MVC 1.0

Model-View-Controller Specification

Ivar Grimstad, Christian Kaltepoth

Version 1.0 Proposed Final Draft, September 2018

Table of Contents

License

This specification is dual licensed under the JCP License and the Apache 2.0 License.

JCP License

IVAR GRIMSTAD 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: JSR-371 MVC ("Specification")
Version: 1.0
Status: Pre-FCS Public Release
Release: Proposed Final Draft Release

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 Ivar Grimstad 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, 2 and 3 below, Ivar Grimstad hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Ivar Grimstad'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 Specification.
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."

3. Distribute applications written to the Specification to third parties for their testing and evaluation use, provided that any such application includes the following notice: "This is an application written to interoperate with 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 implementation upon final completion of the Specification. If you fail to do so, the foregoing grant shall be considered null and void. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Ivar Grimstad 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 Ivar Grimstad 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.[Specification Lead]" or their equivalents in any subsequent naming convention adopted 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 Ivar Grimstad or Ivar Grimstad’s licensors is granted hereunder. Java and Java-related logos, marks and names are trademarks or registered trademarks of Oracle America, 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 IVAR GRIMSTAD. IVAR GRIMSTAD 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. IVAR GRIMSTAD 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 IVAR GRIMSTAD 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 IVAR GRIMSTAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Ivar Grimstad (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 Ivar Grimstad with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Ivar Grimstad 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.

Rev. January 2006

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50 outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written

communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
    a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
    b. You must cause any modified files to carry prominent notices stating that You changed the files; and
    c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
    d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

        You may add Your own copyright statement to Your modifications and may provide

additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Chapter 1. Introduction

Model-View-Controller, or *MVC* for short, is a common pattern in Web frameworks where it is used predominantly to build HTML applications. The *model* refers to the application's data, the *view* to the application's data presentation and the *controller* to the part of the system responsible for managing input, updating models and producing output.

Web UI frameworks can be categorized as *action-based* or *component-based*. In an action-based framework, HTTP requests are routed to controllers where they are turned into actions by application code; in a component-based framework, HTTP requests are grouped and typically handled by framework components with little or no interaction from application code. In other words, in a component-based framework, the majority of the controller logic is provided by the framework instead of the application.

The API defined by this specification falls into the action-based category and is, therefore, not intended to be a replacement for component-based frameworks such as JavaServer Faces (JSF) [1], but simply a different approach to building Web applications on the Java EE platform.

1.1. Goals

The following are goals of the API:

Goal 1

Leverage existing Java EE technologies like CDI [2] and Bean Validation [3].

Goal 2

Define a solid core to build MVC applications without necessarily supporting all the features in its first version.

Goal 3

Build on top of JAX-RS for the purpose of re-using its matching and binding layers.

Goal 4

Provide built-in support for JSPs and Facelets view languages.

1.2. Non-Goals

The following are non-goals of the API:

Non-Goal 1

Define a new view (template) language and processor.

Non-Goal 2

Support standalone implementations of MVC running outside of Java EE.

Non-Goal 3

Support REST services not based on JAX-RS.

Non-Goal 4

Provide built-in support for view languages that are not part of Java EE.

It is worth noting that, even though a standalone implementation of MVC that runs outside of Java EE is a non-goal, this specification shall not intentionally prevent implementations to run in other environments, provided that those environments include support for all the EE technologies required by MVC.

1.3. Additional Information

The issue tracking system for this specification can be found at:

https://github.com/mvc-spec/mvc-spec/issues

The corresponding Javadocs can be found online at:

https://www.mvc-spec.org/

The reference implementation can be obtained from:

https://www.mvc-spec.org/ozark/

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

jsr371-users@googlegroups.com

1.4. Terminology

Most of the terminology used in this specification is borrowed from other specifications such as JAX-RS and CDI. We use the terms *per-request* and *request-scoped* as well as *per-application* and *application-scoped* interchangeably.

1.5. 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 [4].

Assertions defined by this specification are formatted as **[[an-assertion]]** using a descriptive name as the label and are all listed in the Summary of Annotations section.

