDiscounted() {...}

  @Path("{id}")
  public WidgetResource findWidget(@UriParam("id") String id) {
    return lookupWidget(id);
  }
}
```

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 whatever resource class instance is returned by the `findWidget` sub-resource locator (a `WidgetResource`).

**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.*

## 3.4 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 method, a resource class, or to an entity provider (see section 4.3.3). 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 `@ProduceMime` annotation:

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

  @GET
  public String getAll() {...}

  @GET
  @Path("{id}")
  public Widget getWidget(@UriParam("id") String id) {...}

  @GET
  @Path("{id}/description")
  @ProduceMime("text/html")
```

```
  public String getDescription(@UriParam("id") String id) {...}
}

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

In the above:

- The `getAll` resource method returns a `String` in the `application/xml` format,
- The `getDescription` sub-resource method returns a `String` as `text/html`, and
- The `getWidget` sub-resource method returns a `Widget` entity instance that can be mapped to either `application/xml` or `application/json` using the `WidgetProvider` class (see section 4.3 for more information on `MessageBodyWriter`).

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.

## 3.5 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 @ProduceMime("application/atom+xml")
  Feed getFeed();
}

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

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

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

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

## 3.6 Matching Requests to Resource Methods

This section describes how a request is matched to a resource class and method.

### 3.6.1 Request Preprocessing

For the purposes of matching, Request URIs are preprocessed to support URI-based content negotiation as follows:

1. Set
   - $M = \{$`config.getExtensionMappings().keySet()`$\}$
   - $L = \{$`config.getLanguageMappings().keySet()`$\}$
   - Where `config` is an instance of the application-supplied subclass of `ApplicationConfig`.
2. For each extension (a '.' character followed by one or more alphanumeric characters) $e$ in the final path segment:
   (a) Remove the leading '.' character from $e$
   (b) If $e$ is a member of $M$ or $L$ then remove the corresponding extension from the effective request URI.
   (c) If $e$ is a member of $M$ then set the effective value of the `Accept` header to `config.getExtensionMappings().get(`$e$`)`
   (d) Else if $e$ is a member of $L$ then set the effective value of the `Accept-Language` header to `config.getLanguageMappings().get(`$e$`)`

The above preprocessing MUST NOT impact the URIs obtained from an injected `UriInfo`, in particular extensions removed in step 2b MUST still be present in URIs returned from the methods of `UriInfo`. Similarly the methods of `HttpHeaders` MUST return the actual values of the `Accept` and `Accept-Language` headers rather than the effective values set during preprocessing.

### 3.6.2 Request Matching

A request is matched to the corresponding resource method or sub-resource method by comparing the preprocessed request URI, 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, $C = \{\text{root resource classes}\}, E = \{\}$
   (b) For each class in $C$ add a regular expression (computed using the function $R(A)$ described in section 3.6.3) to $E$ as follows:
      - Add $R(T_{\text{class}})$ where $T_{\text{class}}$ 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 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 not found response (HTTP 404 status).

(e) Sort $E$ using the number of literal characters[2] in each member as the primary key (descending order) and the number of capturing groups as a secondary key (descending order).

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

2. Obtain the object that will handle the request:

   (a) If $U$ is null or '/' 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 $T_{\text{method}}$ is the URI path template of the sub-resource method.

      ii. For each sub-resource locator, add $R(T_{\text{locator}})$ where $T_{\text{locator}}$ 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 $T_{\text{method}}$ (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 not found response (HTTP 404 status).

   (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), and the source of each member as tertiary key sorting those derived from $T_{\text{method}}$ ahead of those derived from $T_{\text{locator}}$.

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

   (h) If $R_{\text{match}}$ was derived from $T_{\text{method}}$ then 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) Find the set of resource methods $M$ of $O$ that meet the following criteria:

   - If $U$ is neither empty nor equal to '/', the method must be annotated with a URI template that, when transformed into a regular expression using the process described in section 3.6.3, matches $U$ with a final capturing group value that is either empty or equal to '/'.
