JAX-RS: Java™ API for RESTful Web Services

Version 2.1 Proposed Final Draft
June 15, 2017

Editors:
Pavel Bucek
Santiago Pericas-Geertsen

Comments to: jaxrs-spec@javaee.groups.io

Oracle Corporation
500 Oracle Parkway, Redwood Shores, CA 94065 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 the final release of version 2.1. The issue tracking system for this release can be found at:

https://github.com/jax-rs/api/issues

The corresponding Javadocs can be found online at:

https://jax-rs.github.io/apidocs/2.1

The reference implementation can be obtained from:

https://jersey.github.io

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

jaxrs-spec@javaee.groups.io

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 Java SE 8 The API will make extensive use of annotations and lambda expressions that require Java SE 8 or later.

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

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].

Figure 1.1: Example Java Code

```
package com.example.hello;

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

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

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.

Filter A provider used for filtering requests and responses.

Entity Interceptor A provider used for intercepting calls to message body readers and writers.

Invocation A Client API object that can be configured to issue an HTTP request.

WebTarget The recipient of an Invocation, identified by a URI.

Link A URI with additional meta-data such as a media type, a relation, a title, etc.

1.6 Expert Group Members

This specification is being developed as part of JSR 370 under the Java Community Process. It is the result of the collaborative work of the members of the JSR 370 Expert Group. The following are the present expert group members:

– Sergey Beryozkin (Talend SA)
– Adam Bien (Individual Member)
– Sebastian Dashner (Individual Member)
– Markus Karg (Individual Member)
– Casey Lee (Vision Service Plan)
– Marcos Luna (Individual Member)
– Andy McCright (IBM)
– Julian Reschke (Individual Member)
– Alessio Soldano (Red Hat)

The following are former group members of the JSR 339 Expert Group:

– Jan Algermussen (Individual Member)
– Florent Beniot (OW2)
– Sergey Beryozkin (Talend SA)
– Adam Bien (Individual Member)
– Bill Burke (Red Hat)
– Clinton L. Combs (Individual Member)
– Jian Wu Dai (IBM)
– Bill De Hora (Individual Member)
– Markus Karg (Individual Member)
– Sastry Malladi (eBay, Inc)
– Julian Reschke (Individual Member)
– Guilherme de Azevedo Silveira (Individual Member)
– Synodinos, Dionysios G. (Individual Member)

JAX-RS 1.X has been developed as part of JSR 311 under the Java Community Process. The following were group members of the JSR 311 Expert Group:

– Heiko Braun (Red Hat Middleware LLC)
– Larry Cable (BEA Systems)
– 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 this JSR we received great contributions and suggestions from many individuals. Special thanks to Marek Potociar and Michal Gajdos. Also Gunnar Morling, Ondrej Mihalyi, Arjan Tijms, Guillermo Gonzales de Aguero, Christian Kaltepoth and many, many others. Last but not least to the JSR 370 Expert Group for all their contributions.

During the course of JSR 339 we received many excellent suggestions. Special thanks to Martin Matula, Gerard Davison, Jakub Podlesak and Pavel Bucek from Oracle as well as Pete Muir and Emmanuel Bernard from Red Hat. Also to Gunnar Morling and Ron Sigal (Red Hat) for their suggestions on how to improve resource validation, and to Mattias Arthursson for his insights on hypermedia.

During the course of JSR 311 we received many excellent suggestions on the JSR and Jersey (RI) mailing lists, thanks in particular to James Manger (Telstra) and Reto Bachmann-Gmür (Trialox) for their contributions. The following individuals (all Sun Microsystems at the time) have also made invaluable technical contributions: Roberto Chinnici, Dianne Jiao (TCK), Ron Monzillo, Rajiv Mordani, Eduardo Pelegri-Llopart, Jakub Podlesak (RI) and Bill Shannon.

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.

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 Verification

Specific application requirements are detailed throughout this specification and the JAX-RS Javadocs. Implementations MAY perform verification steps that go beyond what it is stated in this document.

A JAX-RS implementation MAY report an error condition if it detects that two or more resources could result in an ambiguity during the execution of the algorithm described Section 3.7.2. For example, if two resource methods in the same resource class have identical (or even intersecting) values in all the annotations that are relevant to the algorithm described in that section. The exact set of verification steps as well as the error reporting mechanism is implementation dependent.

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 Web application in a `.war` file. The application classes are packaged in `WEB-INF/classes` or `WEB-INF/lib` and required libraries are packaged in `WEB-INF/lib`. See the Servlet specification for full details on packaging of web applications.

It is RECOMMENDED that implementations support the Servlet 3 framework pluggability mechanism to enable portability between containers and to avail themselves of container-supplied class scanning facilities. When using the pluggability mechanism the following conditions MUST be met:

- If *no* `Application` subclass is present, JAX-RS implementations are REQUIRED to dynamically add a servlet and set its name to

  ```
  javax.ws.rs.core.Application
  ```

  and to automatically discover all root resource classes and providers which MUST be packaged with the application. Additionally, the application MUST be packaged with a `web.xml` that specifies a servlet mapping for the added servlet. An example of such a `web.xml` file is:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
    </servlet>
    <servlet-mapping>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
        <url-pattern>/myresources/*</url-pattern>
    </servlet-mapping>
</web-app>
```

- If an `Application` subclass is present:
  - If there is already a servlet that handles this application. That is, a servlet that has an initialization parameter named

    ```
    javax.ws.rs.Application
    ```

    whose value is the fully qualified name of the `Application` subclass, then no additional configuration steps are required by the JAX-RS implementation.

– If *no* servlet handles this application, JAX-RS implementations are REQUIRED to dynamically add a servlet whose fully qualified name must be that of the `Application` subclass. If the `Application` subclass is annotated with `@ApplicationPath`, implementations are REQUIRED to use the value of this annotation appended with "/*" to define a mapping for the added server. Otherwise, the application MUST be packaged with a `web.xml` that specifies a servlet mapping. For example, if `org.example.MyApplication` is the name of the `Application` subclass, a sample `web.xml` would be:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
        <servlet-name>org.example.MyApplication</servlet-name>
    </servlet>
    <servlet-mapping>
        <servlet-name>org.example.MyApplication</servlet-name>
        <url-pattern>/myresources/*</url-pattern>
    </servlet-mapping>
</web-app>
```

When an `Application` subclass is present in the archive, if both `Application.getClasses` and `Application.getSingletons` return an empty collection then all root resource classes and providers packaged in the web application MUST be included and the JAX-RS implementation is REQUIRED to discover them automatically by scanning a `.war` file as described above. If either `getClasses` or `getSingletons` returns a non-empty collection then only those classes or singletons returned MUST be included in the published JAX-RS application.

The following table summarizes the Servlet 3 framework pluggability mechanism:

Condition	**Action**	**Servlet Name**	**`web.xml`**
No `Application` subclass	Add servlet	`javax.ws.rs.core.Application`	Required for servlet mapping
`Application` subclass handled by existing servlet	(none)	(already defined)	Not required
`Application` subclass *not* handled by existing servlet	Add servlet	Subclass name	If no `@ApplicationPath` then required for servlet mapping

Table 2.1: Summary of Servlet 3 framework pluggability cases

If not using the Servlet 3 framework pluggability mechanism (e.g. in a pre-Servlet 3.0 container), the `servlet-class` or `filter-class` element of the `web.xml` descriptor SHOULD name the JAX-RS implementation-supplied servlet or filter class respectively. The `Application` subclass SHOULD be identified using an `init-param` with a `param-name` of `javax.ws.rs.Application`.

Note that the Servlet 3 framework pluggability mechanism described above is based on servlets and not filters. Applications that prefer to use an implementation-supplied filter class must use the pre-Servlet 3.0 configuration mechanism.

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 10 and 11 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.

A JAX-RS implementation is only required to set the annotated field and bean property values of instances created by its runtime. Objects returned by sub-resource locators (see Section 3.4.1) are expected to be initialized by their creator.

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. Types for which a `ParamConverter` is available via a registered `ParamConverterProvider`. See Javadoc for these classes for more information.
2. Primitive types.
3. Types that have a constructor that accepts a single `String` argument.
4. Types that have a static method named `valueOf` or `fromString` with a single `String` argument that return an instance of the type. If both methods are present then `valueOf` MUST be used unless the type is an enum in which case `fromString` MUST be used[1].
5. `List<T>`, `Set<T>`, or `SortedSet<T>`, where *T* satisfies 1, 3 or 4 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 any of the 5 steps listed above is processed directly as described in Section 3.3.4. Other exceptions thrown during construction of field or property values using any of the 5 steps listed above are treated as client errors: if the field or property is annotated with `@MatrixParam`, `@QueryParam` or `@PathParam` then an implementation

[1] Due to limitations of the built-in `valueOf` method that is part of all Java enumerations, a `fromString` method is often defined by the enum writers. Consequently, the `fromString` method is preferred when available.

MUST generate an instance of `NotFoundException` (404 status) that wraps the thrown exception and no entity; if the field or property is annotated with `@HeaderParam` or `@CookieParam` then an implementation MUST generate an instance of `BadRequestException` (400 status) that wraps the thrown exception and no entity. Exceptions MUST 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`, `@PATCH`, `@HEAD` and `@OPTIONS`. 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. Exceptions thrown during construction of `@FormParam` annotated parameter values are treated the same as if the parameter were annotated with `@HeaderParam`.

3.3.2.1 Entity Parameters

The value of a parameter not annotated with `@FormParam` or any of the annotations listed in in Section 3.2, 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 have at most one entity parameter.

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 or of its type parameter T if the return type is CompletionStage<T> (see Section 8.2.2); if the class is an anonymous inner class, its superclass is used instead. 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 MessageBodyWriter as follows:

Return Type	Returned Instance[2]	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 in the following order:

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

1. Instances of `WebApplicationException` and its subclasses 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` or the corresponding subclass, 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 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 that have not been mapped MUST be re-thrown and allowed to propagate to the underlying container.

4. Checked exceptions and throwables that have not been mapped and 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 combination of the deployment context and the application path (see the `@ApplicationPath` annotation).

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`.

The value of a URI path parameter is available for injection via `@PathParam` on a field, property or method parameter. Note that if a URI template is used on a method, a path parameter injected in a field or property may not be available (set to `null`). The following example illustrates this scenario:

```
@Path("widgets")
public class WidgetsResource {
  @PathParam("id") String id;

  @GET
  public WidgetList getWidgets() {
    ...     // id is null here
  }

  @GET
  @Path("{id}")
  public Widget findWidget() {
    return new WidgetResource(id);
  }
}
```

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 or class 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[3].

Absent Such methods, known as *sub-resource locators*, are used to dynamically resolve the object that will handle the request. Sub-resource locators can return objects or classes; if a class is returned then an object is obtained by the implementation using a *suitable* constructor as described in Section 3.1.2. In either case, the resulting object is used to handle the request or to further resolve the object that will handle the request, see 3.7 for further details.

When an object is returned, implementations MUST dynamically determine its class 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
  @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] If the resource class URI template does not end with a '/' character then one is added during the concatenation.

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 MessageBodyReader).

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.

When accepting multiple media types, clients may indicate preferences by using a relative quality factor known as the q parameter. The value of the q parameter, or q-value, is used to sort the set of accepted types. For example, a client may indicate preference for `application/widgets+xml` with a relative quality factor of 1 and for `application/xml` with a relative quality factor of 0.8. Q-values range from 0 (undesirable) to 1 (highly desirable), with 1 used as default when omitted. A `GET` request matched to the `WidgetsResource` class with an accept header of `text/html; q=1, application/widgets+xml; q=0.8` will result in a call to method `getAsHtml` instead of `getAsXML` based on the value of q.

A server can also indicate media type preference using the qs parameter; server preference is only examined when multiple media types are accepted by a client *with the same q-value*. Consider the following example:

```
@Path("widgets2")
public class WidgetsResource2 {

  @GET
  @Produces("application/xml", "application/json")
  public Widgets getWidget() {...}

}
```

Suppose a client issues a `GET` request with an accept header of `application/*; q=0.5, text/html`. Based on this request, the server determines that both `application/xml` and `application/json` are equally preferred by the client with a q-value of 0.5. By specifying a server relative quality factor as part of the `@Produces` annotation, it is possible to control which response media type to select:

```
@Path("widgets2")
public class WidgetsResource2 {

  @GET
  @Produces("application/xml; qs=1", "application/json; qs=0.75")
  public Widgets getWidget() {...}

