

JAX-RS: Java™ API for RESTful Web Services

*Version 2.0 Draft of Early Draft
August 17, 2011*

Editors:
Santiago Pericas-Geertsen
Marek Potociar

Comments to: users@jax-rs-spec.java.net

*Oracle Corporation
500 Oracle Parkway, Redwood Shores, CA 94065 USA.*

ORACLE IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THEM, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

Specification: JAX-RS - Java™ API for RESTful Web Services ("Specification")

Version: draft-of-2.0-early-draft

Status: Draft of 2.0 Early Draft

Release: August 17, 2011

Copyright 2011 Oracle Corporation

500 Oracle Parkway, Redwood Shores, California 94065, U.S.A

All rights reserved.

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Oracle USA, Inc. ("Oracle") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Oracle hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Oracle's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.
2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:
 - (a) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;
 - (b) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and
 - (c) includes the following notice: "This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Oracle intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification

Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Oracle if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY ORACLE. ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE

SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

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

LIMITATION OF LIABILITY

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

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

RESTRICTED RIGHTS LEGEND

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

REPORT

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

Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS

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

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

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

Contents

1	Introduction	1
1.1	Status	1
1.2	Goals	2
1.3	Non-Goals	2
1.4	Conventions	3
1.5	Terminology	3
1.6	Expert Group Members	4
1.7	Acknowledgements	4
2	Applications	5
2.1	Configuration	5
2.2	Validation	5
2.3	Publication	5
2.3.1	Java SE	5
2.3.2	Servlet	6
2.3.3	Other Container	7
3	Resources	9
3.1	Resource Classes	9
3.1.1	Lifecycle and Environment	9
3.1.2	Constructors	9
3.2	Fields and Bean Properties	10
3.3	Resource Methods	11
3.3.1	Visibility	11
3.3.2	Parameters	11
3.3.3	Return Type	11
3.3.4	Exceptions	12

3.3.5	HEAD and OPTIONS	13
3.4	URI Templates	13
3.4.1	Sub Resources	14
3.5	Declaring Media Type Capabilities	15
3.6	Annotation Inheritance	16
3.7	Matching Requests to Resource Methods	17
3.7.1	Request Preprocessing	17
3.7.2	Request Matching	17
3.7.3	Converting URI Templates to Regular Expressions	19
3.8	Determining the MediaType of Responses	20
4	Providers	21
4.1	Lifecycle and Environment	21
4.1.1	Constructors	21
4.2	Entity Providers	21
4.2.1	Message Body Reader	22
4.2.2	Message Body Writer	22
4.2.3	Declaring Media Type Capabilities	23
4.2.4	Standard Entity Providers	23
4.2.5	Transfer Encoding	24
4.2.6	Content Encoding	24
4.3	Context Providers	24
4.3.1	Declaring Media Type Capabilities	25
4.4	Exception Mapping Providers	25
5	Client API	27
5.1	Introduction	27
5.2	Request and Responses	27
5.3	Bootstrapping a Client Instance	27
5.4	Resource Access	28
5.5	Link, Invocation and HttpRequest	28
5.5.1	Building URIs	29
5.6	HTTP Methods	29
5.7	Typed Entities	29
5.8	Asynchronous Requests	30

5.9	Configurable Types	30
6	Filters and Handlers	33
6.1	Introduction	33
6.2	Filters	34
6.3	Handlers	35
6.4	Lifecycle	36
6.5	Binding	36
6.5.1	Static Binding	36
6.5.2	Global Binding	37
6.5.3	Dynamic Binding	37
6.5.4	Binding in Client API	37
6.6	Priorities	37
7	Validation	39
7.1	Introduction	39
7.2	Constraint Annotations	39
7.3	Annotations and Validators	41
7.4	Entity Validation	41
7.5	Annotation Inheritance	42
7.6	Validation Phases and Error Reporting	43
8	Context	45
8.1	Concurrency	45
8.2	Context Types	45
8.2.1	Application	45
8.2.2	URIs and URI Templates	45
8.2.3	Headers	46
8.2.4	Content Negotiation and Preconditions	46
8.2.5	Security Context	47
8.2.6	Providers	47
9	Environment	49
9.1	Servlet Container	49
9.2	Java EE Container	49
9.3	Other	50

10 Runtime Delegate	51
10.1 Configuration	51
A Summary of Annotations	53
B HTTP Header Support	55
C Filter and Handler Extension Points	57
D Change Log	59
D.1 Changes Since 1.0 Release	59
D.2 Changes Since Proposed Final Draft	59
D.3 Changes Since Public Review Draft	59
Bibliography	61

Chapter 1

Introduction

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

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

1.1 Status

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

http://java.net/jira/browse/JAX_RS_SPEC

The corresponding Javadocs can be found online at:

<http://java.net/projects/jax-rs-spec/sources>

The reference implementation can be obtained from:

<http://jersey.java.net/>

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

users@jax-rs-spec.java.net

1.2 Goals

The following are the goals of the API:

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

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

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

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

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

1.3 Non-Goals

The following are non-goals:

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

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

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

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

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

1.4 Conventions

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

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

Figure 1.1: Example Java Code

```

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

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

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

Note: *This is a note.*

1.5 Terminology

Resource class A Java class that uses JAX-RS annotations to implement a corresponding Web resource, see chapter 3.

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

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

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

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

Sub-resource method A method of a *resource class* that is used to handle requests on a sub-resource of the corresponding resource, see section 3.4.1.

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

1.6 Expert Group Members

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

Jan Algermissen (Individual Member)
Florent Benoit
Adam Bien
Bill Burke (Red Hat Middleware LLC)
Clint Combs
Bill De Hora (Individual Member)
Markus Karg (Individual Member)
Jeff Genender
Jerome Louvel
Tony Ng (Ebay)
Reza Rahman
Julian Reschke (Individual Member)
Guilherme Silveira
Dionysios Synodinos
Stefan Tilkov

1.7 Acknowledgements

Chapter 2

Applications

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

2.1 Configuration

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

2.2 Validation

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

2.3 Publication

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

2.3.1 Java SE

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

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

2.3.1.1 JAX-WS

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

2.3.2 Servlet

A JAX-RS application is packaged as a 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 the added servlet MUST be named:

`javax.ws.rs.core.Application`

and all root resource classes and providers packaged in the web application MUST be included in the published JAX-RS application. The application MUST be packaged with a `web.xml` that specifies a servlet mapping for the added servlet.

- If an `Application` subclass is present and there is already a servlet defined that has a servlet initialization parameter named:

`javax.ws.rs.Application`

whose value is the fully qualified name of the `Application` subclass then no new servlet should be added by the JAX-RS implementation's `ContainerInitializer` since the application is already being handled by an existing servlet.

- If an `Application` subclass is present that is not being handled by an existing servlet then the servlet added by the `ContainerInitializer` MUST be named with the fully qualified name of the `Application` subclass. If the `Application` subclass is annotated with `@ApplicationPath` and no servlet-mapping exists for the added servlet then a new servlet mapping is added with the value of the `@ApplicationPath` annotation with `"/"` appended otherwise the existing mapping is used. If the `Application` subclass is not annotated with `@ApplicationPath` then the application MUST be packaged with a `web.xml` that specifies a servlet mapping for the added servlet. It is an error for more than one application to be deployed at the same effective servlet mapping

In either of the latter two cases, if both `Application.getClasses` and `Application.getSingletons` return an empty list then all root resource classes and providers packaged in the web application MUST be included in the published JAX-RS application. If either `getClassess` or `getSingletons` return a non-empty list then only those classes or singletons returned MUST be included in the published JAX-RS application.

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`.

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 8 and 9 for more details.

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

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

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

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

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

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

3.3 Resource Methods

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

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

3.3.1 Visibility

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

3.3.2 Parameters

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

- The `DefaultValue` annotation may be used to supply a default value for parameters
- The `Encoded` annotation may be used to disable automatic URI decoding of parameter values
- Exceptions thrown during construction of parameter values are treated the same as exceptions thrown during construction of field or bean property values, see section 3.2. 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 non-annotated parameter, called the entity parameter, is mapped from the request entity body. Conversion between an entity body and a Java type is the responsibility of an entity provider, see section 4.2.

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

3.3.3 Return Type

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

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

`Response` Results in an entity body mapped from the entity property of the `Response` with the status code specified by the status property of the `Response`. A `null` return value results in a 204 status code.

If the status property of the `Response` is not set: a 200 status code is used for a non-null entity property and a 204 status code is used if the entity property is null.

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

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

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

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

Return Type	Returned Instance ¹	Raw Type	Generic Type
<code>GenericEntity</code>	<code>GenericEntity</code> or subclass	<code>RawType</code> property	<code>Type</code> property
<code>Response</code>	<code>GenericEntity</code> or subclass	<code>RawType</code> property	<code>Type</code> property
<code>Response</code>	<code>Object</code> or subclass	Class of instance	Class of instance
<code>Other</code>	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
<code>GenericEntity</code>	<code>GenericEntity<List<String>></code>	<code>ArrayList<?></code>	<code>List<String></code>
<code>Response</code>	<code>GenericEntity<List<String>></code>	<code>ArrayList<?></code>	<code>List<String></code>
<code>Response</code>	<code>ArrayList<String></code>	<code>ArrayList<?></code>	<code>ArrayList<?></code>
<code>List<String></code>	<code>ArrayList<String></code>	<code>ArrayList<?></code>	<code>List<String></code>

Table 3.2: Example raw and generic types of return values

3.3.4 Exceptions

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

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

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

2. If an exception mapping provider (see section 4.4) is available for the exception or one of its super-classes, 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 **MUST** be re-thrown and allowed to propagate to the underlying container.
4. Checked exceptions and throwables that cannot be thrown directly **MUST** be wrapped in a container-specific exception that is then thrown and allowed to propagate to the underlying container. Servlet-based implementations **MUST** use `ServletException` as the wrapper. JAX-WS Provider-based implementations **MUST** use `WebServiceException` as the wrapper.

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

3.3.5 HEAD and OPTIONS

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

1. Call a method annotated with a request method designator for HEAD or, if none present,
2. Call a method annotated with a request method designator for GET and discard any returned entity.

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

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

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

3.4 URI Templates

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

```

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

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:

```
1  @Path("widget_list/{id}")
2  @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.:

```
1  @Path("widgets/{path:.+}")
2  public class Widget {
3      ...
4  }
```

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

3.4.1 Sub Resources

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

Present Such methods, known as *sub-resource methods*, are treated like a normal resource method (see section 3.3) except the method is only invoked for request URIs that match a URI template created by concatenating the URI template of the resource class with the URI template of the method².

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

The following example illustrates the difference:

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


```

1  @Path("widgets")
2  public class WidgetsResource {
3      @GET
4      @Path("offers")
5      public WidgetList getDiscounted() {...}
6
7      @Path("{id}")
8      public WidgetResource findWidget(@PathParam("id") String id) {
9          return new WidgetResource(id);
10     }
11 }
12
13 public class WidgetResource {
14     public WidgetResource(String id) {...}
15
16     @GET
17     public Widget getDetails() {...}
18 }

```

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

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

3.5 Declaring Media Type Capabilities

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

The following example illustrates the use of these annotations:

```

1  @Path("widgets")
2  @Produces("application/widgets+xml")
3  public class WidgetsResource {
4
5      @GET
6      public WidgetList getAsXML() {...}
7
8      @GET
9      @Produces("text/html")
10     public String getAsHtml() {...}
11
12     @POST
13     @Consumes("application/widgets+xml")
14     public void addWidget(Widget widget) {...}
15 }
16

```

```
17 @Provider
18 @Produces("application/widgets+xml")
19 public class WidgetsProvider implements MessageBodyWriter<Widgets> {...}
20
21 @Provider
22 @Consumes("application/widgets+xml")
23 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.

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 method and its parameters do not have any JAX-RS annotations of its 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 *undefined*.

If a subclass or implementation method has any JAX-RS annotations then *all* of the annotations on the super class or interface method are ignored. E.g.:

```
1 public interface ReadOnlyAtomFeed {
2     @GET @Produces("application/atom+xml")
3     Feed getFeed();
4 }
5
6 @Path("feed")
7 public class ActivityLog implements ReadOnlyAtomFeed {
8     public Feed getFeed() {...}
9 }
```

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

```

1  @Path("feed")
2  public class ActivityLog implements ReadOnlyAtomFeed {
3      @Produces("application/atom+xml")
4      public Feed getFeed() {...}
5  }

```

In the above, the `@GET` annotation on `ReadOnlyAtomFeed.getFeed` is not inherited by `ActivityLog.getFeed` and it would require its own request method designator since it redefines the `@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 normalized³ by following the rules for case, path segment, and percent encoding normalization described in section 6.2.2 of RFC 3986[5]. The normalized request URI **MUST** be reflected in the URIs obtained from an injected `UriInfo`.

3.7.2 Request Matching

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

1. Identify the root resource class:

- (a) Set U = request URI path, C = {root resource classes}, E = {}
- (b) For each class in C add a regular expression (computed using the function $R(A)$ described in section 3.7.3) to E as follows:
 - Add $R(T_{\text{class}})$ where T_{class} is the URI path template specified for the class.
- (c) Filter E by matching each member against U as follows:
 - Remove members that do not match U .
 - Remove members for which the final regular expression capturing group (henceforth simply referred to as a capturing group) value is neither empty nor `'/'` and the class associated with $R(T_{\text{class}})$ had no sub-resource methods or locators.

³Note: some containers might perform this functionality prior to passing the request to an implementation.