Java code and sample data fragments are formatted as shown below:

```
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.

1.6. Specification Leads

The following table lists the current and former specification leads:

Ivar Grimstad (Individual Member)	(Jan 2017 - present)
Christian Kaltepoth (ingenit GmbH & Co. KG)	(May 2017 - present)
Santiago Pericas-Geertsen (Oracle)	(Aug 2014 - Jan 2017)
Manfred Riem (Oracle)	(Aug 2014 - Jan 2017)

1.7. Expert Group Members

This specification is being developed as part of JSR 371 under the Java Community Process. The following are the present expert group members:

Mathieu Ancelin (Individual Member)	Ivar Grimstad (Individual Member)
Neil Griffin (Liferay, Inc)	Joshua Wilson (RedHat)
Rodrigo Turini (Caelum)	Stefan Tilkov (innoQ Deutschland GmbH)
Guilherme de Azevedo Silveira (Individual Member)	Frank Caputo (Individual Member)
Christian Kaltepoth (ingenit GmbH & Co. KG)	Woong-ki Lee (TmaxSoft, Inc.)
Paul Nicolucci (IBM)	Kito D. Mann (Individual Member)
Rahman Usta (Individual Member)	Florian Hirsch (adorsys GmbH & Co KG)
Santiago Pericas-Geertsen (Oracle)	Manfred Riem (Oracle)

1.8. Contributors

The following are the contributors of the specification:

Daniel Dias dos Santos	Phillip Krüger
Andreas Badelt	

1.9. Acknowledgements

During the course of this JSR we received many excellent suggestions. Special thanks to Marek Potociar, Dhiru Pandey and Ed Burns, all from Oracle. In addition, to everyone in the user's alias that followed the expert discussions and provided feedback, including Peter Pilgrim, Ivar Grimstad, Jozef Hartinger, Florian Hirsch, Frans Tamura, Rahman Usta, Romain Manni-Bucau, Alberto Souza, among many others.

Chapter 2. Models, Views and Controllers

This chapter introduces the three components that comprise the architectural pattern: models, views and controllers.

2.1. Controllers

An *MVC controller* is a JAX-RS [5] resource method decorated by `@Controller`. If this annotation is applied to a class, then all resource methods in it are regarded as controllers. Using the `@Controller` annotation on a subset of methods defines a hybrid class in which certain methods are controllers and others are traditional JAX-RS resource methods.

A simple hello-world controller can be defined as follows:

```
1 @Path("hello")
2 public class HelloController {
3
4     @GET
5     @Controller
6     public String hello(){
7         return "hello.jsp";
8     }
9 }
```

In this example, `hello` is a controller method that returns a path to a JavaServer Page (JSP). The semantics of controller methods differ slightly from JAX-RS resource methods; in particular, a return type of `String` is interpreted as a view path rather than text content. Moreover, the default media type for a response is assumed to be `text/html`, but otherwise can be declared using `@Produces` just like in JAX-RS.

A controller's method return type determines how its result is processed:

void

A controller method that returns void is REQUIRED to be decorated by `@View`.

String

A string returned is interpreted as a view path.

Response

A JAX-RS `Response` whose entity's type is one of the above.

The following class defines equivalent controller methods:

```java
@Controller
@Path("hello")
public class HelloController {

    @GET @Path("void")
    @View("hello.jsp")
    public void helloVoid() {
    }

    @GET @Path("string")
    public String helloString() {
        return "hello.jsp";
    }

    @GET @Path("response")
    public Response helloResponse() {
        return Response.status(Response.Status.OK)
            .entity("hello.jsp")
            .build();
    }
}
```

Controller methods that return a non-void type may also be decorated with `@View` as a way to specify a *default* view for the controller. The default view MUST be used only when such a non-void controller method returns a `null` value.

Note that, even though controller methods return types are restricted as explained above, MVC does not impose any restrictions on parameter types available to controller methods: i.e., all parameter types injectable in JAX-RS resources are also available in controllers. Likewise, injection of fields and properties is unrestricted and fully compatible with JAX-RS. Note the restrictions explained in Section Controller Instances.

Controller methods handle a HTTP request directly. Sub-resource locators as described in the JAX-RS Specification [5] are not supported by MVC.

2.1.1. Controller Instances

Unlike in JAX-RS where resource classes can be native (created and managed by JAX-RS), CDI beans, managed beans or EJBs, MVC classes are REQUIRED to be CDI-managed beans only. It follows that a hybrid class that contains a mix of JAX-RS resource methods and MVC controllers must also be CDI managed.

Like in JAX-RS, the default resource class instance lifecycle is *per-request*. Implementations MAY support other lifecycles via CDI; the same caveats that apply to JAX-RS classes in other lifecycles applied to MVC classes. In particular, CDI may need to create proxies when, for example, a per-request instance is as a member of a per-application instance. See [5] for more information on lifecycles and their caveats.

2.1.2. Response

Returning a `Response` object gives applications full access to all the parts in a response, including the headers. For example, an instance of `Response` can modify the HTTP status code upon encountering an error condition; JAX-RS provides a fluent API to build responses as shown next.

```
@GET
@Controller
public Response getById(@PathParam("id") String id) {
    if (id.length() == 0) {
        return Response.status(Response.Status.BAD_REQUEST)
            .entity("error.jsp")
            .build();
    }
    //...
}
```

Direct access to `Response` enables applications to override content types, set character encodings, set cache control policies, trigger an HTTP redirect, etc. For more information, the reader is referred to the Javadoc for the `Response` class.

2.1.3. Redirect and @RedirectScoped

As stated in the previous section, controllers can redirect clients by returning a `Response` instance using the JAX-RS API. For example,

```
@GET
@Controller
public Response redirect() {
    return Response.seeOther(URI.create("see/here")).build();
}
```

Given the popularity of the POST-redirect-GET pattern, MVC implementations are REQUIRED to support view paths prefixed by `redirect:` as a more concise way to trigger a client redirect. Using this prefix, the controller shown above can be re-written as follows:

```
@GET
@Controller
public String redirect() {
    return "redirect:see/here";
}
```

In either case, relative paths are resolved relative to the JAX-RS application path - for more information please refer to the Javadoc for the `seeOther` method. It is worth noting that redirects require client cooperation (all browsers support it, but certain CLI clients may not) and result in a completely new request-response cycle in order to access the intended controller. If a controller

returns a `redirect:` view path, MVC implementations SHOULD use the 303 (See other) status code for the redirect, but MAY prefer 302 (Found) if HTTP 1.0 compatibility is required.

MVC applications can leverage CDI by defining beans in scopes such as request and session. A bean in request scope is available only during the processing of a single request, while a bean in session scope is available throughout an entire web session which can potentially span tens or even hundreds of requests.

Sometimes it is necessary to share data between the request that returns a redirect instruction and the new request that is triggered as a result. That is, a scope that spans at most two requests and thus fits between a request and a session scope. For this purpose, the MVC API defines a new CDI scope identified by the annotation `@RedirectScoped`. CDI beans in this scope are automatically created and destroyed by correlating a redirect and the request that follows. The exact mechanism by which requests are correlated is implementation dependent, but popular techniques include URL rewrites and cookies.

Let us assume that `MyBean` is annotated by `@RedirectScoped` and given the name `mybean`, and consider the following controller:

```
1 @Controller
2 @Path("submit")
3 public class MyController {
4
5     @Inject
6     private MyBean myBean;
7
8     @POST
9     public String post() {
10        myBean.setValue("Redirect about to happen");
11        return "redirect:/submit";
12    }
13
14    @GET
15    public String get() {
16        return "mybean.jsp"; // mybean.value accessed in JSP
17    }
18 }
```

The bean `myBean` is injected in the controller and available not only during the first `POST`, but also during the subsequent `GET` request, enabling *communication* between the two interactions; the creation and destruction of the bean is under control of CDI, and thus completely transparent to the application just like any other built-in scope.

2.2. Models

MVC controllers are responsible for combining data models and views (templates) to produce web application pages. This specification supports two kinds of models: the first is based on CDI `@Named` beans, and the second on the `Models` interface which defines a map between names and

objects. MVC provides view engines for JSP and Facelets out of the box, which support both types. For all other view engines supporting the `Models` interface is mandatory, support for CDI `@Named` beans is OPTIONAL but highly RECOMMENDED.

Let us now revisit our hello-world example, this time also showing how to update a model. Since we intend to show the two ways in which models can be used, we define the model as a CDI `@Named` bean in request scope even though this is only necessary for the CDI case:

```
@Named("greeting")
@RequestScoped
public class Greeting {

    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
    //...
}
```

Given that the view engine for JSPs supports `@Named` beans, all the controller needs to do is fill out the model and return the view. Access to the model is straightforward using CDI injection:

```
@Path("hello")
public class HelloController {

    @Inject
    private Greeting greeting;

    @GET
    @Controller
    public String hello() {
        greeting.setMessage("Hello there!");
        return "hello.jsp";
    }
}
```

This will allow the view to access the greeting using the EL expression `${hello.greeting}`.

Instead of using CDI beans annotated with `@Named`, controllers can also use the `Models` map to pass data to the view:

```
@Path("hello")
public class HelloController {

    @Inject
    private Models models;

    @GET
    @Controller
    public String hello() {
        models.put("greeting", new Greeting("Hello there!"));
        return "hello.jsp";
    }
}
```

In this example, the model is given the same name as that in the `@Named` annotation above, but using the injectable `Models` map instead.

For more information about view engines see the View Engines section.

2.3. Views

A *view*, sometimes also referred to as a template, defines the structure of the output page and can refer to one or more models. It is the responsibility of a *view engine* to process (render) a view by extracting the information in the models and producing the output page.

Here is the JSP page for the hello-world example:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello</title>
    </head>
    <body>
        <h1>${greeting.message}</h1>
    </body>
</html>
```

In a JSP, model properties are accessible via EL [6]. In the example above, the property `message` is read from the `greeting` model whose name was either specified in a `@Named` annotation or used as a key in the `Models` map, depending on which controller from the Models section triggered this view's processing.

Here is the corresponding example using Facelets instead of JSP:

```
<!DOCTYPE html>
<html lang="en" xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Hello</title>
    </h:head>
    <h:body>
        <h:outputText value="#{greeting.message}" />
    </h:body>
</html>
```

2.3.1. Building URIs in a View

A typical application requires to build URIs for the view, which often refer to controller methods within the same application. Typical examples for such URIs include HTML links and form actions. As building URIs manually is difficult and duplicating path patterns between the controller class and the view is error prone, MVC provides a simple way to generate URIs using the `MvcContext` class.

See the following controller as an example:

```
@Controller
@Path("books")
public class BookController {

    @GET
    public String list() {
      // ...
    }

    @GET
    @Path("{id}")
    public String detail( @PathParam("id") long id ) {
      // ...
    }

}
```

Assuming the application is deployed with the context path `/myapp` and is using the application path `/mvc`, URIs for these controller methods can be created with an EL expression like this:

```
<!-- /myapp/mvc/books -->
${mvc.uri('BookController#list')}

<!-- /myapp/mvc/books/1234 -->
${mvc.uri('BookController#detail', { 'isbn': 1234 })}
```

The controller method is referenced using the simple name of the controller class and the corresponding method name separated by `#`. If the URI contains path, query or matrix parameters,

concrete values can be supplied using a map. Please note that the keys of this map must match the parameter name used in the `@PathParam`, `@QueryParam` or `@MatrixParam` annotation. MVC implementations MUST apply the corresponding URI encoding rules depending on whether the value is used in a query, path or matrix parameter.