}
```

With the updated value for `@Produces` in this example, and in response to a `GET` request with an accept header that includes `application/*; q=0.5`, JAX-RS implementations are REQUIRED to select the media type `application/xml` given its higher qs-value. Note that qs-values, just like q-values, are relative and as such are only comparable to other qs-values within the same `@Produces` annotation instance. For more information see Section 3.8.

3.6 Annotation Inheritance

JAX-RS annotations may be used on the methods and method parameters of a super-class or an implemented interface. Such annotations are inherited by a corresponding sub-class or implementation class method provided that the method and its parameters do not have any JAX-RS annotations of their own. Annotations on a super-class take precedence over those on an implemented interface. The precedence over conflicting annotations defined in multiple implemented interfaces is implementation specific. Note that inheritance of class or interface annotations is not supported.

If a subclass or implementation method has any JAX-RS annotations then *all* of the annotations on the superclass 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.

For consistency with other Java EE specifications, it is recommended to always repeat annotations instead of relying on annotation inheritance.

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[4] 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. All exceptions reported by this algorithm MUST be processed as described in Section 3.3.4.

Matching of requests to resource methods proceeds in three stages as follows:

[4]Note: some containers might perform this functionality prior to passing the request to an implementation.

1. Identify a set of candidate root resource classes matching the request:

   Input $U =$ request URI path, $C = \{$root resource classes$\}$

   Output $U =$ final capturing group not yet matched, $C' = \{$root resouce classes matched so far$\}$

   (a) Set $E = \{\}$.

   (b) For each class Z in C add a regular expression (computed using the function $R(A)$ described in Section 3.7.3) to E as follows:

   - Add $R(T_Z)$ where T_Z is the URI path template specified for the class Z.

   Note that two or more classes in C may add the same regular expression to E if they are annotated with the same URI path template (modulo variable names).

   (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 Z has 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 `NotFoundException` (404 status) and no entity.

   (e) Sort E using the number of literal characters[5] 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 '([^/]+?)') as the tertiary key (descending order).

   (f) Set R_{match} to be the first member of E and set U to be the value of the final capturing group of R_{match} when matched against U. Let C' be the set of classes Z such that $R(T_Z) = R_{\text{match}}$. By definition, all root resource classes in C' must be annotated with the same URI path template modulo variable names.

2. Obtain a set of candidate resource methods for the request:

   Input $U =$ final capturing group not yet matched, $C' = \{$root resouce classes matched so far$\}$

   Output $M = \{$candidate resource methods$\}$

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

   $$M = \{\text{resource methods of all classes in } C' \text{ (excluding sub-resource methods)}\}$$

   and go to step 3 if $M \neq \{\}$

   (b) Set $E = \{\}$.

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

   i. For each sub-resource method D, add $R(T_D)$ where T_D is the URI path template of the sub-resource method.

   ii. For each sub-resource locator L, add $R(T_L)$ where T_L is the URI path template of the sub-resource locator.

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

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

- Remove members that do not match U.
- Remove members derived from T_D (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 and the algorithm terminates by generating a `NotFoundException` (404 status) and no entity.

(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 sub-resource methods ahead of those derived from sub-resource locators.

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

(h) Set M as follows,

$$M = \{\text{subresource methods of all classes in } C' \text{ (excluding sub-resource locators)}\}$$

and go to step 3 if $M \neq \{\}$.

(i) Let L be a sub-resource locator such that $R_{\text{match}} = R(T_L)$. Implementations SHOULD report an error if there is more than one sub-resource locator that satisfies this condition. Set U to be the value of the final capturing group of $R(T_L)$ when matched against U, and set C' to be the singleton set containing only the class that defines L.

(j) Go to step 2a.

3. Identify the method that will handle the request:

Input M = candidate resource methods

Output O = instance of resource class matched, D = resource method matched from M

(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 `NotAllowedException` (405 status) and no entity. 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 `NotSupportedException` (415 status) and no entity.
- 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 `NotAcceptableException` (406 status) and no entity.

(b) If after filtering the set M has more than one element, sort it in descending order as follows. First, let us define the *client* media type and the *server* media type as those denoted by the `Accept` header in a request and the `@Produces` annotation on a resource method, respectively. Let a client media type be of the form n/m;q=v_1, a server media type be of the form n/m;qs=v_2 and a *combined* media type of the form n/m;q=v_1;qs=v_2;d=v_3, where the distance factor d is defined below. For any of these types, m could be $*$, or m and n could be $*$ and the values of q and qs are assumed to be 1.0 if absent.

Let $S(p_1, p_2)$ be defined over a client media type p_1 and a server media type p_2 as the function that returns the *most* specific combined type with a distance factor if p_1 and p_2 are compatible and $\perp$ otherwise. For example:

- S(text/html;q=1, text/html;qs=1) = text/html;q=1;qs=1;d=0,
- S(text/*;q=0.5, text/html;qs=0.8) = text/html;q=0.5;qs=0.8;d=1,
- S(*/*;q=0.2, text/*;qs=0.9) = text/*;q=0.2;qs=0.9;d=1,
- S(text/*;q=0.4, application/*;qs=0.3) $=\perp$.

where the d factor corresponds to the number of wildcards matched with a concrete type or subtype. Note that q and qs are not matched, but simply combined in the resulting media type. A total ordering can be defined over combined media types as follows.

We write n_1/m_1;q=v_1;qs=v_1';d=v_1'' $\geq$ n_2/m_2;q=v_2;qs=v_2';d=v_2'' if one of these ordered conditions holds:

i. $n_1/m_1 \succ n_2/m_2$ where the partial order $\succ$ is defined as $n/m \succ n/* \succ */*$,
ii. $n_2/m_2 \not\succ n_1/m_1$ and $v_1 > v_2$,
iii. $n_2/m_2 \not\succ n_1/m_1$ and $v_1 = v_2$ and $v_1' > v_2'$.
iv. $n_2/m_2 \not\succ n_1/m_1$ and $v_1 = v_2$ and $v_1' = v_2'$ and $v_1'' \leq v_2''$.

Note that $\geq$ is a total order even though $\succ$ is a partial order. For example, the following holds text/html;q=1.0;qs=0.7;d=0 $\geq$ application/xml;q=1.0;qs=0.2;d=0 even though text/html is incomparable to application/xml according to $\succ$. Additionally, it is possible under $\geq$ for two types to be equal even though they are not identical [6]. For convenience, we defined $p \geq \perp$ for every media type p.

Given these definitions, we can now sort M in descending order based on $\geq$ as follows[7]:

- Let t be the request content type and C_M a resource method's `@Consumes` set of server media types, we use the media type $\max_{\geq}\{S(t,c) \,|\, (t,c) \in \{t\} \times C_M\}$ as primary key.
- Let A be the request accept header set of client media types and P_M a resource method's `@Produces` set of server media types, we use the media type $\max_{\geq}\{S(a,p) \,|\, (a,p) \in A \times P_M\}$ as secondary key. If there is more than one maximum element [6], implementations SHOULD report a warning and select one of these types in an implementation dependent manner.

(c) Let D be the first resource method in the set M[8] and O an instance of the class that defines D. If after sorting, there is more than one maximum element in M, implementations SHOULD report a warning and select one of these methods in an implementation dependent manner.

Consider the following example and assume the request to be `GET widgets/1`:

```
@Path("widget")
public class WidgetResource {
    private String id;

    public WidgetResource() {
        this("0");
    }

    public WidgetResource(String id) {
        this.id = id;
    }

```

[6] E.g., text/html;q=1.0;qs=0.7;d=0 $\geq$ application/xml;q=1.0;qs=0.7;d=0 and application/xml;q=1.0;qs=0.7;d=0 $\geq$ text/html;q=1.0;qs=0.7;d=0.

[7] If any of these types or sets of types are unspecified, */* and {*/*} are assumed.

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

```
    @GET
    public Widget findWidget() {
        return Widget.findWidgetById(id);
    }
}

@Path("widgets")
public class WidgetsResource {

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

The input and output values for each of the 3 steps in the matching algorithm are as follows:

Step 1 Identify a set of candidate root resource classes matching the request. Let $R(\text{widgets}) = \text{widgets(/.*)?}$ and $R(\text{widget}) = \text{widget(/.*)?}$,

Input $U = \text{widgets/1}$ and $C = \{\text{WidgetResource}, \text{WidgetsResource}\}$

Output $U = \text{/1}$ and $C' = \{\text{WidgetsResource}\}$

Step 2 Obtain a set of candidate resource methods for the request. Let $R(\{\text{id}\}) = \text{([^/]+?)(/.*)?}$,

Input $U = \text{/1}$ and $C' = \{\text{WidgetsResource}\}$

Output $M = \{\text{findWidget}\}$

Step 3 Identify the method that will handle the request,

Input $M = \{\text{findWidget}\}$

Output $O = \text{WidgetResource instance}$ and $D = \text{findWidget}$

Note that the algorithm matches a single root resource class (WidgetsResource) and, consequently, the `@Path` annotation on WidgetResource is ignored for the request `GET widgets/1`.

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[9].
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.getPathParameters`.

[9] Note that the syntax +? denotes a reluctant quantifier as defined in the java.util.regex.Pattern class.

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:

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 and server-side qs-value of p.
6. If $M = \{\}$ then generate a `NotAcceptableException` (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/* $>$ */*), a secondary key of q-value and a tertiary key of qs-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 `NotAcceptableException` (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

Providers in JAX-RS are responsible for various cross-cutting concerns such as filtering requests, converting representations into Java objects, mapping exceptions to responses, etc. A provider can be either *pre-packaged* in the JAX-RS runtime or supplied by an application. All *application-supplied* providers implement interfaces in the JAX-RS API and MAY be annotated with `@Provider` for automatic discovery purposes; the integration of pre-packaged providers into the JAX-RS runtime is implementation dependent.

This chapter introduces some of the basic JAX-RS providers; other providers are introduced in Chapter 5 and Chapter 6.

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 4.1.2) is called, then any requested dependencies are injected (see Chapter 10), then the appropriate provider methods may be called multiple times (simultaneously), and finally the object is made available for garbage collection. Section 10.2.6 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 Automatic Discovery

The annotation `@Provider` is used by a JAX-RS runtime to automatically discover provider classes via mechanisms such as class scanning. A JAX-RS implementation that supports automatic discovery of classes MUST process only those classes that are annotated with `@Provider`.

4.1.2 Constructors

Provider classes that 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 10 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.

4.1.3 Priorities

Application-supplied providers enable developers to extend and customize the JAX-RS runtime. Therefore, an application-supplied provider MUST always be preferred over a pre-packaged one if a single one is required.

Application-supplied providers may be annotated with `@Priority`. If two or more providers are candidates for a certain task, the one with the *highest* priority is chosen: the highest priority is defined to be the one with the *lowest* value in this case. That is, `@Priority(1)` is higher than `@Priority(10)`. If two or more providers are eligible and have identical priorities, one is chosen in an implementation dependent manner. The default priority for all application-supplied providers is `javax.ws.rs.Priorities.USER`.

The general rule about priorities is different for filters and interceptors since these providers are collected into chains. For more information see Section 6.6.

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.

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 may be annotated with `@Provider` for automatic discovery.

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

1. Obtain the media type of the request. If the request does not contain a `Content-Type` header then use `application/octet-stream`.
2. 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.
3. Select the set of `MessageBodyReader` classes that support the media type of the request, see Section 4.2.3.
4. Iterate through the selected `MessageBodyReader` classes and, utilizing the `isReadable` method of each, choose a `MessageBodyReader` provider that supports the desired Java type.

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

5. If step 4 locates one or more suitable `MessageBodyReader`'s then select the one with the highest priority as described in Section 4.1.3 and use its `readFrom` method to map the entity body to the desired Java type.

6. Otherwise, the server runtime MUST generate a `NotSupportedException` (415 status) and no entity (to be processed as described in Section 3.3.4) and the client runtime MUST generate an instance of `ProcessingException`.

See Section 4.5 for more information on handling exceptions thrown in `MessageBodyReader.readFrom`.

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 may be annotated with `@Provider` for automatic discovery.

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

1. Obtain the object that will be mapped to the message 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.

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

4. Sort the selected `MessageBodyWriter` providers with a primary key of generic type where providers whose generic type is the nearest superclass of the object class are sorted first and a secondary key of media type (see 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 one or more suitable `MessageBodyWriter`'s that are equal with respect to the sorting in step 4, then select the one with the highest priority as described in Section 4.1.3 and use its `writeTo` method to map the entity body to the desired Java type.

7. Otherwise, the server runtime MUST generate a generate an `InternalServerErrorException`, a subclass of `WebApplicationException` with its status set to 500, and no entity (to be processed as described in Section 3.3.4) and the client runtime MUST generate a `ProcessingException`.

Experience gained in the field has resulted in the reversal of the sorting keys in step 4 in this specification. This represents a backward incompatible change with respect to JAX-RS 1.X. Implementations of this specification are REQUIRED to provide a backward compatible flag for those applications that rely on the previous ordering. The mechanism defined to enable this flag is implementation dependent.

See Section 4.5 for more information on handling exceptions thrown in `MessageBodyWriter.write`.

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 media types of the form `application/*+xml`).

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

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

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

`java.lang.Boolean`, `java.lang.Character`, `java.lang.Number` Only for `text/plain`. Corresponding primitive types supported via boxing/unboxing conversion.

Depending on the environment, the list of standard entity providers MUST also include those for JSON. For more information about these providers see Sections 11.2.6 and 11.2.7.

When reading zero-length message entities all pre-packaged `MessageBodyReader` implementations, except the JAXB one and those for the (boxed) primitive types above, MUST create a corresponding Java object that represents zero-length data. The pre-packaged JAXB and the pre-packaged primitive type `MessageBodyReader` implementations MUST throw a `NoContentException` for zero-length message entities.

When a `NoContentException` is thrown while reading a server request entity from a `MessageBodyReader` it MUST be translated by the server runtime into a `BadRequestException` wrapping the original `NoContentException` and re-thrown to be processed by any registered exception mappers.

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. More precisely, step 4 in Section 4.2.1 and step 5 in Section 4.2.2 MUST prefer application-provided over pre-packaged entity providers.

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 may be annotated with `@Provider` for automatic discovery. 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

Exception mapping providers map a checked or runtime exception to an instance of `Response`. An exception mapping provider implements the `ExceptionMapper<T>` interface and may be annotated with `@Provider` for automatic discovery.

When a resource class or provider 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 a web resource method had returned the `Response`, see Section 3.3.3. In particular, a mapped `Response` MUST be processed using the ContainerResponse filter chain defined in Chapter 6.

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. If two or more exception providers are applicable, the one with the highest priority MUST be chosen as described in Section 4.1.3.