- (d) If E is empty then no matching resource can be found, the algorithm terminates and an implementation **MUST** generate a `WebApplicationException` with a not found response (HTTP 404 status) and no entity. The exception **MUST** be processed as described in section 3.3.4.
 - (e) Sort E using the number of literal characters⁴ 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 , set U to be the value of the final capturing group of R_{match} when matched against U , and instantiate an object O of the associated class.
2. Obtain the object that will handle the request and a set of candidate methods:
- (a) If U is null or `'/'`, set

$$M = \{\text{resource methods of } O \text{ (excluding sub resource methods)}\}$$
 and go to step 3
 - (b) Set $C = \text{class of } O$, $E = \{\}$
 - (c) For class C add regular expressions to E for each sub-resource method and locator as follows:
 - i. For each sub-resource method, add $R(T_{\text{method}})$ where T_{method} is the URI path template of the sub-resource method.
 - ii. For each sub-resource locator, add $R(T_{\text{locator}})$ where T_{locator} is the URI path template of the sub-resource locator.
 - (d) Filter E by matching each member against U as follows:
 - Remove members that do not match U .
 - Remove members derived from T_{method} (those added in step 2(c)i) for which the final capturing group value is neither empty nor `'/'`.
 - (e) If E is empty then no matching resource can be found, the algorithm terminates and an implementation **MUST** generate a `WebApplicationException` with a not found response (HTTP 404 status) and no entity. The exception **MUST** be processed as described in section 3.3.4.
 - (f) Sort E using the number of literal characters in each member as the primary key (descending order), the number of capturing groups as a secondary key (descending order), the number of capturing groups with non-default regular expressions (i.e. not `'([^\/?]+?)'`) as the tertiary key (descending order), and the source of each member as quaternary key sorting those derived from T_{method} ahead of those derived from T_{locator} .
 - (g) Set R_{match} to be the first member of E
 - (h) If R_{match} was derived from T_{method} , then set

$$M = \{\text{subresource methods of } O \text{ where } R(T_{\text{method}}) = R_{\text{match}}\}$$
 and go to step 3.
 - (i) Set U to be the value of the final capturing group of $R(T_{\text{match}})$ when matched against U , invoke the sub-resource locator method of O and set O to the value returned from that method.
 - (j) Go to step 2a.
3. Identify the method that will handle the request:

⁴Here, literal characters means those not resulting from template variable substitution.

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

- The request method is supported. If no methods support the request method an implementation **MUST** generate a `WebApplicationException` with a method not allowed response (HTTP 405 status) and no entity. The exception **MUST** be processed as described in section 3.3.4. Note the additional support for `HEAD` and `OPTIONS` described in section 3.3.5.
- The media type of the request entity body (if any) is a supported input data format (see section 3.5). If no methods support the media type of the request entity body an implementation **MUST** generate a `WebApplicationException` with an unsupported media type response (HTTP 415 status) and no entity. The exception **MUST** be processed as described in section 3.3.4.
- At least one of the acceptable response entity body media types is a supported output data format (see section 3.5). If no methods support one of the acceptable response entity body media types an implementation **MUST** generate a `WebApplicationException` with a not acceptable response (HTTP 406 status) and no entity. The exception **MUST** be processed as described in section 3.3.4.

(b) Sort M in descending order as follows:

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

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

(c) The request is dispatched to the first Java method in the set⁵.

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 `'([^\s]+?)'` if no regular expression is specified.
4. If the resulting string ends with `'/'` then remove the final character.
5. Append `'(/.*)?'` to the result.

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

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

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 $P = \{‘*/’\}$
4. Obtain the acceptable media types A . If $A = \{\}$, set $A = \{‘*/’\}$
5. Set $M = \{\}$. For each member of A , a :
 - For each member of P , p :
 - If a is compatible with p , add $S(a, p)$ to M , where the function S returns the most specific media type of the pair with the q-value of a .
6. If $M = \{\}$ then generate a `WebApplicationException` with a not acceptable response (HTTP 406 status) and no entity. The exception MUST be processed as described in section 3.3.4. Finish.
7. Sort M in descending order, with a primary key of specificity ($n/m > n/* > */*$) and secondary key of q-value.
8. For each member of M , m :
 - If m is a concrete type, set $M_{\text{selected}} = m$, finish.
9. If M contains ‘*/’ or ‘application/*’, set $M_{\text{selected}} = \text{‘application/octet-stream’}$, finish.
10. Generate a `WebApplicationException` with a not acceptable response (HTTP 406 status) and no entity. The exception MUST be processed as described in section 3.3.4. Finish.

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

Chapter 4

Providers

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

4.1 Lifecycle and Environment

By default a single instance of each provider class is instantiated for each JAX-RS application. First the constructor (see section 4.1.1) is called, then any requested dependencies are injected (see chapter 8), then the appropriate provider methods may be called multiple times (simultaneously), and finally the object is made available for garbage collection. Section 8.2.6 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 Constructors

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

A public constructor MAY include parameters annotated with `@Context`- chapter 8 defines the parameter types permitted for this annotation. Since providers may be created outside the scope of a particular request, only deployment-specific properties may be available from injected interfaces at construction time - request-specific properties are available when a provider method is called. If more than one public constructor can be used then an implementation MUST use the one with the most parameters. Choosing amongst constructors with the same number of parameters is implementation specific, implementations SHOULD generate a warning about such ambiguity.

4.2 Entity Providers

Entity providers supply mapping services between representations and their associated Java types. Entity providers come in two flavors: `MessageBodyReader` and `MessageBodyWriter` described below. In the

absence of a suitable entity provider, JAX-RS implementations are REQUIRED to use the JavaBeans Activation Framework[11] to try to obtain a suitable data handler to perform the mapping instead.

4.2.1 Message Body Reader

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

The following describes the logical¹ steps taken by a JAX-RS implementation when mapping a request 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.
5. If step 4 locates a suitable `MessageBodyReader` then use its `readFrom` method to map the entity body to the desired Java type.
6. Else if a suitable data handler can be found using the JavaBeans Activation Framework[11] then use it to map the entity body to the desired Java type.
7. Else generate a `WebApplicationException` that contains an unsupported media type response (HTTP 415 status) and no entity. The exception MUST be processed as described in section 3.3.4.

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

4.2.2 Message Body Writer

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

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

1. Obtain the object that will be mapped to the response entity body. For a return type of `Response` or subclasses the object is the value of the `entity` property, for other return types it is the returned object.

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

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 response entity body.
4. Sort the selected `MessageBodyWriter` providers with a primary key of media type (see section 4.2.3) and a secondary key of generic type where providers whose generic type is the nearest super-class of the object class are sorted first.
5. Iterate through the sorted `MessageBodyWriter` providers and, utilizing the `isWriteable` method of each, choose an `MessageBodyWriter` that supports the object that will be mapped to the entity body.
6. If step 5 locates a suitable `MessageBodyWriter` then use its `writeTo` method to map the object to the entity body.
7. Else if a suitable data handler can be found using the JavaBeans Activation Framework[11] then use it to map the object to the entity body.
8. Else generate a `WebApplicationException` with an internal server error response (HTTP 500 status) and no entity. The exception **MUST** be processed as described in section 3.3.4.

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

4.2.3 Declaring Media Type Capabilities

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

When choosing an entity provider an implementation sorts the available providers according to the media types they declare support for. Sorting of media types follows the general rule: $x/y < x/* < */*$, i.e. a provider that explicitly lists a media types is sorted before a provider that lists `*/*`.

4.2.4 Standard Entity Providers

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

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

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

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

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

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

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

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

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

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

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

When reading zero-length request entities, all implementation-supplied `MessageBodyReader` implementations except the JAXB-related one **MUST** create a corresponding Java object that represents zero-length data; they **MUST NOT** return null. The implementation-supplied JAXB `MessageBodyReader` implementation **MUST** throw a `WebApplicationException` with a client error response (HTTP 400) for zero-length request entities.

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

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

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

4.2.5 Transfer Encoding

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

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

4.2.6 Content Encoding

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

4.3 Context Providers

Context providers supply context to resource classes and other providers. A context provider class implements the `ContextResolver<T>` interface and is annotated with `@Provider`. E.g. an application wishing

to provide a customized `JAXBContext` to the default JAXB entity providers would supply a class implementing `ContextResolver<JAXBContext>`.