The syntax used above to reference the controller method works well in most cases. However, because of the simple nature of this reference style, it will require controller class names to be unique. Also, the references may break if the controller class or method name changes as part of a refactoring.

Therefore, applications can use the `@UriRef` annotation to define a stable and unique name for a controller method.

```
@Controller
@Path("books")
public class BookController {

    @GET
    @UriRef("book-list")
    public String list() {
      // ...
    }

    // ...

}
```

Given such a controller class, the view can generate a matching URI by referencing the controller method using this reference.

```
<!-- /myapp/mvc/books -->
${mvc.uri('book-list')}
```

Please note that this feature will work with JSP, Facelets and all view engines which support invoking methods on CDI model objects.

Chapter 3. Data Binding

This chapter discusses data binding in the MVC API. Data binding is based on the underlying mechanism provided by JAX-RS, but with additional support for i18n requirements and for handling data binding errors within the controller.

3.1. Introduction

JAX-RS provides support for binding request parameters (like form/query parameters) to resource fields or resource method parameters. Starting with JAX-RS 2.0, developers can also specify validation constraints using Bean Validation annotations. In this case submitted values are automatically validated against the given constraints and rejected if validation fails.

Let's have a look at the following resource for an example:

```
1 @Path("form")
2 public class FormResource {
3
4     @FormParam("age")
5     @Min(18)
6     private int age;
7
8     @POST
9     public Response handlePost() {
10        // ...
11    }
12 }
```

This resource uses a `@FormParam` annotation to bind the value of the `age` form parameter to a resource field. It also uses the Bean Validation annotation `@Min` to specify a constraint on the value.

When JAX-RS binds the submitted data to the field, two types of errors are possible:

Binding Error

This type occurs if JAX-RS is unable to convert the submitted value into the desired target Java type. For the resource shown above, such an error will be thrown if the user submits some arbitrary string like `foobar` which cannot be converted into an integer.

Validation Error

If the submitted value can be converted into the target type, JAX-RS will validate the data according to the Bean Validation constraints. In our example submitting the value 16 would be considered invalid and therefore result in a constraint violation.

Unfortunately the JAX-RS data binding mechanism doesn't work well for web applications:

- Both binding and validation errors will cause JAX-RS to throw an exception which can only be handled by an `ExceptionMapper`. Especially JAX-RS won't execute the resource method if errors were detected. This is problematic, because typically web applications will want to display the

submitted form again and include a message explaining why the submission failed. Implementing such a requirement using an `ExceptionMapper` is not feasible.

- The JAX-RS data binding is not locale-aware. This is a problem especially for numeric data types containing fraction digits (like `double`, `float`, `BigDecimal`, etc). By default, JAX-RS will always assume the US number format.

3.2. @MvcBinding annotation

MVC addresses the shortcomings of the standard JAX-RS data binding by providing a special data binding mode optimized for web applications. You can enable the MVC specific data binding by adding a `@MvcBinding` annotation to the corresponding controller field or method parameter.

The following example shows a controller which uses a MVC binding on a controller field.

```
1 @Controller
2 @Path("form")
3 public class FormController {
4
5     @MvcBinding
6     @FormParam("age")
7     @Min(18)
8     private int age;
9
10    @POST
11    public String processForm() {
12        // ...
13    }
14 }
```

Please note that usually `@MvcBinding` will be used with `@FormParam` and `@QueryParam` bindings, as they are very common in web application. However, depending on the specific use case, it may also be useful to use it with other parameter binding types. Therefore, MVC implementations MUST support `@MvcBinding` with all JAX-RS binding annotations.

The following sections will describe the differences from traditional JAX-RS data binding in detail.

3.3. Error handling with BindingResult

As mentioned in the first section, JAX-RS data binding aborts request processing for any binding or validation error. This means, that a resource method will only be invoked if all bindings were successful.

MVC bindings handle such errors in a different way. An MVC implementation is required to invoke the matched controller method even if binding or validation errors occurred. Controllers can inject a request-scoped instance of `BindingResult` to access details about potential data binding errors. This allows controllers to handle such errors themselves, which typically means that human-readable error messages are presented to the user when the next view is rendered.

The following example shows a controller which uses `BindingResult` to handle data binding errors:

```
@Controller
@Path("form")
public class FormController {

    @MvcBinding
    @FormParam("age")
    @Min(18)
    private int age;

    @Inject
    private BindingResult bindingResult;

    @Inject
    private Models models;

    @POST
    public String processForm() {

        if( bindingResult.isFailed() ) {
            models.put( "errors", bindingResult.getAllMessages() );
            return "form.jsp";
        }

        // process the form request

    }
}
```

Please note that it is very important for a controller to actually check the `BindingResult` for errors if it uses MVC bindings. If a binding failed and the controller processes the value without checking for errors, the bound value may be empty or contain an invalid value.

MVC implementations SHOULD log a warning if a request created data binding errors but the controller didn't invoke any method on `BindingResult`.