To avoid a potentially infinite loop, a single exception mapper must be used during the processing of a request and its corresponding response. JAX-RS implementations MUST NOT attempt to map exceptions thrown while processing a response previously mapped from an exception. Instead, this exception MUST be processed as described in steps 3 and 4 in Section 3.3.4.

Note that exception mapping providers are *not* supported as part of the Client API.

4.5 Exceptions

Exception handling differs depending on whether a provider is part of the client runtime or server runtime. This is covered in the next two sections.

4.5.1 Server Runtime

When a provider method throws an exception, the JAX-RS server runtime will attempt to map the exception to a suitable HTTP response in the same way as described for methods and locators in Section 3.3.4. If the exception is thrown while generating a response, JAX-RS implementations are required to map the exception *only when* the response has not been committed yet.

As explained in Section 4.4, an application can supply exception mapping providers to customize this mapping, but these exception mappers will be ignored during the processing of a *previously mapped* response to avoid entering a potentially infinite loop. For example, suppose a method in a message body reader throws an exception that is mapped to a response via an exception mapping provider; if the message body writer throws an exception while trying to write the mapped response, JAX-RS implementations will not attempt to map the exception again.

4.5.2 Client Runtime

When a provider method throws an exception, the JAX-RS client runtime will map it to an instance of `ProcessingException` if thrown while processing a request, and to a `ResponseProcessingException` if thrown while processing a response.

Note that the client runtime will only throw an instance of `WebApplicationException` (or any of its subclasses) as a result of a response from the server with status codes 3xx, 4xx or 5xx.

Chapter 5

Client API

The Client API is used to access Web resources. It provides a higher-level API than `HttpURLConnection` as well as integration with JAX-RS providers. Unless otherwise stated, types presented in this chapter live in the `javax.ws.rs.client` package.

5.1 Bootstrapping a Client Instance

An instance of `Client` is required to access a Web resource using the Client API. The default instance of `Client` can be obtained by calling `newClient` on `ClientBuilder`. `Client` instances can be configured using methods inherited from `Configurable` as follows:

```
// Default instance of client
Client client = ClientBuilder.newClient();

// Additional configuration of default client
client.property("MyProperty", "MyValue")
      .register(MyProvider.class)
      .register(MyFeature.class);
```

See Chapter 4 for more information on providers. Properties are simply name-value pairs where the value is an arbitrary object. Features are also providers and must implement the `Feature` interface; they are useful for grouping sets of properties and providers (including other features) that are logically related and must be enabled as a unit.

5.2 Resource Access

A Web resource can be accessed using a fluent API in which method invocations are chained to build and ultimately submit an HTTP request. The following example gets a `text/plain` representation of the resource identified by `http://example.org/hello`:

```
Client client = ClientBuilder.newClient();
Response res = client.target("http://example.org/hello")
    .request("text/plain").get();
```

Conceptually, the steps required to submit a request are the following: (i) obtain an instance of `Client` (ii) create a `WebTarget` (iii) create a request from the `WebTarget` and (iv) submit a request or get a prepared `Invocation` for later submission. See Section 5.5 for more information on using `Invocation`.

Method chaining is not limited to the example shown above. A request can be further specified by setting headers, cookies, query parameters, etc. For example:

```
Response res = client.target("http://example.org/hello")
     .queryParam("MyParam","...")
     .request("text/plain")
     .header("MyHeader", "...")
     .get();
```

See the Javadoc for the classes in the `javax.ws.rs.client` package for more information.

5.3 Client Targets

The benefits of using a `WebTarget` become apparent when building complex URIs, for example by extending base URIs with additional path segments or templates. The following example highlights these cases:

```
WebTarget base = client.target("http://example.org/");
WebTarget hello = base.path("hello").path("{whom}");
Response res = hello.resolveTemplate("whom", "world").request("...").get();
```

Note the use of the URI template parameter {`whom`}. The example above gets a representation for the resource identified by `http://example.org/hello/world`.

`WebTarget` instances are *immutable* with respect to their URI (or URI template): methods for specifying additional path segments and parameters return a new instance of `WebTarget`. However, `WebTarget` instances are *mutable* with respect to their configuration. Thus, configuring a `WebTarget` does not create new instances.

```
// Create WebTarget instance base
WebTarget base = client.target("http://example.org/");
// Create new WebTarget instance hello and configure it
WebTarget hello = base.path("hello");
hello.register(MyProvider.class);
```

In this example, two instances of `WebTarget` are created. The instance `hello` inherits the configuration from `base` and it is further configured by registering `MyProvider.class`. Note that changes to `hello`'s configuration do not affect `base`, i.e. inheritance performs a *deep* copy of the configuration. See Section 5.6 for additional information on configurable types.

5.4 Typed Entities

The response to a request is not limited to be of type `Response`. The following example upgrades the status of customer number 123 to "gold status" by first obtaining an entity of type `Customer` and then posting that entity to a different URI:

```
Customer c = client.target("http://examples.org/customers/123")
    .request("application/xml").get(Customer.class);
String newId = client.target("http://examples.org/gold-customers/")
    .request().post(xml(c), String.class);
```

Note the use of the *variant* `xml()` in the call to `post`. The class `javax.ws.rs.client.Entity` defines variants for the most popular media types used in JAX-RS applications.

In the example above, just like in the Server API, JAX-RS implementations are REQUIRED to use entity providers to map a representation of type `application/xml` to an instance of `Customer` and vice versa. See Section 4.2.4 for a list of entity providers that MUST be supported by all JAX-RS implementations.

5.5 Invocations

An invocation is a request that has been prepared and is ready for execution. Invocations provide a *generic interface* that enables a separation of concerns between the creator and the submitter. In particular, the submitter does not need to know how the invocation was prepared, but only how it should be executed: namely, synchronously or asynchronously.

Let us consider the following example[1]:

```
// Executed by the creator
Invocation inv1 = client.target("http://examples.org/atm/balance")
    .queryParam("card", "111122223333").queryParam("pin", "9876")
    .request("text/plain").buildGet();
Invocation inv2 = client.target("http://examples.org/atm/withdrawal")
    .queryParam("card", "111122223333").queryParam("pin", "9876")
    .request().buildPost(text("50.0"));
Collection<Invocation> invs = Arrays.asList(inv1, inv2);

// Executed by the submitter
Collection<Response> ress =
    Collections.transform(invs,
        new F<Invocation, Response>() {
             @Override
             public Response apply(Invocation inv) {
                 return inv.invoke(); } });
```

In this example, two invocations are prepared and stored in a collection by the creator. The submitter then traverses the collection applying a transformation that maps an `Invocation` to a `Response`. The mapping calls `Invocation.invoke` to execute the invocation synchronously; asynchronous execution is also supported by calling `Invocation.submit`. Refer to Chapter 8 for more information on asynchronous invocations.

5.6 Configurable Types

The following Client API types are configurable: `Client`, `ClientBuilder`, and `WebTarget`. Configuration methods are inherited from the `Configurable` interface implemented by all these classes. This

[1]The Collections class in this example is arbitrary and does not correspond to any specific implementation. There are a number of Java collection libraries available that provide this type of functionality.

interface supports configuration of:

Properties Name-value pairs for additional configuration of features or other components of a JAX-RS implementation.

Features A special type of provider that implement the Feature interface and can be used to configure a JAX-RS implementation.

Providers Classes or instances of classes that implement one or more of the provider interfaces from Chapter 4. A provider can be a message body reader, a filter, a context resolver, etc.

The configuration defined on an instance of any of the aforementioned types is inherited by other instances created from it. For example, an instance of WebTarget created from a Client will inherit the Client's configuration. However, any additional changes to the instance of WebTarget will not impact the Client's configuration and vice versa. Therefore, once a configuration is inherited it is also detached (deep copied) from its parent configuration and changes to the parent and child configurations are not be visible to each other.

5.6.1 Filters and Entity Interceptors

As explained in Chapter 6, filters and interceptors are defined as JAX-RS providers. Therefore, they can be registered in any of the configurable types listed in the previous section. The following example shows how to register filters and interceptors on instances of Client and WebTarget:

```
// Create client and register logging filter
Client client = ClientBuilder.newClient().register(LoggingFilter.class);

// Executes logging filter from client and caching filter from target
WebTarget wt = client.target("http://examples.org/customers/123");
Customer c = wt.register(CachingFilter.class).request("application/xml")
    .get(Customer.class);
```

In this example, LoggingFilter is inherited by each instance of WebTarget created from client; an additional provider named CachingFilter is registered on the instance of WebTarget.

5.7 Reactive Clients

Section 8.4 introduces asynchronous programming in the Client API. Asynchronous programming in JAX-RS enables clients to unblock certain threads by pushing work to background threads which can be monitored and possibly waited on (joined) at a later time. This can be accomplished in JAX-RS by either providing an instance of InvocationCallback or operating on the result of type Future<T> returned by an asynchronous invoker —or some combination of both styles.

Using InvocationCallback enables a more *reactive* programming style in which user-provided code activates (or reacts) only when a certain event has occurred. Using callbacks works well for simple cases, but the source code becomes harder to understand when multiple events are in play. For example, when asynchronous invocations need to be composed, combined or in any way operated upon. These type of scenarios may result in callbacks that are nested inside other callbacks making the code far less readable —often referred to as the *pyramid of doom* because of the inherent nesting of calls.

To address the requirement of greater readability and to enable programmers to better reason about asynchronous computations, Java 8 introduces the a new interface called CompletionStage that includes a large number of methods dedicated to managing asynchronous computations.

JAX-RS 2.1 defines a new type of invoker called RxInvoker, as well a default implementation of this type called CompletionStageRxInvoker that is based on the Java 8 type CompletionStage. There is a new rx method which is used in a similar manner to async as described in 8.4. Let us consider the following example:

```
CompletionStage<String> csf = client.target("forecast/{destination}")
    .resolveTemplate("destination", "mars")
    .request()
    .rx()
    .get(String.class);

csf.thenAccept(System.out::println);
```

This example first creates an asynchronous computation of type CompletionStage<String>, and then simply waits for it to complete and displays its result (technically, a second computation of type CompletionStage<Void> is created on the last line simply to consume the result of the first computation).

The value of CompletionStage becomes apparent when multiple asynchronous computations are necessary to accomplish a task. The following example obtains, in parallel, a price and a forecast for a destination and makes a reservation only if the desired conditions are met.

```
CompletionStage<Number> csp = client.target("price/{destination}")
    .resolveTemplate("destination", "mars")
    .request()
    .rx()
    .get(Number.class);

CompletionStage<String> csf = client.target("forecast/{destination}")
    .resolveTemplate("destination", "mars")
    .request()
    .rx()
    .get(String.class);

csp.thenCombine(csf, (price, forecast) ->
    reserveIfAffordableAndWarm(price, forecast));
```

Note that the Consumer passed in the call to method thenCombine requires the values of each stage to be available and, therefore, can only be executed after the two parallel stages are completed.

As we shall see in the next section, support for CompletionStage is the *default* for all JAX-RS implementations, but other reactive APIs may also be supported as extensions.

5.7.1 Reactive API Extensions

There have been several proposals for reactive APIs in Java. All JAX-RS implementations MUST support an invoker for CompletionStage as shown above. Additionally, JAX-RS implementations MAY support other reactive APIs using an extension built into the Client API.

RxJava [11] is a popular reactive library available in Java. The type representing an asynchronous computation in this API is called an `Observable`. An implementation may support this type by providing a new invoker as shown in the following example:

```
Client client = client.register(ObservableRxInvokerProvider.class);

Observable<String> of = client.target("forecast/{destination}")
    .resolveTemplate("destination", "mars")
    .request()
    .rx(ObservableRxInvoker.class)    // overrides default invoker
    .get(String.class);

of.subscribe(System.out::println);
```

First, a provider for the new invoker must be registered on the `Client` object. Second, the type of the invoker must be specified as a parameter to the `rx` method. Note that because this is a JAX-RS extension, the actual names for the provider and the invoker in the example above are implementation dependent. The reader is referred to the documentation for the JAX-RS implementation of choice for more information.

Version 2.0 of RxJava [12] has been completely re-written on top of the Reactive-Streams specification. This new architecture prompted the introduction of a new type called `Flowable`. JAX-RS implementations could easily support this new version by implementing a new provider (such as `FlowableRxInvokerProvider`) and using the same pattern shown in the example above.

5.8 Executor Services

Executor services can be used to submit asynchronous tasks for execution. JAX-RS applications can specify executor services while building a `Client` instance. Two methods are provided in `ClientBuilder` for this purpose, namely, `executorService` and `scheduledExecutorService`.

In an environment that supports the Concurrency Utilities for Java EE [13], such as the Java EE Full Profile, implementations MUST use `ManagedExecutorService` and `ManagedScheduledExecutorService`, respectively. The reader is referred to the Javadoc of `ClientBuilder` for more information about executor services.

Chapter 6

Filters and Interceptors

Filters and entity interceptors can be registered for execution at well-defined extension points in JAX-RS implementations. They are used to extend an implementation in order to provide capabilities such as logging, confidentiality, authentication, entity compression, etc.