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

4.3.1 Declaring Media Type Capabilities

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

When choosing a context provider an implementation sorts the available providers according to the media types they declare support for. Sorting of media types follows the general rule: `x/y < x/* < */*`, i.e. a provider that explicitly lists a media type is sorted before a provider that lists `*/*`.

4.4 Exception Mapping Providers

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

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

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

Chapter 5

Client API

5.1 Introduction

This chapter introduces the reader to the Client API. This API can be used to access Web resources. Unless otherwise stated, types presented in this chapter live in the `javax.ws.rs.client` package.

5.2 Request and Responses

The JAX-RS 1.X types `javax.ws.rs.core.Request` and `javax.ws.rs.core.Response` have been designed specifically for the Server API. These classes do not provide the necessary support to satisfy all the use cases covered by the Client API. Two new interfaces, `javax.ws.rs.core.HttpRequest` and `javax.ws.rs.core.HttpResponse`, are hereby introduced to accommodate the new set of requirements. As indicated by their location in the core package, their scope is *not* limited to the Client API.

5.3 Bootstrapping a Client Instance

An instance of `Client` is required to access a Web resource using the Client API. Instances of `Client` can be obtained using `ClientFactory` and, optionally, an instance of `ClientConfiguration`. For example,

```
1 // Default client using default configuration
2 Client client = ClientFactory.newClient();
3
4 // Default client using custom configuration
5 ClientConfiguration cfg = new DefaultClientConfiguration();
6 cfg.enable("CUSTOM_FEATURE");
7 Client confClient = ClientFactory.newClient(cfg);
```

The example above demonstrates the creation of `Client` instances using the default `ClientBuilderFactory`. Further customization is possible by providing a different implementation of `ClientBuilderFactory` and, optionally, a different implementation of `ClientConfiguration` that is specific to that factory. In the following example, a custom configuration is used to create a custom client that supports caching:

```
1 // Custom client using custom configuration
```

```
2 MyClientConfiguration myCfg = new MyClientConfiguration();
3 myCfg.enableCaching();
4 MyClient customConfClient =
5     ClientFactory.newClientBy(MyClientBuilderFactory.class).create(myCfg);
```

Note the call to the method `enableCaching` which is part of the extended configuration defined by `MyClientConfiguration`, and supported by `MyClientBuilderFactory`.

5.4 Resource Access

The Client API provides a few different ways in which to access a Web resource. First, a resource can be accessed using the fluent method chaining approach whereby methods are called to configure a request using the builder pattern. The following example gets a `text/plain` representation of the resource identified by the `http://example.org/hello` URI:

```
1 Client client = ClientFactory.newClient();
2 HttpResponse res = client.link("http://example.org/hello")
3     .get().accept("text/plain").invoke();
```

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

```
1 Client client = ClientFactory.newClient();
2 HttpResponse res = client.link("http://example.org/hello")
3     .get().accept("text/plain").header("MyHeader", "...")
4     .queryParams("MyParam", "...").invoke();
```

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

5.5 Link, Invocation and HttpRequest

A `Link` is a builder for instances of type `Invocation`. An `Invocation` is an *invocable* version of a `core.HttpRequest`. The following example shows the relationships between these types:

```
1 // Get a Link from a URI
2 Link uri = client.link("http://example.org/");
3 // Get an Invocation by specifying an HTTP method
4 Invocation inv = uri.get();
5 // Upcast to HttpRequest
6 HttpRequest req = inv;
```

The minimum requirement to obtain an instance of `Invocation` is to provide a URI and an HTTP method. Even though instances of `core.HttpRequest` are not directly invocable, they can be used to access Web resources when passed as parameters to the `Client.invoke` (synchronous) and `Client.queue` (asynchronous) method families.

Instances of `core.HttpRequest` can also be created directly from `Client` without the need to create an `Invocation` first by calling `request` instead of `link`:

```

1 // Get an HttpRequest from client
2 HttpRequest req = client.request("http://example.org/").get();
3 HttpResponse res = client.invoke(req);

```

5.5.1 Building URIs

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

```

1 Link base = client.link("http://example.org/");
2 Link hello = base.path("hello").path("{whom}");
3 HttpResponse res = hello.pathParam("whom", "world").get().invoke();

```

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

5.6 HTTP Methods

The Client API provides built-in support for configuring the following HTTP methods in a request: GET, PUT, POST, DELETE, HEAD, OPTIONS and TRACE. Additional verbs can be specified using a generic method that takes a string. For example, a WebDAV client could configure a request as follows:

```

1 Link file = client.link("http://examples.org/index.html");
2 HttpRequest copyRequest = file.method("COPY")
3     .header("Destination", "http://examples.org/backup/index.html")
4     .header("Overwrite", "T");

```

5.7 Typed Entities

The response to an invocation is not limited to be of type `core.HttpResponse`. It is possible to request a specific entity response type when calling `Invocation.invoke` (or `Client.invoke`) as follows:

```

1 Customer c = client.link("http://examples.org/customers/123").get()
2     .accept("application/xml").invoke(Customer.class);
3 String newId = client.link("http://examples.org/gold-customers/")
4     .post().type("application/xml").entity(c).invoke(String.class);

```

In the example above, just like in the Server API, JAX-RS implementations are **REQUIRED** to use registered providers (implementing `MessageBodyReader` and `MessageBodyWrite`) 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.

5.8 Asynchronous Requests

All the examples shown thus far execute synchronous invocations: the calling thread is suspended until the response is returned from the server. There are numerous use cases for which this form of processing is insufficient. The Client API supports asynchronous invocations via the `Client.queue` and `Invocation.queue` method families. In either case, the calling thread is free to continue its normal execution while the server is processing the request.

Two processing models are supported. In the first, a request is queued and an instance of `Future<T>` (for some fixed `T`) is returned; in the second, an instance of `InvocationCallback<S>` (for some fixed `S`) is registered as a callback object and an instance of `Future<S>` is returned as a result of the queue operation.

```
1  Invocation copyRequest = ...;
2
3  // (1) Get a Future object for this request
4  Future<HttpResponse> futureRes = copyRequest.queue();
5  // ... do something ...
6  if (futureRes.isDone()) {
7      HttpResponse res = futureRes.get();
8      // ...
9  } else {
10     futureRes.cancel(true);           // taking too long!
11 }
12
13 // (2) Get a Future and register an InvocationCallback
14 futureRes = copyRequest.queue(new InvocationCallback<HttpResponse>() {
15     @Override
16     public void onComplete(Future<HttpResponse> futureRes) {
17         HttpResponse res = futureRes.get();
18         // ... do something ...
19     }
20 });
```

In either processing model, the asynchronous invocation can be controlled by using the `Future` object returned as part of the queueing process. Namely, its status can be checked and it can be cancelled. In the second processing model, the method `InvocationCallback.onComplete` is called when the asynchronous operation terminates. An asynchronous operation terminates upon receiving a response, detecting an error or as a result of calling `cancel` on the `Future` returned at queueing time. Hence, the use of `Future` in the parameter of `InvocationCallback.onComplete`.