3.4. Converting to Java types

The standard JAX-RS data binding doesn't work very well for web application, because it isn't locale-aware and some standard HTML form elements submit data which cannot easily be bound to matching Java types (e.g. checkboxes are submitting `on` if checked and JAX-RS is expecting `true` for boolean values).

MVC implementations are required to apply the following data conversion rules if a binding is annotated with `@MvcBinding`.

3.4.1. Numeric types

Implementations MUST support `int`, `long`, `float`, `double`, `BigDecimal`, `BigInteger` and corresponding wrapper types for MVC bindings. Support for other numeric types is optional. When converting values to these numeric Java types, MVC implementations MUST use the current *request locale* for parsing non-empty strings. Typically, an implementation will use a `NumberFormat` instance initialized with the corresponding locale for converting the data. Empty strings are either converted to `null` or to the default value of the corresponding primitive data type. Please refer to the Internationalization section for details about the MVC request locale.

3.4.2. Boolean type

When an MVC implementation converts a non-empty string to a `boolean` primitive type or the `java.lang.Boolean` wrapper type, it MUST convert both `true` and `on` to the boolean `true` and all others strings to `false`. Empty strings are converted to `false` in case of the primitive `boolean` type and to `null` for the wrapper type.

3.4.3. Other types

The conversion rules for all other Java types are implementation-specific.

Chapter 4. Security

4.1. Introduction

Guarding against malicious attacks is a great concern for web application developers. In particular, MVC applications that accept input from a browser are often targetted by attackers. Two of the most common forms of attacks are cross-site request forgery (CSRF) and cross-site scripting (XSS). This chapter explores techniques to prevent these type of attacks with the aid of the MVC API.

4.2. Cross-site Request Forgery

Cross-site Request Forgery (CSRF) is a type of attack in which a user, who has a trust relationship with a certain site, is mislead into executing some commands that exploit the existence of such a trust relationship. The canonical example for this attack is that of a user unintentionally carrying out a bank transfer while visiting another site.

The attack is based on the inclusion of a link or script in a page that accesses a site to which the user is known or assumed to have been authenticated (trusted). Trust relationships are often stored in the form of cookies that may be active while the user is visiting other sites. For example, such a malicious site could include the following HTML snippet:

```
<img src="http://yourbank.com/transfer?from=yours&to=mine&sum=1000000">
```

This will result in the browser executing a bank transfer in an attempt to load an image.

In practice, most sites require the use of form posts to submit requests such as bank transfers. The common way to prevent CSRF attacks is by embedding additional, difficult-to-guess data fields in requests that contain sensible commands. This additional data, known as a token, is obtained from the trusted site but unlike cookies it is never stored in the browser.

MVC implementations provide CSRF protection using the `Csrf` object and the `@CsrfProtected` annotation. The `Csrf` object is available to applications via the injectable `MvcContext` type or in EL as `mvc.csrf`. For more information about `MvcContext`, please refer to the MVC Context section.

Applications may use the `Csrf` object to inject a hidden field in a form that can be validated upon submission. Consider the following JSP:

```
<html>
    <head>
        <title>CSRF Protected Form</title>
    </head>
    <body>
        <form action="csrf" method="post" accept-charset="utf-8">
            <input type="submit" value="Click here"/>
            <input type="hidden" name="${mvc.csrf.name}"
                value="${mvc.csrf.token}"/>
        </form>
    </body>
</html>
```

The hidden field will be submitted with the form, giving the MVC implementation the opportunity to verify the token and ensure the validity of the post request.

Another way to convey this information to and from the client is via an HTTP header. MVC implementations are REQUIRED to support CSRF tokens both as form fields (with the help of the application developer as shown above) and as HTTP headers.

The application-level property `javax.mvc.security.CsrfProtection` enables CSRF protection when set to one of the possible values defined in `javax.mvc.security.Csrf.CsrfOptions`. The default value of this property is `CsrfOptions.EXPLICIT`. Any other value than `CsrfOptions.OFF` will automatically inject a CSRF token as an HTTP header. The actual name of this header is implementation dependent.

Automatic validation is enabled by setting this property to `CsrfOptions.IMPLICIT`, in which case all post requests must include either an HTTP header or a hidden field with the correct token. Finally, if the property is set to `CsrfOptions.EXPLICIT` then application developers must annotate controllers using `@CsrfProtected` to manually enable validation as shown in the following example.

```
@Path("csrf")
@Controller
public class CsrfController {

    @GET
    public String getForm() {
        return "csrf.jsp"; // Injects CSRF token
    }

    @POST
    @CsrfProtected // Required for CsrfOptions.EXPLICIT
    public void postForm(@FormParam("greeting") String greeting) {
        // Process greeting
    }
}
```

MVC implementations are required to support CSRF validation of tokens for controllers annotated

with `@POST` and consuming the media type `x-www-form-urlencoded`; other media types and scenarios may also be supported but are OPTIONAL.

If CSRF protection is enabled for a controller method and the CSRF validation fails (because the token is either missing or invalid), the MVC implementation MUST throw a `javax.mvc.security.CsrfValidationException`. The implementation MUST provide a default exception mapper for this exception which handles it by responding with a 403 (Forbidden) status code. Applications MAY provide a custom exception mapper for `CsrfValidationException` to change this default behavior.