6.1 Introduction

Entity interceptors wrap around a method invocation at a specific extension point. Filters execute code at an extension point but without wrapping a method invocation. There are four extension points for filters: ClientRequest, ClientResponse, ContainerRequest and ContainerResponse. There are two extension points for entity interceptors: ReadFrom and WriteTo. For each of these extension points, there is a corresponding interface:

```
public interface ClientRequestFilter {
    void filter(ClientRequestContext requestContext) throws IOException;
}
public interface ClientResponseFilter {
    void filter(ClientRequestContext requestContext,
        ClientResponseContext responseContext) throws IOException;
}
public interface ContainerRequestFilter {
    void filter(ContainerRequestContext requestContext) throws IOException;
}
public interface ContainerResponseFilter {
    void filter(ContainerRequestContext requestContext,
        ContainerResponseContext responseContext) throws IOException;
}
```

```
public interface ReaderInterceptor {
    Object aroundReadFrom(ReaderInterceptorContext context)
        throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
public interface WriterInterceptor {
    void aroundWriteTo(WriterInterceptorContext context)
        throws java.io.IOException, javax.ws.rs.WebApplicationException;
}
```

A *client* filter is a class that implements `ClientRequestFilter` or `ClientResponseFilter` or both. A *container* filter is a class that implements `ContainerRequestFilter` or `ContainerResponseFilter`, or both. An entity interceptor is a class that implements `ReaderInterceptor` or `WriterInterceptor`, or both. Filters and entity interceptors are providers and, as such, may be annotated with `@Provider` for automatic discovery.

In the Client API, a `ClientRequestFilter` is executed as part of the invocation pipeline, before the HTTP request is delivered to the network; a `ClientResponseFilter` is executed upon receiving a server response, before control is returned to the application. In the Server API, a `ContainerRequestFilter` is executed upon receiving a request from a client; a `ContainerResponseFilter` is executed as part of the response pipeline, before the HTTP response is delivered to the network.

A globally-bound (see Section 6.5.1) `ContainerRequestFilter` is a container filter executed after resource matching *unless* it is annotated with `@PreMatching`. The use of this annotation on this type of filters defines a new extension point for applications to use, namely PreMatchContainerRequest. Certain `ContainerRequestContext` methods may not be available at this extension point.

An entity interceptor implementing `ReaderInterceptor` wraps around calls to `MessageBodyReader`'s method `readFrom`. An entity interceptor implementing `WriterInterceptor` wraps around calls to `MessageBodyWriter`'s method `writeTo`. JAX-RS implementations are REQUIRED to call registered interceptors when mapping representations to Java types and vice versa. See Section 4.2 for more information on entity providers.

Please refer to Appendix C for some diagrams on the client and server processing pipelines that show the interaction between filters and entity interceptors.

6.2 Filters

Filters are grouped into *filter chains*. There is a separate filter chain for each extension point introduced in the previous section, namely: ClientRequest, ClientResponse, ContainerRequest, ContainerResponse and PreMatch-ContainerRequest. Filters in a chain are sorted based on their priorities (see Section 6.6) and are executed in order.

The following example shows an implementation of a container logging filter: each method simply logs the message and returns.

```
@Provider
class LoggingFilter implements ContainerRequestFilter,
                               ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext)
        throws IOException {
        log(requestContext);
    }

    @Override
    public void filter(ContainerRequestContext requestContext,
        ContainerResponseContext responseContext) throws IOException {
        log(responseContext);
    }
    ...
}
```

`ContainerRequestContext` is a mutable class that provides request-specific information for the filter, such as the request URI, message headers, message entity or request-scoped properties. The exposed setters allow (certain) modification of the request before it is processed by the resource method. Similarly, there is a corresponding `ContainerResponseContext` that provides response-specific information.

Request filters implementing `ClientRequestFilter` or `ContainerRequestFilter` can stop the execution of their corresponding chains by calling `abortWith(Response)` in their corresponding context object. If this method is invoked, JAX-RS implementations are REQUIRED to abort execution of the chain and treat the response object as if produced by calling the resource method (Server API) or executing the HTTP invocation (Client API). For example, upon a cache hit, a client *caching* filter may call `abortWith(Response)` to abort execution and optimize network access.

As stated above, a `ContainerRequestFilter` that is annotated with `@PreMatching` is executed upon receiving a client request but *before* a resource method is matched. Thus, this type of filter has the ability to modify the input to the matching algorithm (see Section 3.7.2) and, consequently, alter its outcome. The following example uses a `ContainerRequestFilter` annotated with `@PreMatching` to tunnel requests via POST by using the X-HTTP-Method-Override header to overwrite the HTTP method prior to resource matching.

```
@Provider
@PreMatching
public class HttpMethodOverrideFilter implements ContainerRequestFilter {

    @Override
    public void filter(ContainerRequestContext requestContext)
        throws IOException {
        if (requestContext.getMethod().equalsIgnoreCase("POST")) {
            String override = requestContext.getHeaders()
                .getFirst("X-HTTP-Method-Override");
            if (override != null) {
                requestContext.setMethod(override);
            }
        }
    }
}
```

6.3 Entity Interceptors

An entity interceptor implements interface `ReaderInterceptor` or `WriterInterceptor`, or both. There is an *interceptor chain* for each kind of entity interceptor. Entity interceptors in a chain are sorted based on their priorities (see Section 6.6) and are executed in order.

As part of the JAX-RS processing pipeline (see Appendix C), entity interceptors wrap calls to the methods `readFrom` in classes implementing `MessageBodyReader` and `writeTo` in classes implementing `MessageBodyWriter`. An interceptor SHOULD explicitly call the context method `proceed` to continue the execution of the chain. Because of their wrapping nature, failure to call this method will prevent execution of the wrapped method in the corresponding message body reader or message body writer.

The following example shows an implementation of a GZIP entity interceptor that provides deflate and inflate capabilities [1].

[1] This class is not intended to be a complete implementation of this interceptor.

```
@Provider
class GzipInterceptor implements ReaderInterceptor, WriterInterceptor {

    @Override
    Object aroundReadFrom(ReaderInterceptorContext ctx) ... {
        if (isGzipped(ctx)) {
            InputStream old = ctx.getInputStream();
            ctx.setInputStream(new GZIPInputStream(old));
            try {
                return ctx.proceed();
            } finally {
                ctx.setInputStream(old);
            }
        } else {
            return ctx.proceed();
        }
    }

    @Override
    void aroundWriteTo(WriterInterceptorContext ctx) ... {
        OutputStream old = ctx.getOutputStream();
        GZIPOutputStream gzipOutputStream = new GZIPOutputStream(old);
        ctx.setOutputStream(gzipOutputStream);
        updateHeaders(ctx);
        try {
            ctx.proceed();
        } finally {
            gzipOutputStream.finish();
            ctx.setOutputStream(old);
        }
    }
    ...
}
```

The context types, `ReaderInterceptorContext` and `WriterInterceptorContext`, provide read and write access to the parameters of the corresponding wrapped methods. In the example shown above, the input and output streams are wrapped and updated in the context object before proceeding. JAX-RS implementations MUST use the last parameter values set in the context object when calling the wrapped methods `MessageBodyReader.readFrom` and `MessageBodyWriter.writeTo`.

It is worth noting that a `readFrom` or a `writeTo` that is called directly from application code, e.g. via the injection of a `Providers` instance, will *not* trigger the execution of any entity interceptors since it is not part of the normal JAX-RS processing pipeline.

6.4 Lifecycle

By default, just like all the other providers, a single instance of each filter or entity interceptor is instantiated for each JAX-RS application. First the constructor is called, then any requested dependencies are injected, then the appropriate methods are called (simultaneously) as needed. Implementations MAY offer alternative lifecycle options beyond the default one. See Section 4.1 for additional information.

6.5 Binding

Binding is the process by which a filter or interceptor is associated with a resource class or method (Server API) or an invocation (Client API). The forms of binding presented in the next sections are only supported as part of the Server API. See Section 6.5.4 for binding in the Client API.

6.5.1 Global Binding

Global binding is the default type of binding. A filter or interceptor that has no annotations is assumed to be bound globally, i.e. it applies to all the resource methods in an application. Like any other provider, a filter or interceptor can be registered manually (e.g., via `Application` or `Configuration`) or be discovered automatically. Note that for a filter or interceptor to be automatically discovered it MUST be annotated with `@Provider` (see Section 4.1.1).

For example, the `LoggingFilter` defined in Section 6.2 is both automatically discovered (it is annotated with `@Provider`) and bound globally. If this filter is part of an application, requests and responses will be logged for all resource methods.

As stated in Section 6.1, a global `ContainerRequestFilter` is executed after resource matching unless annotated with `@PreMatching`. A global filter that injects `ResourceInfo`, and generally depends on resource information for its execution, must not be annotated with `@PreMatching`.

6.5.2 Name Binding

A filter or interceptor can be associated with a resource class or method by declaring a new *binding* annotation à la CDI [14]. These annotations are declared using the JAX-RS meta-annotation `@NameBinding` and are used to decorate both the filter (or interceptor) and the resource method or resource class. For example, the `LoggingFilter` defined in Section 6.2 can be bound to the method `hello` in `MyResourceClass`, instead of globally, as follows:

```
@Provider
@Logged
class LoggingFilter implements ContainerRequestFilter,
                               ContainerResponseFilter {
    ...
}
```

```
@Path("/")
public class MyResourceClass {
    @Logged
    @GET
    @Produces("text/plain")
    @Path("{name}")
    public String hello(@PathParam("name") String name) {
        return "Hello " + name;
    }
}
```

According to the semantics of `LoggingFilter`, the request will be logged before the `hello` method is called and the response will be logged after it returns. The declaration of the `@Logged` annotation is shown next.

```
@NameBinding
@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface Logged { }
```

Multiple filters and interceptors can be bound to a single resource method using additional annotations. For example, given the following filter:

```
@Provider
@Authenticated
class AuthenticationFilter implements ContainerRequestFilter {
    ...
}
```

method `hello` above could be decorated with `@Logged` and `@Authenticated` in order to provide both logging and authentication capabilities to the resource.

A filter or interceptor class can be decorated with multiple binding annotations. In this case, in accordance with the semantics described in CDI [14], all those annotations must be present in the resource class or method for the binding to be established. For example, if `LoggingFilter` is defined as follows:

```
@Provider
@Logged @Verbose
class LoggingFilter implements ContainerRequestFilter,
                               ContainerResponseFilter {
    ...
}
```

then method `hello` above must be annotated with both `@Logged` and `@Verbose` for the binding to be in effect.

Binding annotations can also be applied to resource classes and `Application` subclasses. Binding annotations that decorate resource classes apply to *all* resource methods defined in them. Binding annotations that decorate `Application` subclasses can also be used to bind filters and interceptors globally, even if other annotations are present. For example, `LoggingFilter` as defined at the beginning of this section can be bound globally as follows:

```
@Logged
public class MyApplication extends Application {
    ...
}
```

Note that returning filters or interceptors from the methods `getClasses` or `getSingletons` in an application subclass will bind them globally only if they are *not* decorated with a name binding annotation. If they are decorated with at least one name binding annotation, the application subclass must be annotated as shown above in order for those filters or interceptors to be bound globally. See Chapter 2 for more information on JAX-RS applications.

6.5.3 Dynamic Binding

The annotation-based forms of binding presented thus far are *static*. Dynamic binding is also supported using dynamic features. A dynamic feature is a provider that implements the `DynamicFeature` interface. These providers are used to augment the set of filters and entity interceptors bound to a resource method.

The following example defines a dynamic feature that binds the filter `LoggingFilter` —assumed not globally bound for the purpose of this example— with all the resource methods in `MyResource` that are annotated with `@GET`.

```
@Provider
public final class DynamicLoggingFilterFeature implements DynamicFeature {

  @Override
  public void configure(ResourceInfo resourceInfo,
                        FeatureContext context) {
    if (MyResource.class.isAssignableFrom(resourceInfo.getResourceClass())
      && resourceInfo.getResourceMethod().isAnnotationPresent(GET.class)) {
      context.register(new LoggingFilter());
    }
  }
}
```

The overridden method in this provider updates the `Configuration` object assigned to each resource method; the information about the resource method is provided in the form of a `ResourceInfo` instance. JAX-RS implementations SHOULD resolve dynamic features for filters and interceptors once for each resource method. It is RECOMMENDED to process dynamic features at application deployment time.

6.5.4 Binding in Client API

Binding in the Client API is accomplished via API calls instead of annotations. `Client`, `Invocation`, `Invocation.Builder` and `WebTarget` are all configurable types: their configuration can be accessed using the methods inherited from the `Configurable` interface. See Section 5.6 for more information.

6.6 Priorities

The order in which filters and interceptors are executed as part of their corresponding chains is controlled by the `@Priority` annotation defined in [15]. Priorities are represented by integer numbers. Execution chains for extension points ContainerRequest, PreMatchContainerRequest, ClientRequest, ReadFrom and WriteTo are sorted in *ascending order*; the lower the number the higher the priority. Execution chains for extension points ContainerResponse and ClientResponse are sorted in *descending order*; the higher the number the higher the priority. These rules ensure that response filters are executed in reversed order of request filters.

The `Priorities` class in JAX-RS defines a set of built-in priorities for security, header decorators, decoders and encoders. The default binding priority is `javax.ws.rs.Priorities.USER`. For example, the priority of an authentication filter can be set as follows:

```
@Provider
@Authenticated
@Priority(Priorities.AUTHENTICATION)
public class AuthenticationFilter implements ContainerRequestFilter {
    ...
}
```

Note that even though, as explained in Section 6.5.4, annotations are not used for binding in the Client API, they are still used to define priorities. Therefore, if a priority other than the default is required, the `@Priority` annotation must be used for a filter or interceptor registered with the Client API.

The order in which filters and interceptors that belong to the same priority class are executed is implementation dependent.

6.7 Exceptions

6.7.1 Server Runtime

When a filter or interceptor method throws an exception, the server runtime will process the exception as described in Section 4.5.1. As explained in Section 4.4, an application can supply exception mapping providers. At most one exception mapper MUST be used in a single request processing cycle to avoid potentially infinite loops.