Typed entities are supported just like in the synchronous case, i.e. by specifying the type of the expected entity as a parameter:

```
1  Future<Customer> fc = client.link("http://examples.org/customers/123")
2      .get().accept("application/xml").queue(Customer.class);
3  // ... do something ...
4  Customer c = fc.get();
```

5.9 Configurable Types

The Client API types `Client`, `ClientConfiguration`, `Link` and `Invocation` all implement the `Configurable` interface. This interface supports configuration of:

Features Identified by unique strings; can be enabled or disabled in order to configure a JAX-RS implementation.

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

Providers Classes or singleton instances of classes annotated by `@Provider`. A provider can be a message body reader, a filter, a context provider, etc. See Chapter 4 for more information.

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

Chapter 6

Filters and Handlers

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

6.1 Introduction

Handlers 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 two extension points for filters: Pre and Post. There are two extension points for handlers: ReadFrom and WriteTo. For each of these extension points, there is a corresponding interface:

```
1 public interface RequestFilter {
2     FilterAction preFilter(FilterContext ctx) throws IOException;
3 }
4 public interface ResponseFilter {
5     FilterAction postFilter(FilterContext ctx) throws IOException;
6 }

1 public interface ReadFromHandler<T> {
2     T readFrom(ReadFromHandlerContext<T> context) throws IOException;
3 }
4 public interface WriteToHandler<T> {
5     void writeTo(WriteToHandlerContext<T> context) throws IOException;
6 }
```

A filter class is a class that implements `RequestFilter` or `ResponseFilter` (or both) and is annotated by `@Provider`. A handler class is a class that implements `ReadFromHandler` or `WriteToHandler` (or both) and is annotated by `@Provider`.

In the Client API, filters implementing `RequestFilter` **MUST** be executed *before* the HTTP invocation and *before* all handlers implementing `WriteToHandler`; filters implementing `ResponseFilter` **MUST** be executed *after* the HTTP invocation and *before* all handlers implementing `ReadFromHandler`. In the Server API, filters implementing `RequestFilter` **MUST** be executed *before* the resource method is called and *before* all handlers implementing `ReadFromHandler`; filters implementing `ResponseFilter` **MUST** be executed *after* the HTTP invocation and *before* all handlers implementing `WriteToHandler`.

Editors Note 6.1 *If filters can read or write entities, the rules above would be violated. For example, a server request filter could call `ctx.getRequest().getEntity(Customer.class)` forcing the `ReadFrom` handlers to be called.*

A handler implementing `ReadFromHandler` wraps around calls to `MessageBodyReader.readFrom`. A handler implementing `WriteToHandler` wraps around calls to `MessageBodyWrite.writeTo`. JAX-RS implementations are **REQUIRED** to call registered handlers when mapping representations to Java types and vice versa. See Section 4.2 for more information on entity providers.

6.2 Filters

As stated above, a filter implements interface `RequestFilter` or `ResponseFilter` or both. Multiple filters are grouped in *filter chains*. Filters in a chain are sorted based on their priorities (see Section 6.6) and are executed in order.

A call to a filter's `preFilter` or `postFilter` methods returns a `FilterAction` which is an enumeration of two values: `STOP` and `NEXT`. If a filter returns `NEXT`, implementations are **REQUIRED** to proceed with the rest of the filter chain; if a filter returns `STOP`, implementations are **REQUIRED** to abort execution of the filter chain.

The following example shows an implementation of a logging filter: each method simply logs the message and returns `NEXT` to continue with the remainder of the filter chain.

```
1  @Provider
2  public class LoggingFilter implements RequestFilter, ResponseFilter {
3
4      @Override
5      public FilterAction preFilter(FilterContext ctx) throws IOException {
6          logRequest(ctx.getRequest());
7          return FilterAction.NEXT;
8      }
9
10     @Override
11     public FilterAction postFilter(FilterContext ctx) throws IOException {
12         logResponse(ctx.getResponse());
13         return FilterAction.NEXT;
14     }
15     ...
16 }
```

A `FilterContext` provides access to the request, the response (if available) as well as the ability to create and set new responses. Once the execution of a filter chain is completed, either by reaching the end of the chain or due to a filter returning `STOP`, JAX-RS implementations **MUST** get the response returned by `FilterContext.getResponse`. If this method returns null, normal execution is resumed; otherwise, the response returned is used and the HTTP invocation (Client API) or the resource method invocation (Server API) is omitted. See Appendix C for more information.

6.3 Handlers

A handler implements interface `ReadFromHandler` or `WriteToHandler` or both. Multiple handlers are grouped in *handler chains*. Handlers in a chain are sorted based on their priorities (see Section 6.6) and are executed in order.

Handlers wrap calls to the methods `MessageBodyReader.readFrom` or `MessageBodyWrite.writeTo`. Handlers **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 handler that provides deflate and inflate capabilities ¹

```

1  @Provider
2  public class GzipHandler implements ReadFromHandler, WriteToHandler {
3
4      @Override
5      public Object readFrom(ReadFromHandlerContext ctx) throws IOException {
6          InputStream old = ctx.getInputStream();
7          ctx.setInputStream(new GZIPInputStream(old));
8          try {
9              return ctx.proceed();
10         } finally {
11             ctx.setInputStream(old);
12         }
13     }
14
15     @Override
16     public void writeTo(WriteToHandlerContext ctx) throws IOException {
17         OutputStream old = ctx.getOutputStream();
18         GZIPOutputStream gzipOutputStream = new GZIPOutputStream(old);
19         ctx.setOutputStream(gzipOutputStream);
20         try {
21             ctx.proceed();
22         } finally {
23             gzipOutputStream.finish();
24             ctx.setOutputStream(old);
25         }
26     }
27     ...
28 }

```

The context types, `ReadFromHandlerContext` and `WriteToHandlerContext`, 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 `MessageBodyWrite.writeTo`.

¹This class is not intended to be a complete implementation of a GZIP handler.

6.4 Lifecycle

By default, just like all the other providers, a single instance of each filter or handler 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 handler or filter is associated with a resource class or method (Server API) or an HTTP invocation (Client API). The forms of binding presented in the next sections (static, global and dynamic) are only supported as part of the Server API. See Section 6.5.4 for binding in the Client API.

6.5.1 Static Binding

A handler or filter can be associated with a resource class or method by declaring a new *binding* annotation à la CDI [12]. These annotations are declared using the JAX-RS meta-annotation `@NameBinding` and are used to decorate both the filter (or handler) and the resource method (or class). For example, the `LoggingFilter` defined in Section 6.2 can be bound to `MyResourceClass.hello` as follows:

```
1  @Provider
2  @Logged
3  public class LoggingFilter implements RequestFilter, ResponseFilter {
4      ...
5  }

1  @Path("/")
2  public class MyResourceClass {
3      @Logged
4      @GET
5      @Produces("text/plain")
6      @Path("{name}")
7      public String hello(@PathParam("name") String name) {
8          return "Hello " + name;
9      }
10 }
```

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

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

Binding annotations that decorate resource classes apply to all the resource methods defined in them. Naturally, multiple binding annotations are permitted per class or method. The order in which the corresponding filters or handlers will be executed depends on their priorities (see Section 6.6).

6.5.2 Global Binding

Occasionally, it is useful to declare a handler or filter as being bound to all the resource classes in an application. This can be accomplished by using the `@GlobalBinding` annotation directly on the handler or filter class without the need to declare a new binding annotation. For example, the `LoggingFilter` defined in Section 6.2 can be defined as follows:

```
1  @Provider
2  @GlobalBinding
3  public class LoggingFilter implements RequestFilter, ResponseFilter {
4      ...
5  }
```

If this filter is registered as part of an application, requests and responses will be logged for all resource methods.