4.3. Cross-site Scripting

Cross-site scripting (XSS) is a type of attack in which snippets of scripting code are injected and later executed when returned back from a server. The typical scenario is that of a website with a search field that does not validate its input, and returns an error message that includes the value that was submitted. If the value includes a snippet of the form `<script>...</script>` then it will be executed by the browser when the page containing the error is rendered.

There are lots of different variations of this the XSS attack, but most can be prevented by ensuring that the data submitted by clients is properly *sanitized* before it is manipulated, stored in a database, returned to the client, etc. Data escaping/encoding is the recommended way to deal with untrusted data and prevent XSS attacks.

MVC applications can gain access to encoders through the `MvcContext` object; the methods defined by `javax.mvc.security.Encoders` can be used by applications to contextually encode data in an attempt to prevent XSS attacks. The reader is referred to the Javadoc for this type for further information.

Chapter 5. Events

This chapter introduces a mechanism by which MVC applications can be informed of important events that occur while processing a request. This mechanism is based on CDI events that can be fired by implementations and observed by applications.

5.1. Observers

The package `javax.mvc.event` defines a number of event types that MUST be fired by implementations during the processing of a request. Implementations MAY extend this set and also provide additional information on any of the events defined by this specification. The reader is referred to the implementation's documentation for more information on event support.

Observing events can be useful for applications to learn about the lifecycle of a request, perform logging, monitor performance, etc. The events `BeforeControllerEvent` and `AfterControllerEvent` are fired around the invocation of a controller. Please note that `AfterControllerEvent` is always fired, even if the controller fails with an exception.

```java
/**
 * <p>Event fired before a controller is called but after it has been matched.</p>
 *
 * <p>For example:
 * <pre><code>    public class EventObserver {
 *         public void beforeControllerEvent(&#64;Observes BeforeControllerEvent e) {
 *             ...
 *        }
 *    }</code></pre>
 *
 * @author Santiago Pericas-Geertsen
 * @see javax.enterprise.event.Observes
 * @since 1.0
 */
public interface BeforeControllerEvent extends MvcEvent {

    /**
     * Access to the current request URI information.
     *
     * @return URI info.
     * @see javax.ws.rs.core.UriInfo
     */
    UriInfo getUriInfo();

    /**
     * Access to the current request controller information.
     *
     * @return resources info.
     * @see javax.ws.rs.container.ResourceInfo
     */
    ResourceInfo getResourceInfo();
}
```

```
/**
 * <p>Event fired after a controller method returns. This event is always fired,
 * even if the controller methods fails with an exception. Must be fired after
 * {@link javax.mvc.event.BeforeControllerEvent}.</p>
 *
 * <p>For example:
 * <pre><code>    public class EventObserver {
 *         public void afterControllerEvent(&#64;Observes AfterControllerEvent e) {
 *             ...
 *        }
 *    }</code></pre>
 *
 * @author Santiago Pericas-Geertsen
 * @author Christian Kaltepoth
 * @see javax.enterprise.event.Observes
 * @since 1.0
 */
public interface AfterControllerEvent extends MvcEvent {

    /**
     * Access to the current request URI information.
     *
     * @return URI info.
     * @see javax.ws.rs.core.UriInfo
     */
    UriInfo getUriInfo();

    /**
     * Access to the current request controller information.
     *
     * @return resources info.
     * @see javax.ws.rs.container.ResourceInfo
     */
    ResourceInfo getResourceInfo();
}
```

Applications can monitor these events using an observer as shown next.

```
@ApplicationScoped
public class EventObserver {

    public void onBeforeController(@Observes BeforeControllerEvent e) {
        System.out.println("URI: " + e.getUriInfo().getRequestURI());
    }

    public void onAfterController(@Observes AfterControllerEvent e) {
        System.out.println("Controller: " +
        e.getResourceInfo().getResourceMethod());
    }
}
```

Observer methods in CDI are defined using the `@Observes` annotation on a parameter position. The class `EventObserver` is a CDI bean in application scope whose methods `onBeforeController` and `onAfterController` are called before and after a controller is called.

Every event generated must include a unique ID whose getter is defined in `MvcEvent`, the base type for all events. Moreover, each event includes additional information that is specific to the event; for example, the events shown in the example above allow applications to get information about the request URI and the resource (controller) selected.