   - 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 3.2.5.

[2]Here, literal characters means those not resulting from template variable substitution.

- The media type of the request entity body (if any) is a supported input data format (see section 3.4). 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 3.4). 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 $M$ a follows:

- The primary key is the media type of input data. Methods whose `@ConsumeMime` value most closely match the media type of the request are sorted first.
- The secondary key is the `@ProduceMime` value. Methods whose value of `@ProduceMime` most closely match the value of the request accept header are sorted first.

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. See section 14.1 of [4] for more details.

(c) If $M$ is not 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.

### 3.6.3 Converting URI Templates to Regular Expressions

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

1. If $A$.encode $=$ true, 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 the URI template variables[3] with the regular expression '(.*?)'.
4. If the resulting string ends with '/' then remove the final character.
5. If $A$.limited $=$ true, append '(/.*)?' to the result, else 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`.

## 3.7 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, $M_{\text{selected}}$, at run time:

1. Gather the set of producible media types $P$:
   - If the method is annotated with `@ProduceMime`, set $P = \{V(\text{method})\}$ where $V(t)$ represents the values of `@ProduceMime` on the specified target $t$.

[3] The regular expression to match a URI path template variable is \{([\w-\. ~]+?)\}.

- Else if the class is annotated with `@ProduceMime`, 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.

2. If $P = \{\}$, set $P = \{\text{'*/*'}\}$
3. Obtain the acceptable media types $A$. If $A = \{\}$, set $A = \{\text{'*/*'}\}$
4. Sort $A$ and $P$ in descending order, each with a primary key of q-value and secondary key of specificity ('n/m' > 'n/*' > '*/*').
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 supplied list.
6. If $M = \{\}$ then return a not acceptable response (HTTP 406 status), finish.
7. For each member of $M, m$:
   - If $m$ is a concrete type, set $M_{\text{selected}} = m$, finish.
8. If $M$ contains '*/*' or 'application/*', set $M_{\text{selected}} =$ 'application/octet-stream', finish.
9. Return a not acceptable response (HTTP 406 status), 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 `@ProduceMime`.

# 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 4.2) 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.

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.

## 4.2 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.

## 4.3 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.

### 4.3.1 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 3.6 describes how the Java method is chosen.
2. Select the set of `MessageBodyReader` classes that support the media type of the request, see section 4.3.3.
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. If step 3 fails to locate a suitable `MessageBodyReader` then generate an unsupported media type response (HTTP 415 status).

### 4.3.2 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. Obtain the effective value of `@ProduceMime` (see section 3.4) and intersect that with the requested response formats to obtain set of permissible media types for the response entity body. Note that section 3.6 ensures that this set will not be empty.
3. Select the set of `MessageBodyWriter` providers that support (see section 4.3.3) one or more of the permissible media types for the response entity body.
4. Sort the selected `MessageBodyWriter` providers as described in section 4.3.3.
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.

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

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

7. If step 5 fails to locate a suitable `MessageBodyWriter` then generate a not acceptable response (HTTP 406 status).

### 4.3.3 Declaring Media Type Capabilities

Message body readers and writers MAY restrict the media types they support using the `@ConsumeMime` and `@ProduceMime` 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 */*. Quality parameter values are also used such that x/y;q=1.0 < x/y;q=0.7.

### 4.3.4 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 text media types (`text/*`).

**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.

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.

### 4.3.5 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.

**Editors Note 4.1** *Should JAX-RS require support for specific transfer encodings ?*

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

### 4.3.6 Content Encoding

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

## 4.4 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.

Section 5.1.6 describes how to access a context provider from a resource class or provider.

# 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 resource classes (see chapter 3) and providers (see chapter 4). This chapter describes these facilities.