6.5.3 Dynamic Binding

A handler or filter that implements the `DynamicBinding` interface is bound to a resource method at runtime. Binding is determined based on the value returned by the `isBound` method, regardless of whether the handler or filter is decorated with `@GlobalBinding` or a binding annotation derived from `@NameBinding`. That is, if annotated with `@GlobalBinding`, the `isBound` method will be called for every resource method in the application; if annotated with a binding annotation derived from `@NameBinding`, it will be called for every resource method for which it was bound via that annotation.

6.5.4 Binding in Client API

Binding in the Client API is accomplished via API calls instead of annotations. More specifically, filters and handlers are registered like any other provider by calling `Configurable.register` methods. `Client`, `ClientConfiguration`, `Link` and `Invocation` all implement the `Configurable` interface. See Section 5.9 for more information.

6.6 Priorities

The order in which handlers or filters are executed as part of their corresponding chains is controlled by the `@BindingPriority` annotation. Priorities are represented by integer numbers: the *lower* the number, the *higher* the priority. The default priority for all handlers and filters—when an instance of `@BindingPriority` is absent or is present but without any value specified—is `BindingPriority.USER`.

The `BindingPriority` class defines additional built-in priorities for security, header decorators, decoders and encoders. For example, the priority of authentication filter can be set as follows:

```
1  @Provider
2  @Authenticated
3  @BindingPriority(BindingPriority.SECURITY)
4  public class AuthenticationFilter implements RequestFilter {
5      ...
6  }
```

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 `@BindingPriority` annotation must be used for a filter or handler registered using the Client API. JAX-RS implementations are **REQUIRED** to sort filters and handlers based on `BindingPriority.value()`, using `BindingPriority.USER` as the default if the annotation is absent. The order in which filters or handlers that belong to the same priority class are executed is *undefined*.

Chapter 7

Validation

7.1 Introduction

Validation is the process of verifying that some data obeys one or more pre-defined constraints. The Bean Validation specification [13] defines an API to validate Java Beans. As described in this chapter, JAX-RS 2.0 provides native support for validating resource classes based on the concepts presented in [13].

7.2 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 binding 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 JAX-RS 1.X, any additional validation of these values would need to be performed programmatically. JAX-RS 2.0 introduces support for declarative validation based on the Bean Validation specification [13]. For example, consider the following resource class augmented with *constraint* annotations:

```
1  @Path("/")
2  class MyResourceClass {
3
4      @POST
5      @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
6      public void registerUser(
7          @NotNull @FormParam("firstName") String firstName,
8          @NotNull @FormParam("lastName") String lastName,
9          @Email @FormParam("email") String email) {
10         ...
11     }
12 }
```

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 property setters as we shall explain shortly. Rather than using method parameters, the `MyResourceClass` shown above could have been defined as follows:

```
1  @Path("/")
2  class MyResourceClass {
3
4      @NotNull @FormParam("firstName")
5      private String firstName;
6
7      @NotNull @FormParam("lastName")
8      private String lastName;
9
10     private String email;
11
12     @FormParam("email")
13     public void setEmail(@Email String email) {
14         this.email = email;
15     }
16
17     ...
18 }
```

Note that in this version, `firstName` and `lastName` are fields and `email` is a resource class property. Constraint annotations on property setters **MUST** be specified in the parameter instead of the method as seen in the example. This is the only case in which a JAX-RS binding annotation is not adjacent to a constraint annotation.

Editors Note 7.1 *Should JAX-RS 2.0 allow binding annotations on setter parameters in addition to setter methods to lift this restriction while maintaining backward compatibility?. In hindsight, JAX 1.X should have annotated the parameter instead of the setter method.*

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 either of the two *name* fields in `MyResourceClass` are not empty. Using such an annotation, the example above can be written follows:

```
1  @Path("/")
2  @NonEmptyNames
3  class MyResourceClass {
4
5      @NotNull @FormParam("firstName")
6      private String firstName;
7
8      @NotNull @FormParam("lastName")
9      private String lastName;
10
11     private String email;
12
13     @FormParam("email")
14     public void setEmail(@Email String email) {
15         this.email = email;
16     }
17     ...
18 }
```

Constraint annotations on resource classes are useful for defining cross-field and cross-property constraints. The order in which these validation steps take place is explained in Section ??.

7.3 Annotations and Validators

Annotation constraints and validators are defined in accordance with the Bean Validation specification [13]. The `@Email` annotation shown above is defined using the Bean Validation `@Constraint` meta-annotation:

```

1  @Target( { METHOD, FIELD, PARAMETER })
2  @Retention(RUNTIME)
3  @Constraint(validatedBy = EmailValidator.class)
4  public @interface Email {
5      String message() default "{foo.bar.validation.constraints.email}";
6      Class<?>[] groups() default {};
7      Class<? extends Payload>[] payload() default {};
8  }

```

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

```

1  public class EmailValidator implements ConstraintValidator<Email, String> {
2      public void initialize(Email email) {
3          ...
4      }
5
6      public boolean isValid(String value, ConstraintValidatorContext context) {
7          ...
8      }
9  }

```

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.

7.4 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`:

```

1  @CheckUser1
2  class User { ... }
3
4  @Path("/")
5  class MyResourceClass {
6
7      @POST
8      @Consumes("application/xml")
9      public void registerUser(@Valid User user) {

```

```
10      ...
11    }
12 }
```

In this case, the validator associated with `@CheckUser1` 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.

```
1  @Path("/")
2  class MyResourceClass {
3
4      @POST
5      @Consumes("application/xml")
6      public void registerUser(@CheckUser2 User user) {
7          ...
8      }
9  }
```

In the example above, `@CheckUser2` rather than `@CheckUser1` 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.

Response entity bodies returned from resource methods can be validated in a similar manner by annotating the resource method itself. To exemplify, assuming both `@CheckUser1` and `@CheckUser2` are required to be checked before returning a user, the `getUser` method can be annotated as shown next:

```
1  @Path("/")
2  class MyResourceClass {
3
4      @GET
5      @Path("{id}")
6      @Produces("application/xml")
7      @Valid @CheckUser2
8      public User getUser(@PathParam("id") String id) {
9          User u = findUser(id);
10         return u;
11     }
12     ...
13 }
```

Note that `@CheckUser2` is explicitly listed and `@CheckUser1` is triggered by the presense of the `@Valid` annotation —see definition of `User` class earlier in this section.

7.5 Annotation Inheritance

The rule for inheritance of constraint annotations is the same as that for all the other JAX-RS annotations (see Section 3.6). Namely, constraint annotations on methods and method parameters are inherited from interfaces and super-classes, with the latter taking precedence over the former when sharing common methods. For example:

```

1  interface MyInterface {
2      @GET
3      @Path("{id}")
4      @Produces("application/xml")
5      @CheckUser1
6      public User getUser(@Pattern("[0-9]+") @PathParam("id") String id);
7  }
8
9  @Path("/")
10 class MyResourceClass implements MyInterface {
11
12      public User getUser(String id) {
13          User u = findUser(id);
14          return u;
15      }
16      ...
17  }

```

In the example above, the constraint annotations `@CheckUser1` and `@Pattern` will be inherited by the `getUser` method in `MyResourceClass`. If the `getUser` method in `MyResourceClass` is decorated with any annotations, constraint or otherwise, all of the annotations in the interface `MyInterface` will be ignored. Naturally, since fields in super-classes that are visible in subclasses cannot be overridden, all their annotations (including their constraint annotations) are inherited.