The View Engines section describes the algorithm used by implementations to select a specific view engine for processing; after a view engine is selected, the method `processView` is called. The events `BeforeProcessViewEvent` and `AfterProcessViewEvent` are fired around this call. Please note that `AfterProcessViewEvent` is always fired, even if the view engine fails with an exception.

```
/**
 * <p>Event fired after a view engine has been selected but before its
 * {@link javax.mvc.engine.ViewEngine#processView(javax.mvc.engine.ViewEngineContext)}
 * method is called. Must be fired after {@link javax.mvc.event.ControllerRedirectEvent},
 * or if that event is not fired, after {@link javax.mvc.event.AfterControllerEvent}.</p>
 *
 * <p>For example:
 * <pre><code>    public class EventObserver {
 *         public void beforeProcessView(&#64;Observes BeforeProcessViewEvent e) {
 *            ...
 *        }
 *    }</code></pre>
 *
 * @author Santiago Pericas-Geertsen
 * @see javax.enterprise.event.Observes
 * @since 1.0
 */
public interface BeforeProcessViewEvent extends MvcEvent {

    /**
     * Returns the view being processed.
     *
     * @return the view.
     */
    String getView();

    /**
    * Returns the {@link javax.mvc.engine.ViewEngine} selected by the implementation.
            *
            * @return the view engine selected.
            */
    Class<? extends ViewEngine> getEngine();
}
```

```
/**
 * <p>Event fired after the view engine method
 * {@link javax.mvc.engine.ViewEngine#processView(javax.mvc.engine.ViewEngineContext)}
 * returns. This event is always fired, even if the view engine fails with an exception.
 * Must be fired after {@link javax.mvc.event.BeforeProcessViewEvent}.</p>
 *
 * <p>For example:
 * <pre><code>    public class EventObserver {
 *         public void afterProcessView(&#64;Observes AfterProcessViewEvent e) {
 *             ...
 *         }
 *    }</code></pre>
 *
 * @author Santiago Pericas-Geertsen
 * @author Christian Kaltepoth
 * @see javax.enterprise.event.Observes
 * @since 1.0
 */
public interface AfterProcessViewEvent extends MvcEvent {

    /**
     * Returns the view being processed.
     *
     * @return the view.
     */
    String getView();

    /**
     * Returns the {@link javax.mvc.engine.ViewEngine} selected by the implementation.
     *
     * @return the view engine selected.
     */
    Class<? extends ViewEngine> getEngine();
}
```

These events can be observed in a similar manner:

```
@ApplicationScoped
public class EventObserver {

    public void onBeforeProcessView(@Observes BeforeProcessViewEvent e) {
        // ...
    }

    public void onAfterProcessView(@Observes AfterProcessViewEvent e) {
        // ...
    }
}
```

To complete the example, let us assume that the information about the selected view engine needs

to be conveyed to the client. To ensure that this information is available to a view returned to the client, the `EventObserver` class can inject and update the same request-scope bean accessed by such a view:

```
@ApplicationScoped
public class EventObserver {

    @Inject
    private EventBean eventBean;

    public void onBeforeProcessView(@Observes BeforeProcessViewEvent e) {
        eventBean.setView(e.getView());
        eventBean.setEngine(e.getEngine());
    }
    // ...
}
```

For more information about the interaction between views and models, the reader is referred to the Models section.

The last event supported by MVC is `ControllerRedirectEvent`, which is fired just before the MVC implementation returns a redirect status code. Please note that this event MUST be fired after `AfterControllerEvent`.

```java
import java.net.URI;

/**
 * <p>Event fired when a controller triggers a redirect. Only the
 * status codes 301 (moved permanently), 302 (found), 303 (see other) and
 * 307 (temporary redirect) are REQUIRED to be reported. Note that the
 * JAX-RS methods
 * {@link javax.ws.rs.core.Response#seeOther(java.net.URI)}} and
 * {@link javax.ws.rs.core.Response#temporaryRedirect(java.net.URI)}}
 * use the status codes to 303 and 307, respectively. Must be
 * fired after {@link javax.mvc.event.AfterControllerEvent}.</p>
 *
 * <p>For example:
 * <pre><code>    public class EventObserver {
 *         public void onControllerRedirect(&#64;Observes ControllerRedirectEvent e) {
 *             ...
 *         }
 *    }</code></pre>
 *
 * @author Santiago Pericas-Geertsen
 * @see javax.enterprise.event.Observes
 * @since 1.0
 */
public interface ControllerRedirectEvent extends MvcEvent {

    /**
     * Access to the current request URI information.
     *
     * @return URI info.
     * @see javax.ws.rs.core.UriInfo
     */
    UriInfo getUriInfo();

    /**
     * Access to the current request controller information.
     *
     * @return resources info.
     * @see javax.ws.rs.container.ResourceInfo
     */
    ResourceInfo getResourceInfo();

    /**
     * The target of the redirection.
     *
     * @return URI of redirection.
     */
    URI getLocation();
}
```

CDI events fired by implementations are *synchronous*, so it is recommended that applications carry out only simple tasks in their observer methods, avoiding long-running computations as well as

blocking calls. For a complete list of events, the reader is referred to the Javadoc for the `javax.mvc.event` package.

Event reporting requires the MVC implementations to create event objects before firing. In high-throughput systems without any observers the number of unnecessary objects created may not be insignificant. For this reason, it is RECOMMENDED for implementations to consider smart firing strategies when no observers are present.

Chapter 6. Applications

This chapter introduces the notion of an MVC application and explains how it relates to a JAX-RS application.

6.1. MVC Applications

An MVC application consists of one or more JAX-RS resources that are annotated with `@Controller` and, just like JAX-RS applications, zero or more providers. If no resources are annotated with `@Controller`, then the resulting application is a JAX-RS application instead. In general, everything that applies to a JAX-RS application also applies to an MVC application. Some MVC applications may be *hybrid* and include a mix of MVC controllers and JAX-RS resource methods.