A response mapped from an exception MUST be processed using the ContainerResponse filter chain and the WriteTo interceptor chain (if an entity is present in the mapped response). The number of entries in these chains depends on whether a resource method has been matched or not at the time the exception is thrown. There are two cases:

1. If a web resource has been matched before the exception was thrown, then the filters in ContainerResponse and the interceptors in WriteTo will include everything that has been bound to the method as well as globally;
2. Otherwise, only global filters and interceptors will be included.

Note that a filter or interceptor invoked in case 2 will not have access to resource-dependent information, such as that returned by an injectable instance of `ResourceInfo`.

6.7.2 Client Runtime

When a filter or interceptor method throws an exception, the client runtime will process the exception as described in Section 4.5.2.

Chapter 7

Validation

Validation is the process of verifying that some data obeys one or more pre-defined constraints. The Bean Validation specification [16] defines an API to validate Java Beans. This chapter describes how JAX-RS provides native support for validating resource classes based on the concepts presented in [16]. See Section 11.2.5 for more information on implementation requirements.

7.1 Constraint Annotations

The Server API provides support for extracting request values and mapping them into Java fields, properties and parameters using annotations such as `@HeaderParam`, `@QueryParam`, etc. It also supports mapping of request entity bodies into Java objects via non-annotated parameters (i.e., parameters without any JAX-RS annotations). See Chapter 3 for additional information.

In earlier versions of JAX-RS, any additional validation of these values needed to be performed programmatically. This version of JAX-RS introduces support for declarative validation based on the Bean Validation specification [16].

The Bean Validation specification [16] supports the use of *constraint annotations* as a way of declaratively validating beans, method parameters and method returned values. For example, consider the following resource class augmented with constraint annotations:

```
@Path("/")
class MyResourceClass {

  @POST
  @Consumes("application/x-www-form-urlencoded")
  public void registerUser(
    @NotNull @FormParam("firstName") String firstName,
    @NotNull @FormParam("lastName") String lastName,
    @Email @FormParam("email") String email) {
    ...
  }
}
```

The annotations `@NotNull` and `@Email` impose additional constraints on the form parameters `firstName`, `lastName` and `email`. The `@NotNull` constraint is built-in to the Bean Validation API; the `@Email` constraint is assumed to be user defined in the example above. These constraint annotations are not restricted to method parameters, they can be used in any location in which the JAX-RS binding annotations are allowed with the exception of constructors and property setters. Rather than using method parameters, the `MyResourceClass` shown above could have been written as follows:

```
@Path("/")
class MyResourceClass {

  @NotNull @FormParam("firstName")
  private String firstName;

  @NotNull @FormParam("lastName")
  private String lastName;

  private String email;

  @FormParam("email")
  public void setEmail(String email) {
    this.email = email;
  }

  @Email
  public String getEmail() {
    return email;
  }
  ...
}
```

Note that in this version, `firstName` and `lastName` are fields initialized via injection and `email` is a resource class property. Constraint annotations on properties are specified in their corresponding getters.

Constraint annotations are also allowed on resource classes. In addition to annotating fields and properties, an annotation can be defined for the entire class. Let us assume that `@NonEmptyNames` validates that one of the two *name* fields in `MyResourceClass` is provided. Using such an annotation, the example above can be extended as follows:

```
@Path("/")
@NonEmptyNames
class MyResourceClass {

  @NotNull @FormParam("firstName")
  private String firstName;

  @NotNull @FormParam("lastName")
  private String lastName;

  private String email;
  ...
}
```

Constraint annotations on resource classes are useful for defining cross-field and cross-property constraints.

7.2 Annotations and Validators

Annotation constraints and validators are defined in accordance with the Bean Validation specification [16]. The `@Email` annotation shown above is defined using the Bean Validation `@Constraint` meta-annotation:

```
@Target( { METHOD, FIELD, PARAMETER })
@Retention(RUNTIME)
@Constraint(validatedBy = EmailValidator.class)
public @interface Email {
  String message() default "{com.example.validation.constraints.email}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

The `@Constraint` annotation must include a reference to the validator class that will be used to validate decorated values. The `EmailValidator` class must implement `ConstraintValidator<Email, T>` where `T` is the type of values being validated. For example,

```
public class EmailValidator implements ConstraintValidator<Email, String> {
  public void initialize(Email email) {
    ...
  }

  public boolean isValid(String value, ConstraintValidatorContext context) {
   ...
  }
}
```

Thus, `EmailValidator` applies to values annotated with `@Email` that are of type `String`. Validators for different types can be defined for the same constraint annotation.

Constraint annotations must also define a `groups` element to indicate which processing groups they are associated with. If no groups are specified (as in the example above) the `Default` group is assumed. For simplicity, JAX-RS implementations are NOT REQUIRED to support processing groups other than `Default`. In what follows, we assume that constraint validation is carried out in the `Default` processing group.

7.3 Entity Validation

Request entity bodies can be mapped to resource method parameters. There are two ways in which these entities can be validated. If the request entity is mapped to a Java bean whose class is decorated with Bean Validation annotations, then validation can be enabled using `@Valid`:

```
@StandardUser
class User {

  @NotNull
  private String firstName;
  ...
}
```

```

@Path("/")
class MyResourceClass {

  @POST
  @Consumes("application/xml")
  public void registerUser(@Valid User user) {
    ...
  }
}
```

In this case, the validator associated with `@StandardUser` (as well as those for non-class level constraints like `@NotNull`) will be called to verify the request entity mapped to `user`. Alternatively, a new annotation can be defined and used directly on the resource method parameter.

```
@Path("/")
class MyResourceClass {

  @POST
  @Consumes("application/xml")
  public void registerUser(@PremiumUser User user) {
    ...
  }
}
```

In the example above, `@PremiumUser` rather than `@StandardUser` will be used to validate the request entity. These two ways in which validation of entities can be triggered can also be combined by including `@Valid` in the list of constraints. The presence of `@Valid` will trigger validation of *all* the constraint annotations decorating a Java bean class. This validation will take place in the `Default` processing group unless the `@ConvertGroup` annotation is present. See [16] for more information on `@ConvertGroup`.

Response entity bodies returned from resource methods can be validated in a similar manner by annotating the resource method itself. To exemplify, assuming both `@StandardUser` and `@PremiumUser` are required to be checked before returning a user, the `getUser` method can be annotated as shown next:

```
@Path("/")
class MyResourceClass {

  @GET
  @Path("{id}")
  @Produces("application/xml")
  @Valid @PremiumUser
  public User getUser(@PathParam("id") String id) {
    User u = findUser(id);
    return u;
  }
  ...
}
```

Note that `@PremiumUser` is explicitly listed and `@StandardUser` is triggered by the presence of the `@Valid` annotation —see definition of `User` class earlier in this section.

7.4 Default Validation Mode

According to [16], validation is enabled by default only for the so called *constrained* methods. Getter methods as defined by the Java Beans specification are not constrained methods, so they will not be validated by default. The special annotation `@ValidateOnExecution` defined in [16] can be used to selectively enable and disable validation. For example, you can enable validation on method `getEmail` shown above as follows:

```
@Path("/")
class MyResourceClass {

  @Email
  @ValidateOnExecution
  public String getEmail() {
    return email;
  }
  ...
}
```

The default value for the `type` attribute of `@ValidateOnExecution` is `IMPLICIT` which, in the example above, results in method `getEmail` being validated. See [16] for more information on other uses of this annotation.

Note that if validation for getter methods is *enabled* and a resource method's signature obeys the rules for getters, the resource method may be (unintentionally) invoked during validation. Conversely, if validation for getter methods is *disabled* and the *matching* resource method's signature obeys the rules for getters, the JAX-RS runtime will still validate the method (i.e., the validation preference will be ignored) before invocation.

7.5 Annotation Inheritance

The rules for inheritance of constraint annotation are defined in [16]. It is worth noting that these rules are incompatible with those defined in Section 3.6. Generally speaking, constraint annotations in [16] are cumulative (can be strengthen) across a given type hierarchy while JAX-RS annotations are inherited or, overridden and ignored.

The goal of this specification is to enable validation of JAX-RS resources by leveraging existing Bean Validation implementations. Therefore, JAX-RS implementations MUST follow the constraint annotation rules defined in [16].

7.6 Validation and Error Reporting

Constraint annotations are allowed in the same locations as the following annotations: `@MatrixParam`, `@QueryParam`, `@PathParam`, `@CookieParam`, `@HeaderParam` and `@Context`, *except* in class constructors and property setters. Specifically, they are allowed in resource method parameters, fields and property getters as well as resource classes, entity parameters and resource methods (return values).

The default resource class instance lifecycle is per-request in JAX-RS. Implementations MAY support other lifecycles; the same caveats related to the use of other JAX-RS annotations in resource classes apply to constraint annotations. For example, a constraint validation annotating a constructor parameter in a resource class whose lifecycle is singleton will only be executed once.

JAX-RS implementations SHOULD use the following process to validate resource class instances after they have been instantiated:

Phase 1 Inject field values and initialize bean properties as described in Section 3.2.

Phase 2 Validate annotations on fields, property getters (if enabled) and the resource class. The order in which these validations are executed is implementation dependent.

Phase 3 Validate annotations on parameters passed to the resource method matched.

Phase 4 If no constraint violations found thus far, invoke resource method and validate returned value.

The exception model in [16] defines a base class `javax.validation.ValidationException` and a few subclasses to report errors that are specific to constraint definitions, constraint declarations, group definitions and constraint violations. JAX-RS implementations MUST provide a default exception mapper (see Section 4.4) for `javax.validation.ValidationException` according to the following rules:

1. If the exception is of type `javax.validation.ValidationException` or any of its subclasses *excluding* `javax.validation.ConstraintViolationException`, then it is mapped to a response with status code 500 (Internal Server Error).
2. If the exception is an instance of `javax.validation.ConstraintViolationException`, then:
   (a) If the exception was thrown while validating a method return type, then it is mapped to a response with status code 500 (Internal Server Error) [1].
   (b) Otherwise, it is mapped to a response with status code 400 (Bad Request).

In all cases, JAX-RS implementations SHOULD include a response entity describing the source of the error; however, the exact content and format of this entity is beyond the scope of this specification. As described in Section 4.4, applications can provide their own exception mappers and, consequently, customize the default mapper described above.

[1]The property path of a ConstraintViolation provides information about the location from which an exception originated. See Javadoc for more information.

Chapter 8

Asynchronous Processing

This chapter describes the asynchronous processing capabilities in JAX-RS. Asynchronous processing is supported both in the Client API and in the Server API.

8.1 Introduction

Asynchronous processing is a technique that enables a better and more efficient use of processing threads. On the client side, a thread that issues a request may also be responsible for updating a UI component; if that thread is blocked waiting for a response, the user's perceived performance of the application will suffer. Similarly, on the server side, a thread that is processing a request should avoid blocking while waiting for an external event to complete so that other requests arriving to the server during that period of time can be attended[1].

8.2 Server API

8.2.1 AsyncResponse

Synchronous processing requires a resource method to produce a response upon returning control back to the JAX-RS implementation. Asynchronous processing enables a resource method to inform the JAX-RS implementation that a response is not readily available upon return but will be produced at a future time. This can be accomplished by first *suspending* and later *resuming* the client connection on which the request was received.

Let us illustrate these concepts via an example:

```
@Path("/async/longRunning")
public class MyResource {

    @GET
    public void longRunningOp(@Suspended final AsyncResponse ar) {
        executor.submit(
            new Runnable() {
```

[1]The maximum number of request threads is typically set by the administrator; if that upper bound is reached, subsequent requests will be rejected.

```
                public void run() {
                    executeLongRunningOp();
                    ar.resume("Hello async world!");
                }
            });
    }
    ...
}
```

A resource method that elects to produce a response asynchronously must inject as a method parameter an instance of the class `AsyncResponse` using the special annotation `@Suspended`. In the example above, the method `longRunningOp` is called upon receiving a `GET` request. Rather than producing a response immediately, this method forks a (non-request) thread to execute a long running operation and returns immediately. Once the execution of the long running operation is complete, the connection is resumed and the response returned by calling `resume` on the injected instance of `AsyncResponse`.

For more information on executors, concurrency and thread management in a Java EE environment, the reader is referred to JSR 236 [13]. For more information about executors in the JAX-RS Client API see Section 5.8.

8.2.1.1 Timeouts and Callbacks

A timeout value can be specified when suspending a connection to avoid waiting for a response indefinitely. The default unit is milliseconds, but any unit of type `java.util.concurrent.TimeUnit` can be used. The following example sets a timeout of 15 seconds and registers an instance of `TimeoutHandler` in case the timeout is reached before the connection is resumed.