7.6 Validation Phases and Error Reporting

Constraint annotations are allowed in the same locations as the following annotations: `@MatrixParam`, `@QueryParam`, `@PathParam`, `@CookieParam`, `@HeaderParam` and `@Context`. Namely, in public constructor parameters, method parameters, fields and bean properties. In addition, they can also decorate resource classes, entity parameters and resource methods. Constraint annotations on bean properties are only allowed on setter parameters and are checked exactly once when the resource class is instantiated.

In sub-resource classes, whose instances are returned by sub-resource locators, constraint annotations follow the same restrictions as other annotations. Namely, as stated in Section 3.2, instances returned by sub-resource locators are expected to be initialized by their creator and field and bean properties are not modified by the JAX-RS implementation. As a general rule, JAX-RS implementations are only **REQUIRED** to check validation constraints on the values that they modify. It follows that constraint annotations are *not* supported on sub-resource classes fields, properties and constructors, but only in methods.

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 annotations in resource class apply to constraint annotations. For example, a constraint validation annotating a constructor parameter in a resource class whose lifecycle is singleton (per application) will only be executed once.

When processing a request, is it often desirable to collect and return as many violations as possible rather than aborting execution after the first violation is encountered. JAX-RS implementations are **REQUIRED** to use the following process to validate root resource class instances in the per-request lifecycle:

Phase 1 Validate annotations on parameters passed to the resource class constructor.

Phase 2 Validate annotations on field injections and property setters as they are initialized and invoked, respectively.

Phase 3 Validate annotations on resource classes.

Phase 4 Validate annotations on parameters passed to the resource method selected for invocation.

The set of constraint violations is cumulative from phase 1 to phase 4. If after phase 4 the set of constraint violations is non-empty, implementations **MUST** not invoke the resource method but instead return a response with a status code 400 (Bad Request) and an entity that includes a description of all the violations encountered; the actual representation of such an entity is implementation dependent. If during any of these phases, an exception of type `java.lang.RuntimeException` is thrown, implementations **MUST** abort the validation process and return a response with a status code 400 (Bad Request) and an entity that includes a description of all the violations collected up to that point.

In summary, implementations must collect as many violations as possible until all phases are completed or an unrecoverable error is detected. Note, in particular, that constructors and property setters may be called and fields may be initialized even if the values passed as parameters or used as initializers are deemed invalid.

Chapter 8

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.

8.1 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 8.2, 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.

8.2 Context Types

This section describes the types of context available to resource classes, providers and `Application` subclasses.

8.2.1 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.

8.2.2 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:

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

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

8.2.3 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:

```
1  @GET
2  @Produces{"text/plain"}
3  public String listHeaderNames(@Context HttpHeaders headers) {
4      StringBuilder buf = new StringBuilder();
5      for (String header: headers.getRequestHeaders().keySet()) {
6          buf.append(header);
7          buf.append("\n");
8      }
9      return buf.toString();
10 }
```

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.

8.2.4 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:

```
1  @PUT
2  public Response updateFoo(@Context Request request, Foo foo) {
3      EntityTag tag = getCurrentTag();
4      ResponseBuilder responseBuilder = request.evaluatePreconditions(tag);
5      if (responseBuilder != null)
```



```
6         return responseBuilder.build();
7     else
8         return doUpdate(foo);
9 }
```

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

8.2.5 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.

8.2.6 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.

Chapter 9

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 8.2 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.

9.1 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.

9.2 Java EE Container

This section describes the additional requirements that apply to a JAX-RS implementation when combined in a product that supports these other Java specifications:

- In a product that also supports the Servlet specification, implementations **MUST** support JAX-RS applications that are packaged as a web application. Implementations **MUST** behave as if built using the Servlet 3 pluggability mechanism, see section 2.3.2.

- In a product that also supports Managed Beans, implementations **MUST** support use of Managed Beans as root resource classes, providers and `Application` subclasses. In a product that also supports JSR 299, implementations **MUST** similarly support use of JSR299-style managed beans. Providers and `Application` subclasses **MUST** be singletons or use application scope.
- In a product that also supports EJB, an implementation **MUST** support use of stateless and singleton session beans as root resource classes, providers and `Application` subclasses. JAX-RS annotations **MAY** be applied to a bean's local interface or directly to a no-interface bean. If an `ExceptionHandler` for a `EJBException` or subclass is not included with an application then exceptions thrown by an EJB resource class or provider method **MUST** be treated as EJB application exceptions: the embedded cause of the `EJBException` **MUST** be unwrapped and processed as described in section 3.3.4.

The following additional requirements apply when using Managed Beans, JSR299-style Managed 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.

9.3 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 8.2.

Chapter 10

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 10.1. 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.

10.1 Configuration

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

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

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

Appendix A

Summary of Annotations

Annotation	Target	Description
Consumes	Type or method	Specifies a list of media types that can be consumed.
Produces	Type or method	Specifies a list of media types that can be produced.
GET	Method	Specifies that the annotated method handles HTTP GET requests.
POST	Method	Specifies that the annotated method handles HTTP POST requests.
PUT	Method	Specifies that the annotated method handles HTTP PUT requests.
DELETE	Method	Specifies that the annotated method handles HTTP DELETE requests.
HEAD	Method	Specifies that the annotated method handles HTTP HEAD requests. Note that HEAD may be automatically handled, see section 3.3.5.
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.
FormParam	Parameter, field or method	Specifies that the value of a method parameter is to be extracted from a form parameter in a request entity body. The value of the annotation identifies the name of a form parameter. Note that whilst the annotation target allows use on fields and methods, the specification only requires support for use on resource method parameters.

Annotation	Target	Description
MatrixParam	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a URI matrix parameter. The value of the annotation identifies the name of a matrix parameter.
CookieParam	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP cookie. The value of the annotation identifies the name of a the cookie.
HeaderParam	Parameter, field or method	Specifies that the value of a method parameter, class field, or bean property is to be extracted from a HTTP header. The value of the annotation identifies the name of a HTTP header.
Encoded	Type, constructor, method, field or parameter	Disables automatic URI decoding for path, query, form and matrix parameters.
DefaultValue	Parameter, field or method	Specifies a default value for a field, property or method parameter annotated with @QueryParam, @MatrixParam, @CookieParam, @FormParam or @HeaderParam. The specified value will be used if the corresponding query or matrix parameter is not present in the request URI, if the corresponding form parameter is not in the request entity body, or if the corresponding HTTP header is not included in the request.
Context	Field, method or parameter	Identifies an injection target for one of the types listed in section 8.2 or the applicable section of chapter 9.
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.

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 8.2.3; response headers not listed here may be set using the `ResponseBuilder.header` method.