## 5.1 Context Types

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

### 5.1.1 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:

```
@HttpMethod(GET)
@ProduceMime{"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();
}
```

### 5.1.2 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:

```
@HttpMethod(GET)
@ProduceMime{"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 response headers may be provided using the `Response` interface, see 3.2.3 for more details.

### 5.1.3 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:

```
@HttpMethod(PUT)
public Response updateFoo(@Context Request request, Foo foo) {
    EntityTag tag = getCurrentTag();
    ResponseBuilder responseBuilder = request.evaluate(tag, null);
    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.

### 5.1.4 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.

### 5.1.5 Message Body Workers

The `MessageBodyWorkers` interface allows for lookup of `MessageBodyReader` and `MessageBodyWriter` instances based on a set of search criteria including support media and Java type. An instance of `MessageBodyWorkers` can be injected into a class field or method parameter using the `@Context` annotation.

This interface is expected to be primarily of interest to entity provider authors wishing to use other entity providers to process a composite entity.

### 5.1.6 Context Resolver

Section 4.4 describes how an application can supply a `ContextResolver` for a particular context type. An instance of `ContextResolver` can be injected into a class field or method parameter using the `@Context` annotation.

The generic type of the annotation target is used select providers of the desired context type. The injected instance is not an instance of an application supplied context provider, rather it is a proxy that iterates through all the providers of the desired context type until one returns a non-`null` context for the type supplied in `getContext`. If no providers return a non-`null` context then the `getContext` method returns `null`.

## 5.2 Injection Scope

When the `@Context` annotation is applied to a class field, an implementation is only required to inject the applicable context into those class instances created by the implementation runtime. Objects returned by sub-resource locators (see section 3.3.1) are expected to be initialized by their creator and are not subject to resource injection by the implementation runtime.

# Chapter 6

# Environment

The container-managed resources available to a JAX-RS resource class or provider depend on the environment in which it is deployed. Section 5.1 describes the types of context available 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.

## 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

**Editors Note 6.1** *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.*

## 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 5.1.

# 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`, 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.

A JAX-RS implementation may rely on a particular implementation of `RuntimeDelegate` being used – overriding the supplied `RuntimeDelegate` instance with an application-supplied alternative is not recommended and may cause unexpected problems.

## 7.1 Configuration

If not supplied by injection, the `RuntimeDelegate` implementation class is determined 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.xml.ws.spi.Provider` 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 |
|---|---|---|
| `ConsumeMime` | Type or method | Specifies a list of media types that can be consumed. |
| `ProduceMime` | Type or method | Specifies a list of media types that can be consumed. |
| `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 3.2.5. |
| `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 | Specifies that the value of a method parameter is to be extracted from the request URI path. The value of the annotation identifies the name of a URI template parameter. |
| `QueryParam` | Parameter | Specifies that the value of a method parameter is to be extracted from a URI query parameter. The value of the annotation identifies the name of a query parameter. |
| `MatrixParam` | Parameter | Specifies that the value of a method parameter is to be extracted from a URI matrix parameter. The value of the annotation identifies the name of a matrix parameter. |
| `CookieParam` | Parameter | Specifies that the value of a method parameter is to be extracted from a HTTP cookie. The value of the annotation identifies the name of a the cookie. |
| `HeaderParam` | Parameter | Specifies that the value of a method parameter is to be extracted from a HTTP header. The value of the annotation identifies the name of a HTTP header. |

| Annotation | Target | Description |
|---|---|---|
| `Encoded` | Type, constructor, method or parameter | Disables automatic URI decoding for path, query and matrix parameters. |
| `DefaultValue` | Parameter | Specifies a default value for a method parameter annotated with `@QueryParam`, `@MatrixParam`, `@CookieParam` or `@HeaderParam`. The specified value will be used if the corresponding query or matrix parameter is not present in the request URI, or if the corresponding HTTP header is not included in the request. |
| `Context` | Field or parameter | Identifies an injection target for one of the types listed in section 5.1 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

# 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.