```
    @GET
    public void longRunningOp(@Suspended final AsyncResponse ar) {
        // Register handler and set timeout
        ar.setTimeoutHandler(new TimeoutHandler() {
            public void handleTimeout(AsyncResponse ar) {
                ar.resume(Response.status(SERVICE_UNAVAILABLE).entity(
                    "Operation timed out -- please try again").build());
                }
        });
        ar.setTimeout(15, SECONDS);

        // Execute long running operation in new thread
        executor.execute(
            new Runnable() {
                public void run() {
                    executeLongRunningOp();
                    ar.resume("Hello async world!");
                }
            });
    }
```

JAX-RS implementations are REQUIRED to generate a `ServiceUnavailableException`, a subclass of `WebApplicationException` with its status set to 503, if the timeout value is reached and no timeout handler is registered. The exception MUST be processed as described in section 3.3.4. If a registered timeout handler resets the timeout value or resumes the connection and returns a response, JAX-RS implementations MUST NOT generate an exception.

It is also possible to register callbacks on an instance of `AsyncResponse` in order to listen for processing completion (`CompletionCallback`) and connection termination (`ConnectionCallback`) events. See Javadoc for `AsyncResponse` for more information on how to register these callbacks. Note that support for `ConnectionCallback` is OPTIONAL.

8.2.2 CompletionStage

An alternative approach to the injection of `AsyncResponse` is for a resource method to return an instance of `CompletionStage<T>` as an indication to the underlying JAX-RS implementation that asynchronous processing is enabled. The example from Section 8.2.1 can be re-written using `CompletionStage` as follows:

```
@Path("/async/longRunning")
public class MyResource {

    @GET
    public CompletionStage<String> longRunningOp() {
        CompletableFuture<String> cs = new CompletableFuture<>();
        executor.submit(
            new Runnable() {
                public void run() {
                    executeLongRunningOp();
                    cs.complete("Hello async world!");
                }
            });
        return cs;
    }
    ...
}
```

In this example, a `CompletableFuture` instance is created and returned in the resource method; the call to method `complete` on that instance is executed only after the long running operation terminates.

8.3 EJB Resource Classes

As stated in Section 11.2.4, JAX-RS implementations in products that support EJB must also support the use of stateless and singleton session beans as root resource classes. When an EJB method is annotated with `@Asynchronous`, the EJB container automatically allocates the necessary resources for its execution. Thus, in this scenario, the use of an `Executor` is unnecessary to generate an asynchronous response.

Consider the following example:

```
@Stateless
@Path("/")
class EJBResource {

    @GET @Asynchronous
    public void longRunningOp(@Suspended AsyncResponse ar) {
```

```
            executeLongRunningOp();
            ar.resume("Hello async world!");
        }
    }
```

There is no explicit thread management needed in this case since that is under the control of the EJB container. Just like the other examples in this chapter, the response is produced by calling `resume` on the injected `AsyncResponse`. Hence, the return type of `longRunningOp` is simply `void`.

8.4 Client API

The fluent API supports asynchronous invocations as part of the invocation building process. By default, invocations are synchronous but can be set to run asynchronously by calling the `async` method and (optionally) registering an instance of `InvocationCallback` as shown next:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://example.org/customers/{id}");
target.resolveTemplate("id", 123).request().async().get(
    new InvocationCallback<Customer>() {
        @Override
        public void completed(Customer customer) {
            // Do something
        }
        @Override
        public void failed(Throwable throwable) {
            // Process error
        }
    });
```

Note that in this example, the call to `get` after calling `async` returns immediately without blocking the caller's thread. The response type is specified as a type parameter to `InvocationCallback`. The method `completed` is called when the invocation completes successfully and a response is available; the method `failed` is called with an instance of `Throwable` when the invocation fails.

All asynchronous invocations return an instance of `Future<T>` here the type parameter `T` matches the type specified in `InvocationCallback`. This instance can be used to monitor or cancel the asynchronous invocation:

```
Future<Customer> ff = target.resolveTemplate("id", 123).request().async()
    .get(new InvocationCallback<Customer>() {
        @Override
        public void completed(Customer customer) {
            // Do something
        }
        @Override
        public void failed(Throwable throwable) {
            // Process error
        }
    });

// After waiting for a while ...
```

```
if (!ff.isDone()) {
    ff.cancel(true);
}
```

Even though it is recommended to pass an instance of `InvocationCallback` when executing an asynchronous call, it is not mandated. When omitted, the `Future<T>` returned by the invocation can be used to gain access to the response by calling the method `Future.get`, which will return an instance of `T` if the invocation was successful or `null` if the invocation failed.

Chapter 9

Server-Sent Events

9.1 Introduction

Server-sent events (SSE) is a specification originally introduced as part of HTML5 by the W3C, but is currently maintained by the WHATWG [17]. It provides a way to establish a one-way channel from a server to a client. The connection is long running: it is re-used for multiple events sent from the server, yet it is still based on the HTTP protocol. Clients request the opening of an SSE connection by using the special media type `text/event-stream` in the `Accept` header.

Events are structured and contain several fields, namely, `event`, `data`, `id`, `retry` and `comment`. SSE is a messaging protocol where the `event` field corresponds to a topic, and where the `id` field can be used to validate event order and guarantee continuity. If a connection is interrupted for any reason, the `id` can be sent in a request header for a server to re-play past events —although this is an optional behavior that may not be supported by all implementations. Event payloads are conveyed in the `data` field and must be in text format; `retry` is used to control re-connects and finally `comment` is a general purpose field that can also be used to keep connections alive.

9.2 Client API

The JAX-RS client API for SSE was inspired by the corresponding JavaScript API in HTML5, but with changes that originate from the use of a different language. The entry point to the client API is the type `SseEventSource`, which provides a fluent builder similarly to other classes in the JAX-RS API. An `SseEventSource` is constructed from a `WebTarget` that is already configured with a resource location; `SseEventSource` does not duplicate any functionality in `WebTarget` and only adds the necessary logic for SSE.

The following example shows how to open an SSE connection and read some messages for a little while:

```
WebTarget target = client.target("http://...");
try (SseEventSource source = SseEventSource.target(target).build()) {
  source.register(System.out::println);
  source.open();
  Thread.sleep(500);      // Consume events for just 500 ms
} catch (InterruptedException e) {
  // falls through
}
```

As seen in this example, an SseEventSource implements AutoCloseable. Before opening the source, the client registers an event consumer that simply prints each event. Additional handlers for other life-cycle events such as onComplete and onError are also supported, but for simplicity only onEvent is shown in the example above.

9.3 Server API

The JAX-RS SSE server API is used to accept connections and send events to one or more clients. A resource method that injects an SseEventSink and produces the media type text/event-stream is an SSE resource method.

The following example accepts SSE connections and uses an executor thread to send 3 events before closing the connection:

```
@GET
@Path("eventStream")
@Produces(MediaType.SERVER_SENT_EVENTS)
public void eventStream(@Context SseEventSink eventSink,
                        @Context Sse sse) {
  executor.execute(() -> {
    try (SseEventSink sink = eventSink) {
      eventSink.send(sse.newEvent("event1"));
      eventSink.send(sse.newEvent("event2"));
      eventSink.send(sse.newEvent("event3"));
    }
  });
}
```

SSE resource methods follow a similar pattern to those for asynchronous processing (see Section 8.1) in that the object representing the incoming connection, in this case SseEventSink, is injected into the resource method.

The example above also injects the Sse type which provides factory methods for events and broadcasters. See section 9.4 for more information about broadcasting. Note that, just like SseEventSource, the interface SseEventSink is also auto-closeable, hence the use of the *try-with-resources* statement above.

Method send on SseEventSink returns a CompletionStage<?> as a way to provide a *handle* to the action of asynchronously sending a message to a client.

9.4 Broadcasting

Applications may need to send events to multiple clients simultaneously. This action is called *broadcasting* in JAX-RS. Multiple SseEventSink's can be registered on a single SseBroadcaster.

A broadcaster can only be created by calling method newBroadcaster on the injected Sse instance. The life-cycle and scope of an SseBroadcaster is fully controlled by applications and not the JAX-RS runtime. The following example shows the use of broadcasters, note the @Singleton annotation on the resource class:

```
@Path("/")
```

```
@Singleton
public class SseResource {

  @Context
  private Sse sse;

  private volatile SseBroadcaster sseBroadcaster;

  @PostConstruct
  public init() {
    this.sseBroadcaster = sse.newBroadcaster();
  }

  @GET
  @Path("register")
  @Produces(MediaType.SERVER_SENT_EVENTS)
  public void register(@Context SseEventSink eventSink) {
    eventSink.send(sse.newEvent("welcome!"));
    sseBroadcaster.register(eventSink);
  }

  @POST
  @Path("broadcast")
  @Consumes(MediaType.MULTIPART_FORM_DATA)
  public void broadcast(@FormParam("event") String event) {
    sseBroadcaster.broadcast(sse.newEvent(event));
  }
}
```

The `register` method on a broadcaster is used to add a new `SseEventSink`; the `broadcast` method is used to send an SSE event to all registered consumers.

9.5 Processing Pipeline

Connections from SSE clients are represented by injectable instances of `SseEventSink`. There are some similarities between SSE and asynchronous processing (see Chapter 8). Asynchronous responses can be resumed at most once while an `SseEventSink` can be used multiple times to stream individual events.

For compatibility purposes, implementations MUST initiate processing of an SSE response when either the first message is sent or when the resource method returns, whichever happens first. The initial SSE response, which may only include the HTTP headers, is processed using the standard JAX-RS pipeline as described in Appendix C. Each subsequent SSE event may include a different payload and thus require the use of a specific message body writer. Note that since this use case differs slightly from the normal JAX-RS pipeline, implementations SHOULD NOT call entity interceptors on each individual event [1].

9.6 Environment

The `SseEventSource` class uses the existing JAX-RS mechanism based on `RuntimeDelegate` to find an implementation using the service name `javax.ws.rs.sse.SseEventSource.Builder`. The majority

[1] As a matter of fact, there is no API to bind entity interceptors to individual SSE events.

of types in the `javax.ws.rs.sse` are thread safe; the reader is referred to the Javadoc for more information on thread safety.

Chapter 10

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 `Application` subclasses (see Section 2.1), root resource classes (see Chapter 3), and providers (see Chapter 4). This chapter describes these facilities.

10.1 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 10.2, 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.

10.2 Context Types

This section describes the types of context available to providers (client and server) as well as resource classes and `Application` subclasses (server only). Except for `Configuration` and `Providers`, which are injectable in both client and server-side providers, all the other types are server-side only.

10.2.1 Application

The instance of the application-supplied `Application` subclass can be injected into a class field or method parameter using the `@Context` annotation. Access to the `Application` subclass instance allows configuration information to be centralized in that class. Note that this cannot be injected into the `Application` subclass itself since this would create a circular dependency.

10.2.2 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.

10.2.3 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.

10.2.4 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.

10.2.5 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 principal, information about roles assumed by the requester, whether the request arrived over a secure channel and the authentication scheme used.

10.2.6 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. It is injectable in both client and server providers.

10.2.7 Resource Context

The `ResourceContext` interface provides access to instantiation and initialization of resource or subresource classes in the default per-request scope. It can be injected to help with creation and initialization, or just initialization, of instances created by an application.

Let us revisit the example from Section 3.4.1 with some simple modifications:

```
@Path("widgets")
public class WidgetsResource {
  @Context
  private ResourceContext rc;

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

public class WidgetResource {
  @Context
  private HttpHeaders headers;

  public WidgetResource(String id) {...}

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

```
}
```

Note that the instance returned by the resource locator `findWidget` in `WidgetsResource` is initialized using the injected `ResourceContext` before it is returned. Without this step, the `headers` field in `WidgetResource` will not be properly initialized.

10.2.8 Configuration

Both the client and the server runtime configurations are available for injection via `@Context`. These configurations are available for injection in providers (client or server) and resource classes (server only).

As an example, consider the case of a client logging filter that not only logs messages but also logs information about certain features enabled during the processing of a request:

```
public class LoggingFilter implements ClientRequestFilter {

  @Context
  private Configuration config;

  @Override
  public void filter(ClientRequestContext ctx) throws IOException {
    if (config.isEnabled(MyFeature.class)) {
      logMyFeatureEnabled(ctx);
    }
    logMessage(ctx);
  }
  ...
}
```

A client runtime configuration is injected in the filter shown above and its `isEnabled` method called to check if `MyFeature` is enabled.

Chapter 11

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 10.2 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.

11.1 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.

Servlet filters may trigger consumption of a request body by accessing request parameters. In a servlet container the @FormParam annotation and the standard entity provider for `application/x-www-form--urlencoded` MUST obtain their values from the servlet request parameters if the request body has already been consumed. Servlet APIs do not differentiate between parameters in the URI and body of a request so URI-based query parameters may be included in the entity parameter.

11.2 Integration with Java EE Technologies

This section describes the additional requirements that apply to a JAX-RS implementation when combined in a product that supports the following specifications.

11.2.1 Servlets

In a product that also supports the Servlet specification, implementations MUST support JAX-RS applications that are packaged as a Web application. See Section 2.3.2 for more information Web application packaging.

It is RECOMMENDED for a JAX-RS implementation to provide asynchronous processing support, as defined in Chapter 8, by enabling asynchronous processing (i.e., `asyncSupported=true`) in the underlying Servlet 3 container. It is OPTIONAL for a JAX-RS implementation to support asynchronous processing when running on a Servlet container whose version is prior to 3.

As explained in Section 11.1, injection of Servlet-defined types is possible using the `@Context` annotation. Additionally, web application's `<context-param>` and servlet's `<init-param>` can be used to define application properties passed to server-side features or injected into server-side JAX-RS components. See Javadoc for `Application.getProperties` for more information.

11.2.2 Managed Beans

In a product that supports Managed Beans, implementations MUST support the use of Managed Beans as root resource classes, providers and `Application` subclasses.

For example, a bean that uses a managed-bean interceptor can be defined as a JAX-RS resource as follows:

```
@ManagedBean
@Path("/managedbean")
public class ManagedBeanResource {

    public static class MyInterceptor {
        @AroundInvoke
        public String around(InvocationContext ctx) throws Exception {
            System.out.println("around() called");
            return (String) ctx.proceed();
        }
    }

    @GET
    @Produces("text/plain")
    @Interceptors(MyInterceptor.class)
    public String getIt() {
        return "Hi managedbean!";
    }
}
```

The example above uses a managed-bean interceptor to intercept calls to the resource method `getIt`. See Section 11.2.8 for additional requirements on Managed Beans.

11.2.3 Context and Dependency Injection (CDI)

In a product that supports CDI, implementations MUST support the use of CDI-style Beans as root resource classes, providers and `Application` subclasses. Providers and `Application` subclasses MUST be singletons or use application scope.

For example, assuming CDI is enabled via the inclusion of a `beans.xml` file, a CDI-style bean that can be defined as a JAX-RS resource as follows:

```
@Path("/cdibean")
public class CdiBeanResource {

    @Inject MyOtherCdiBean bean;         // CDI injected bean

    @GET
    @Produces("text/plain")
    public String getIt() {
        return bean.getIt();
    }
}
```

The example above takes advantage of the type-safe dependency injection provided in CDI by using another bean, of type `MyOtherCdiBean`, in order to return a resource representation. See Section 11.2.8 for additional requirements on CDI-style Beans.

11.2.4 Enterprise Java Beans (EJBs)

In a product that supports EJBs, an implementation MUST support the use of stateless and singleton session beans as root resource classes, providers and `Application` subclasses. JAX-RS annotations can be applied to methods in an EJB's local interface or directly to methods in a no-interface EJB. Resource class annotations (like `@Path`) MUST be applied to an EJB's class directly following the annotation inheritance rules defined in Section 3.6.

For example, a stateless EJB that implements a local interface can be defined as a JAX-RS resource class as follows:

```
@Local
public interface LocalEjb {

    @GET
    @Produces("text/plain")
    public String getIt();
}

@Stateless
@Path("/stateless")
public class StatelessEjbResource implements LocalEjb {

    @Override
    public String getIt() {
        return "Hi stateless!";
    }
}
```

JAX-RS implementations are REQUIRED to discover EJBs by inspecting annotations on classes and local interfaces; they are not REQUIRED to read EJB deployment descriptors (ejb-jar.xml). Therefore, any information in an EJB deployment descriptor for the purpose of overriding EJB annotations or providing additional meta-data will likely result in a non-portable JAX-RS application.

If an `ExceptionMapper` for a `EJBException` or subclass is not included with an application then exceptions thrown by an EJB resource class or provider method MUST be unwrapped and processed as described in Section 3.3.4.

See Section 8.3 for more information on asynchronous EJB methods and Section 11.2.8 for additional requirements on EJBs.

11.2.5 Bean Validation

In a product that supports the Bean Validation specification [16], implementations MUST support resource validation using constraint annotations as described in Chapter 7. Otherwise, support for resource validation is OPTIONAL.

11.2.6 Java API for JSON Processing

In a product that supports the Java API for JSON Processing (JSON-P) [18], implementations MUST support entity providers for `JsonValue` and all of its sub-types: `JsonStructure`, `JsonObject`, `JsonArray`, `JsonString` and `JsonNumber`.

Note that other types from the JSON-P API such as `JsonParser`, `JsonGenerator`, `JsonReader` and `JsonWriter` can also be integrated into JAX-RS applications using the entity providers for `InputStream` and `StreamingOutput`.

11.2.7 Java API for JSON Binding

In a product that supports the Java API for JSON Binding (JSON-B) [19], implementations MUST support entity providers for all Java types supported by JSON-B in combination with the following media types: `application/json`, `text/json` as well as any other media types matching `*/json` or `*/*+json`.

Note that if JSON-B and JSON-P are both supported in the same environment, entity providers for JSON-B take precedence over those for JSON-P for all types except `JsonValue` and its sub-types.

11.2.8 Additional Requirements

The following additional requirements apply when using Managed Beans, CDI-style Beans or EJBs as resource classes, providers or `Application` subclasses:

- Field and property injection of JAX-RS resources MUST be performed prior to the container invoking any `@PostConstruct` annotated method.
- Support for constructor injection of JAX-RS resources is OPTIONAL. Portable applications MUST instead use fields or bean properties in conjunction with a `@PostConstruct` annotated method. Implementations SHOULD warn users about use of non-portable constructor injection.
- Implementations MUST NOT require use of `@Inject` or `@Resource` to trigger injection of JAX-RS annotated fields or properties. Implementations MAY support such usage but SHOULD warn users about non-portability.

11.3 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 10.2.

Chapter 12

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 12.1. 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.

12.1 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. Use the Java SE class `java.util.ServiceLoader` to attempt to load an implementation from `META-INF/services/javax.ws.rs.ext.RuntimeDelegate`. Note that this may require more

than one call to method `ServiceLoader.load(Class, ClassLoader)` in order to try both the context class loader and the current class loader as explained above [1].

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.

[1] Earlier versions of JAX-RS did not mandate the use `ServiceLoader`. This backward-compatible change that started in JAX-RS 2.1 is to ensure forward compatibility with the Java SE 9 module system.

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.
`PATCH`	Method	Specifies that the annotated method handles HTTP PATCH requests.
`HEAD`	Method	Specifies that the annotated method handles HTTP HEAD requests. Note that HEAD may be automatically handled, see Section 3.3.5.
`OPTIONS`	Method	Specifies that the annotated method handles HTTP OPTIONS requests.
`ApplicationPath`	Type	Specifies the resource-wide application path that forms the base URI of all root resource classes.
`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.

Annotation	Target	Description
`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.
`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 10.2 or the applicable section of Chapter 11.
`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.
Since JAX-RS 2.0		
`NameBinding`	Annotation	Meta-annotation to create annotations for binding filters or interceptors to resource methods and applications. Name binding is only supported as part of the Server API.
`Suspended`	Parameter	Indicates that a resource method is asynchronous. I.e., that it does not produce a response upon returning. JAX-RS implementations will suspend the incoming connection until a response becomes available.
`PreMatching`	Type	Global binding annotation that can be applied to a container filter to indicate that it should be applied globally and before the resource method is matched.

Annotation	Target	Description
`BeanParam`	Parameter, field or method	Can be used to inject a user-defined bean whose fields and properties may be annotated with JAX-RS param annotations.
`ConstrainedTo`	Type	Can be used to restrict the applicability of a provider to just the Client API or just the Server API. If omitted, a provider can be used in either context.
`ParamConverter.Lazy`	Type	Indicates that a conversion of a default value delegated to a `ParamConverter` SHOULD occur only when the value is actually requested.

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 10.2.3; 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 10.2.4.
`Accept-Encoding`	Processed by runtime if application uses `Request.selectVariant` method, see Section 10.2.4.
`Accept-Language`	Processed by runtime if application uses `Request.selectVariant` method, see Section 10.2.4.
`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 10.2.5.
`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 message 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.
`Date`	Included in responses automatically as per HTTP/1.1.

Header	Description
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 10.2.4.
If-Modified-Since	Processed by runtime if application uses corresponding Request.evaluatePreconditions method, see Section 10.2.4.
If-None-Match	Processed by runtime if application uses corresponding Request.evaluatePreconditions method, see Section 10.2.4.
If-Unmodified-Since	Processed by runtime if application uses corresponding Request.evaluatePreconditions method, see Section 10.2.4.
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.

Appendix C

Processing Pipeline

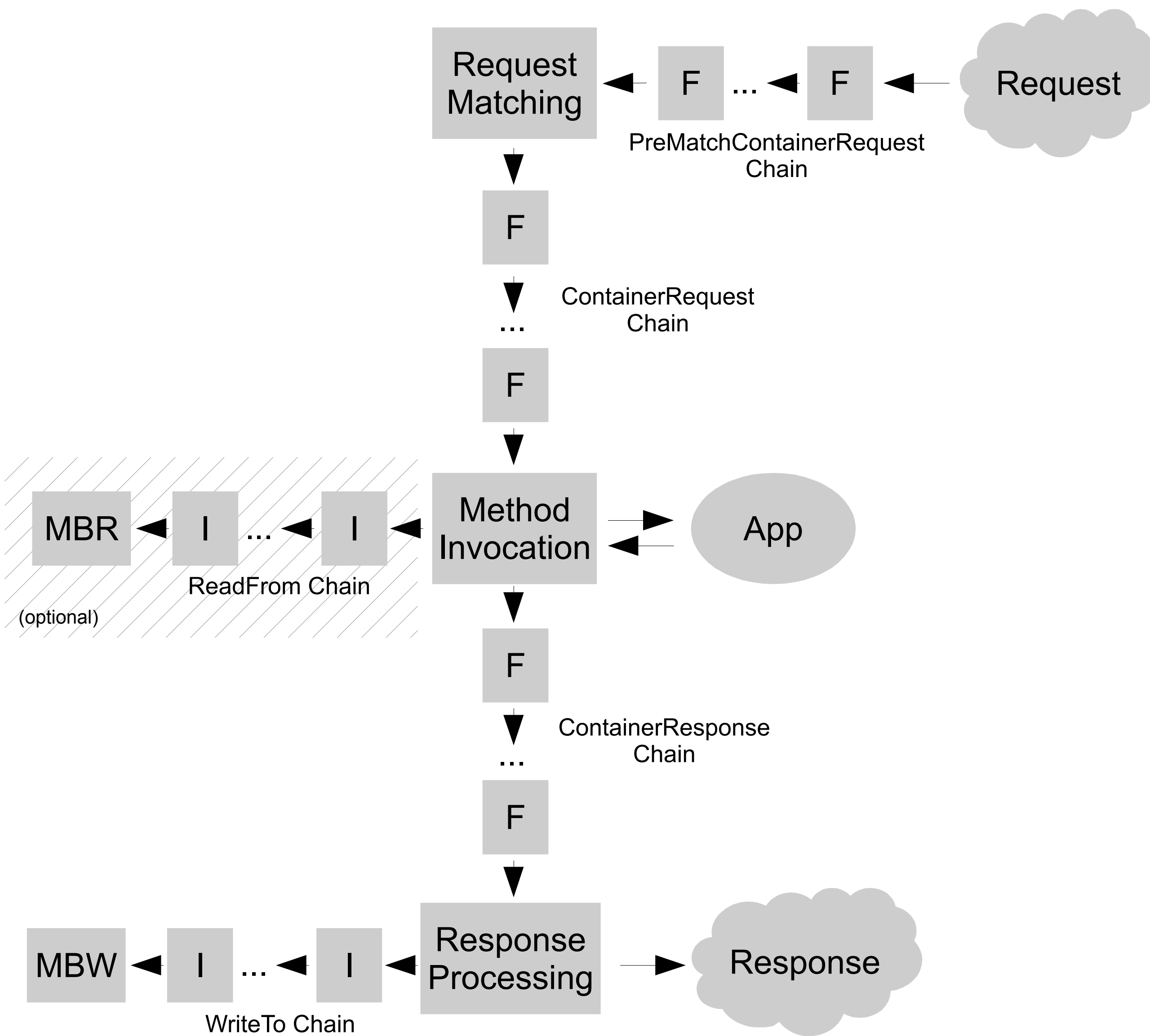

Figure C.1: JAX-RS Server Processing Pipeline.

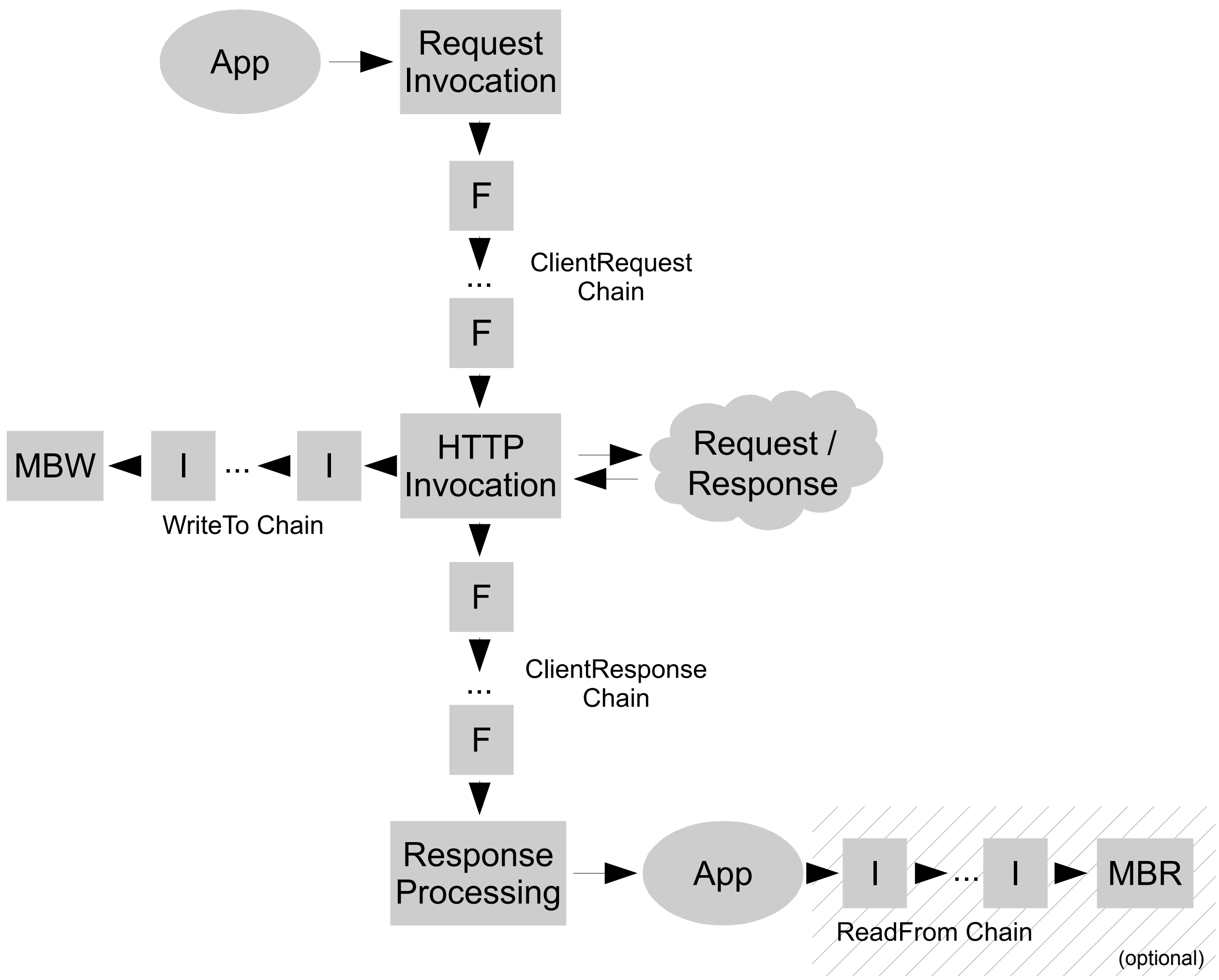

Figure C.2: JAX-RS Client Processing Pipeline.

Appendix D

Change Log

D.1 Changes Since 2.1 Public Review

- Section 9.5: New section describing the SSE processing pipeline.
- Section 3.4.1: Allow sub-resource locators to return classes as well.
- Section 5.8: Removed defaults for all environments not supporting the Java Concurrenty Utilities for Java EE API.
- Section 11.2.6: Support for `JsonValue` and all its sub-types.
- Section 11.2.7: New section for JSON-B support.
- Section 8.2.2: New section for `CompletionStage<T>` as return type in a resource method.
- Section 4.1.3: New section introducing support for `@Priority` for all providers.
- Section 4.2: Modified algorithm steps to add support for `@Priority` in selection of message body readers and writers.

D.2 Changes Since 2.1 Early Draft

- Chapter 9: API changes and removal of Flow API references.
- Section 3.3: Added `@PATCH` to list of supported HTTP methods.
- Section 5.8: New section about executor services.
- Section 12.1: Use of `ServiceLoader` in first step of algorithm.

D.3 Changes Since 2.0 Final Release

- Section 3.2: Clarified exception handling for all 5 steps used to convert a string to a Param. Allowed the combination of `List<T>`, `Set<T>`, or `SortedSet<T>` and `ParamConverter`.
- Section 3.7.2: Defined client and server media types.

- Section 5.7: New section introducing support for reactive clients.
- Chapter 9: New chapter describing the Server-Sent Events API.

D.4 Changes Since 2.0 Proposed Final Draft

- Section 4.2.1 and 4.2.2: Updated last logical step separating client and server runtimes.
- Section 4.2.4: New exception `NoContentException` to handle zero-length entities.
- Section 4.5 and 6.7: New subsections to separate client and server processing of exceptions.
- Section 5.6: `Invocation` and `Invocation.Builder` are no longer configurable types.
- Section 5.6.1: Updated samples based on changes to Client API.
- Section 7.4: Updated section based on changes to Bean Validation API.
- Section 7.6: Minor re-wording of Phase 2.
- Section 8.2.1.1: Improved example to return a response if a timeout expires.
- Section 8.4: Updated samples based on changes to Client API.
- Section 10.2: Clarified server vs. client injectable types.
- Section 10.2.8: New section about injection of `Configuration` instances.
- Section 11.2.5: New section stating that support for resource validation is required only in a product that supports Bean Validation.

D.5 Changes Since 2.0 Public Review Draft

- In Javadocs: Method `getSize` in class `MessageBodyWriter` is deprecated.
- Chapter 3 and 4: Replaced `WebApplicationException` by corresponding subclass depending on HTTP status code.
- Section 3.2: New step for `ParamConverter`.
- Section 4.2.4: Special case for zero-length entities and primitive types.