Header	Description
Accept	Used by runtime when selecting a resource method, compared to value of <code>@Produces</code> annotation, see section 3.5.
Accept-Charset	Processed by runtime if application uses <code>Request.selectVariant</code> method, see section 8.2.4.
Accept-Encoding	Processed by runtime if application uses <code>Request.selectVariant</code> method, see section 8.2.4.
Accept-Language	Processed by runtime if application uses <code>Request.selectVariant</code> method, see section 8.2.4.
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 <code>SecurityContext</code> , see section 8.2.5.
Cache-Control	See <code>CacheControl</code> class and <code>ResponseBuilder.cacheControl</code> method.
Content-Encoding	Response header set by application using <code>Response.ok</code> or <code>ResponseBuilder.variant</code> .
Content-Language	Response header set by application using <code>Response.ok</code> , <code>ResponseBuilder.language</code> , or <code>ResponseBuilder.variant</code> .
Content-Length	Processed automatically for requests, set automatically in responses if value is provided by the <code>MessageBodyWriter</code> used to serialize the response entity.
Content-Type	Request header used by runtime when selecting a resource method, compared to value of <code>@Consumes</code> annotation, see section 3.5. Response header either set by application using <code>Response.ok</code> , <code>ResponseBuilder.type</code> , or <code>ResponseBuilder.variant</code> , or set automatically by runtime (see section 3.8).
Cookie	See <code>Cookie</code> class and <code>HttpHeaders.getCookies</code> method.
Date	Included in responses automatically as per HTTP/1.1.

Header	Description
ETag	See <code>EntityTag</code> class, <code>Response.notModified</code> method and <code>ResponseBuilder.tag</code> method.
Expect	Depends on underlying container.
Expires	Set by application using the <code>ResponseBuilder.expires</code> method.
If-Match	Processed by runtime if application uses corresponding <code>Request.evaluatePreconditions</code> method, see section 8.2.4.
If-Modified-Since	Processed by runtime if application uses corresponding <code>Request.evaluatePreconditions</code> method, see section 8.2.4.
If-None-Match	Processed by runtime if application uses corresponding <code>Request.evaluatePreconditions</code> method, see section 8.2.4.
If-Unmodified-Since	Processed by runtime if application uses corresponding <code>Request.evaluatePreconditions</code> method, see section 8.2.4.
Last-Modified	Set by application using the <code>ResponseBuilder.lastModified</code> method.
Location	Set by application using the applicable <code>Response</code> method or directly using the <code>ResponseBuilder.location</code> method.
Set-Cookie	See <code>NewCookie</code> class and <code>ResponseBuilder.cookie</code> method.
Transfer-Encoding	See section 4.2.5.
Vary	Set by application using <code>Response.notAcceptable</code> method or <code>ResponseBuilder.variants</code> method.
WWW-Authenticate	Depends on container.

Appendix C

Filter and Handler Extension Points

The locations of the extension points `Pre`, `Post`, `ReadFrom` and `WriteTo` are easier to explain using some pseudo-code. The methods `clientSideRequestProcessor` and `serverSideRequestProcessor` represent the Client API and Server API frameworks, respectively. The `ReadFrom` extension point is *not* present in `clientSideRequestProcessor`: this extension point would be executed in response to the user application code requesting the entity response object.

Note how the response object is obtained from the `FilterContext` upon completion of the `Pre` and `Post` filter chains, and how the HTTP invocation (Client API) or the resource method invocation (Server API) is bypassed if a response has been set in that context object.

```
1  public HttpResponse clientSideRequestProcessor(HttpRequest req) {
2      HttpResponse res = null;
3
4      // Invoke Pre filters
5      FilterContext fc = newFilterContext(req, res);
6      for (RequestFilter f : getRequestFilters()) {
7          FilterAction action = f.preFilter(fc);
8          if (action == FilterAction.STOP) break;
9      }
10     res = fc.getResponse();
11
12     if (res == null) {
13         // Execute WriteTo handler chain
14         WriteToHandlerContext whc = newWriteToHandlerContext(req);
15         getFirstWriterToHandler(req).writeTo(whc);
16
17         // Actual HTTP request invocation
18         res = executeHttpRequest(req);
19     }
20
21     // Invoke Post filters
22     fc = newFilterContext(req, res);
23     for (ResponseFilter f : getResponseFilters()) {
24         FilterAction action = f.preFilter(fc);
25         if (action == FilterAction.STOP) break;
26     }
27     res = fc.getResponse();
28     return res;
29 }
```

```
1  public HttpServletResponse serverSideRequestProcessor(HttpServletRequest req, Method m) {
2      HttpServletResponse res = null;
3
4      // Invoke Pre filters
5      FilterContext fc = newFilterContext(req, res, m);
6      for (RequestFilter f : getRequestFilters()) {
7          FilterAction action = f.preFilter(fc);
8          if (action == FilterAction.STOP) break;
9      }
10     res = fc.getResponse();
11
12     if (res == null) {
13         // Execute Read handler chain
14         ReadFromHandlerContext rfc = newReadFromHandlerContext(req, m);
15         getFirstReadFromHandler(req).read(rfc);
16
17         // Actual resource method invocation
18         res = executeMethodInvocation(req, m);
19     }
20
21     // Invoke Post filters
22     fc = newFilterContext(req, res);
23     for (ResponseFilter f : getResponseFilters()) {
24         FilterAction action = f.preFilter(fc);
25         if (action == FilterAction.STOP) break;
26     }
27     res = fc.getResponse();
28
29     // Execute Write handler chain
30     WriteToHandlerContext whc = newWriteToHandlerContext(req, res, m);
31     getFirstWriteToHandler(m).write(whc);
32
33     return res;
34 }
```

Appendix D

Change Log

D.1 Changes Since 1.0 Release

- Section 2.3.2: new requirements for Servlet 3 containers.
- Section 9.2: requirements for Java EE 6 containers.
- Section 4.2.4: requirements on standard entity providers when presented with an empty request 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 8.2.5: fix typo.
- Section 9.1: additional considerations related to filters consuming request bodies.

D.2 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 10: Clarify that an implementation can supply an alternate `RuntimeDelegate` API class.

D.3 Changes Since Public Review Draft

- Chapter 2: Renamed `ApplicationConfig` class to `Application`.
- Chapter 3: `UriBuilder` reworked to always encode components.

- Sections 3.1.2 and 4.1.1: Added requirement to warn when choice of constructor is ambiguous.
- Section 3.2: `FormParam` no longer required to be supported on fields or properties.
- Section 3.3.3: Added text describing how to determine raw and generic types from method return type and returned instance.
- Section 3.4: Template parameters can specify the regular expression that forms their capturing group.
- Section 3.7.1: Make pre-processed URIs available rather than original request URI. Added URI normalization.
- Section 3.7.1: Removed URI-based content negotiation.
- Section 3.7.2: Reorganized the request matching algorithm to remove redundancy and improve readability, no functional change.
- Section 3.7.3: Changes to regular expressions to eliminate edge cases.
- Section 4.2: Added requirement to use JavaBean Activation Framework when no entity provider can be found.
- Section 4.2.4: Require standard JAXB entity providers to use application-supplied JAXB contexts in preference to their own.
- Section 4.3: Added support for specifying media type capabilities of context providers.
- Section 8.2: Removed `ContextResolver` from list of injectable resources.
- Section 8.2.6: 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] Bill Shannon. JavaBeans Activation Framework. JSR, JCP, May 2006. See <http://jcp.org/en/jsr/detail?id=925>.
- [12] Gavin King. Context and Dependency Injection for the Java Platform. JSR, JCP, December 2009. See <http://jcp.org/en/jsr/detail?id=299>.
- [13] Technical report.