- Chapter 5: Updated samples and text related to the configuration of Client API types. Method `configuration` has been removed in favor of the `Configurable` interface.
- Chapter 5: `ClientFactory` renamed to `ClientBuilder`.
- Chapter 5: Dropped support for `@Uri` annotation.
- Section 6.3: New paragraph clarifying that entity interceptors are not called if a `readFrom` or `writeTo` method is called directly from application code.
- Section 6.3: Improved `GzipInterceptor` sample.

- Section 6.5.2: Clarified name binding with multiple annotations. Made semantics compatible with CDI interceptor binding. Name binding with `Application` subclasses.
- Section 6.6: Annotation `@BindingPriority` replaced by common annotation `@Priority`.
- Section 6.7: Clarified processing of a response mapped from an exception.
- Section 8.2: Updated samples in this section. New citation for JSR 236.
- Section 8.2.1.1: Improved example.
- Section 8.4: Fixed signature of method `failed` in samples.
- Section 7.4: New section about default validation and `@ValidateExecutable`.
- Section 7.6: Restored validation phases.
- Section 11.2.6: New section about integration with JSON-P.

D.6 Changes Since 2.0 Early Draft (Third Edition)

- Chapter “Validation”: Removed from PR. JAX-RS 2.0 will defer to CDI for Bean Validation. Only those resource classes that are CDI beans would be validated.
- Section 3.3.4: Clarified that the steps must be followed in order and that throwables are propagated to the underlying container only if they cannot be mapped.
- Section 3.3: Added `@OPTIONS` to list.
- Section 3.3.3: New note about anonymous inner classes.
- Section 3.7.2: Allow multiple root resource classes to share same URI in algorithm. Note changes in output and input of steps 1 and 2, respectively.
- Section 4.2.2: Throw new exception `InternalServerErrorException`.
- Section 4.2: Removed steps that required the use of the JavaBeans Activation Framework[20] to map objects to representations and vice-versa. The EG deemed this feature to be confusing and not portable across implementations.
- Section 4.2.4: Support for pre-packaged readers and writers for Java types `java.lang.Boolean`, `java.lang.Character`, `java.lang.Number` and media type `text/plain`.
- Section 4.2.4: Detailed behavior for readers and zero-length request entities.
- Section 6.5.3: `DynamicBinder` replaced by `DynamicFeature`.
- Section 6.7: Clarified processing of responses mapped using exception mappers.
- Chapter 8: Updated sections related to the Server API. `@Suspended` annotation, timeouts and callbacks.
- Section 8.2.1.1: Throw new exception `ServiceUnavailableException`.
- Section 10.2.7: New section for `ResourceContext`.

- Section 11.2.4: Clarified location of annotations in accordance to the JAX-RS annotation inheritance rules.
- Chapter 6: Replaced `@PostMatching` by `@PreMatching`. Post-matching is now the default.
- Appendix A: Edited section on JAX-RS 2.0 annotations.

D.7 Changes Since 2.0 Early Draft (Second Edition)

- Section 3.4: New example showing a scenario in which a `@PathParam` is not be available for injection.
- Section 3.7.2: Formalization of certain parts of the matching algorithm that were vague. Defined formal ordering between media types and highlighted situations in which implementations should report a warning if the matching is ambiguous.
- Section 3.7.2: New sample showing the resource matching algorithm in action.
- Section 3.7.3: New footnote about syntax of regular expression.
- Chapter 4: The annotation `@Provider` is now only required for automatic discovery of providers (typically via class scanning). It is no longer required for a provider that is manually registered in classes such as `Application` or `Configuration`.
- Section 4.1.1: New section about automatic discovery of provider classes. Only those annotated with `@Provider` must be discovered automatically.
- Chapter 5: Features are now providers and can be registered as such. A feature can no longer be disabled.
- Chapter 5: Class `Target` renamed to `WebTarget`. Removed text describing how to use a builder factory class (no longer supported). A few other minor updates and typos fixed.
- Chapter 6: Extension points for filters revised. New filter interfaces `ClientRequestFilter` and `ClientResponseFilter` in Client API and new filter interfaces `ContainerRequestFilter` and `ContainerResponseFilter` in Server API. Corresponding context classes also updated in chapter. A `ContainerRequestFilter` is executed before resource matching unless annotated with `@PostMatching`. It is no longer possible to share a filter implementation between the Client and Server APIs; entity interceptors are still shareable.
- Chapter 6: Section on the relationship between filters and entity interceptors (including diagram) dropped.
- Section 6.5.1: Clarified global binding in relation to the new semantics of `@Provider` for automatic discovery.
- Section 6.5.3: The `DynamicBinding` interface, intended to be implemented by filters and entity interceptors, is replaced by `DynamicBinder`. A dynamic binder is a new type of provider that binds filters and entity interceptors with resource methods.
- Chapter "Validation": Use media type names instead of Java constants for clarity. More descriptive names for constraint annotations. Changed inheritance rules for constraint annotations to follow those defined in [16]. New note about `@Valid` support for return values. Fixed phase number typo.

- Section 8.2.1.1: New sentence about calling `suspend` more than once or in a method annotated with `@Suspend`.
- Section 8.3: New section about EJB resource classes with methods annotated with `@Asynchronous`.
- Chapter 11: Re-structured chapter with sub-sections for each of the EE technologies with which JAX-RS integrates. New examples added.

D.8 Changes Since 2.0 Early Draft

- Section 1.6: Updated 2.0 expert group member list.
- Section 1.7: Updated acknowledgements for 2.0 version.
- Section 4.5: New section that describes exception handling for all providers. Related changes to reference this new section.
- Section 6.7: New section for filter and interceptor exceptions. Related changes to reference this new section.
- Section 3.7.2: Updated step 2a so that it only goes to step 3 when $M \neq \{\}$.
- Section 3.7.2: New sentence about the use of a server quality parameter (qs-value) during request matching.
- Chapter 6: New extension point for pre-match filters that are executed before resource matching.
- Chapter 6: Filter methods no longer return a next action; the execution of a request filter chain is stopped by setting a response in the associated context.
- Chapter 6: Handlers renamed to entity interceptors for clarity (in this log included).
- Section 6.5.1: Global binding is now the default for filters and entity interceptors. The annotation `@GlobalBinding` has been dropped.
- Section 6.6: Clarified reverse ordering of the response filter chain based on binding priorities.
- Appendix C: Removed from this version after changes to Chapter 6.
- Chapter "Validation": Moved to an instantiate-then-validate strategy in which validation of constructor parameters and setters is no longer supported. Simplified validation process to better align with Bean Validation 1.1 [16]. In particular, validation of resource classes, fields and property getters are all carried out in a single step.

D.9 Changes Since 1.1 Release

- Section 1.1: Updated URLs to JSR pages, etc.
- Section 1.3: Removed Client APIs as non-goal.
- Section 1.5: Added new terminology.
- Section 1.6: List 2.0 expert group members.

- Section 1.7: acknowledgements for 2.0 version.
- Chapter 2: Somewhat generic section on validation removed to avoid confusion with the type of validation defined in Chapter "Validation".
- Section 2.3.2: Clarified used of Servlet 3 framework pluggability. Added sample web.xml files and a table summarizing all cases.
- Section 3.3.2.1: Clarified notion of entity parameter as a parameter not annotated with any JAX-RS annotation.
- Section 3.5: Explained use of quality factor q. Introduced server-side quality factor qs and included example.
- Section 3.6: Added sentence about conflicting annotations and recommendation to repeat annotations for consistency with other Java EE specifications.
- Section 3.7.1: Highlighted input and output for each step in algorithm. Minor edits to simplify presentation.
- Section 3.8: Updated algorithm to support server-side quality factor qs.
- Chapter 5: New chapter Client API.
- Chapter 6: New chapter Filters and Interceptors.
- Chapter "Validation": New chapter Validation.
- Chapter 8: New chapter Asynchronous Processing.
- Appendix A: New section for 2.0 annotations.
- Appendix C: New appendix describing filter and interceptor extension points.

D.10 Changes Since 1.0 Release

- Section 2.3.2: New requirements for Servlet 3 containers.
- Section 11.2: Requirements for Java EE 6 containers.
- Section 4.2.4: Requirements on standard entity providers when presented with an empty message entity.
- Section 4.2.2: Add closeness of generic type as secondary sort key.
- Section 4.2.1: Default to application/octet-stream if a request does not contain a content-type header.
- Section 3.2: Add support for static fromString method.
- Section 3.6: Clarify annotation inheritance.
- Section 10.2.5: Fix typo.
- Section 11.1: Additional considerations related to filters consuming request bodies.

D.11 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`.
- Section 4.2.2: Fixed algorithm to refer to Section 3.8 instead of restating it. Fixed status code returned when the media type has been determined but an appropriate message body writer cannot be located.
- Chapter 12: Clarify that an implementation can supply an alternate `RuntimeDelegate` API class.

D.12 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 4.1.2: 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 10.2: Removed `ContextResolver` from list of injectable resources.
- Section 10.2.6: 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] RxJava: a library for composing asynchronous and event-based programs using observable sequences for the java VM. See https://github.com/ReactiveX/RxJava.

[12] RxJava 2.0: What's different in 2.0. See https://github.com/ReactiveX/RxJava/wiki/What's-different-in-2.0.

[13] Anthony Lai. Concurrency Utilities for Java EE. JSR, JCP, March 2013. See http://jcp.org/en/jsr/detail?id=236.

[14] Gavin King. Context and Dependency Injection for the Java Platform. JSR, JCP, December 2009. See http://jcp.org/en/jsr/detail?id=299.

[15] Rajiv Mordani. Common Annotations for the Java Platform. JSR, JCP, July 2005. See http://jcp.org/en/jsr/detail?id=250.

[16] Emmanuel Bernard. Bean Validation 1.1. JSR, JCP, March 2013. See http://jcp.org/en/jsr/detail?id=349.

[17] Server-Sent Events. See https://html.spec.whatwg.org/#server-sent-events.

[18] Dmitry Kornilov. Java API for JSON Processing 1.1. JSR, JCP, May 2017. See http://jcp.org/en/jsr/detail?id=374.

[19] Dmitry Kornilov. Java API for JSON Binding. JSR, JCP, July 2017. See http://jcp.org/en/jsr/detail?id=367.

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