The controllers and providers that make up an application are configured via an application-supplied subclass of `Application` from JAX-RS. An implementation MAY provide alternate mechanisms for locating controllers, but as in JAX-RS, the use of an `Application` subclass is the only way to guarantee portability.

The path in the application's URL space in which MVC controllers live must be specified either using the `@ApplicationPath` annotation on the application subclass or in the web.xml as part of the `url-pattern` element. MVC applications SHOULD use a non-empty path or pattern: i.e., "/" or "/*" should be avoided whenever possible. The reason for this is that MVC implementations often forward requests to the Servlet container, and the use of the aforementioned values may result in the unwanted processing of the forwarded request by the JAX-RS servlet once again.

6.2. MVC Context

MVC applications can inject an instance of `MvcContext` to access configuration, security and path-related information. Instances of `MvcContext` are provided by implementations and are always in request scope. For convenience, the `MvcContext` instance is also available using the name `mvc` in EL.

As an example, a view can refer to a controller by using the base path available in the `MvcContext` object as follows:

```
<a href="${mvc.basePath}/books">Click here</a>
```

For more information on security see the Chapter on Security; for more information about the `MvcContext` in general, refer to the Javadoc for the type.

6.3. Providers in MVC

Implementations are free to use their own providers in order to modify the standard JAX-RS pipeline for the purpose of implementing the MVC semantics. Whenever mixing implementation and application providers, care should be taken to ensure the correct execution order using priorities.

6.4. Annotation Inheritance

MVC applications MUST follow the annotation inheritance rules defined by JAX-RS. Namely, MVC annotations may be used on methods of a super-class or an implemented interface. Such annotations are inherited by a corresponding sub-class or implementation class method provided that the method does not have any MVC or JAX-RS annotations of its own: i.e., if a subclass or implementation method has any MVC or JAX-RS annotations then all of the annotations on the superclass or interface method are ignored.

Annotations on a super-class take precedence over those on an implemented interface. The precedence over conflicting annotations defined in multiple implemented interfaces is implementation dependent. Note that, in accordance to the JAX-RS rules, inheritance of class or interface annotations is not supported.

6.5. Configuration in MVC

Implementations MUST support configuration via the native JAX-RS configuration mechanism but MAY support other configuration sources.

There are concrete configurations, that all MVC the implementations are `REQUIRED` the support such as:

- `ViewEngine.VIEW_FOLDER`
- `Csrf.CSRF_PROTECTION`

Here's a simple example of how you can configure a custom location for the view folder other than the `/WEB-INF/views`, simply by overwriting the `getProperties` method of the subclass `Application`:

```
1 @ApplicationPath("resources")
2 public class MyApplication extends Application {
3
4     @Override
5     public Map<String, Object> getProperties() {
6         final Map<String, Object> map = new HashMap<>();
7         map.put(ViewEngine.VIEW_FOLDER, "/jsp/");
8         return map;
9     }
10 }
```

Chapter 7. View Engines

This chapter introduces the notion of a view engine as the mechanism by which views are processed in MVC. The set of available view engines is extensible via CDI, enabling applications as well as other frameworks to provide support for additional view languages.

7.1. Introduction

A *view engine* is responsible for processing views. In this context, processing entails (i) locating and loading a view (ii) preparing any required models and (iii) rendering the view and writing the result back to the client.

Implementations MUST provide built-in support for JSPs and Facelets view engines. Additional engines may be supported via an extension mechanism based on CDI. Namely, any CDI bean that implements the `javax.mvc.engine.ViewEngine` interface MUST be considered as a possible target for processing by calling its `supports` method, discarding the engine if this method returns `false`.

This is the interface that must be implemented by all MVC view engines:

```
/**
 * <p>View engines are responsible for processing views and are discovered
 * using CDI. Implementations must look up all instances of this interface,
 * and process a view as follows:
 * <ol>
 *     <li>Gather the set of candidate view engines by calling {@link #supports(String)}
 *     and discarding engines that return <code>false</code>.</li>
 *     <li>Sort the resulting set of candidates using priorities. View engines
 *     can be decorated with {@link javax.annotation.Priority} to indicate
 *     their priority; otherwise the priority is assumed to be {@link ViewEngine#PRIORITY_APPLICATION}.</li>
 *     <li>If more than one candidate is available, choose one in an
 *     implementation-defined manner.</li>
 *     <li>Fire a {@link javax.mvc.event.BeforeProcessViewEvent} event.</li>
 *     <li>Call method {@link #processView(ViewEngineContext)} to process view.</li>
 *     <li>Fire a {@link javax.mvc.event.AfterProcessViewEvent} event.</li>
 * </ol>
 * <p>The default view engines for JSPs and Facelets use file extensions to determine
 * support. Namely, the default JSP view engine supports views with extensions <code>jsp</code>
 * and <code>jspx</code>, and the one for Facelets supports views with extension
 * <code>xhtml</code>.</p>
 *
 * @author Santiago Pericas-Geertsen
 * @see javax.annotation.Priority
 * @see javax.mvc.event.BeforeProcessViewEvent
 * @since 1.0
 */
@SuppressWarnings("unused")
public interface ViewEngine {

    /**
     * Name of property that can be set to override the root location for views in an archive.
     *
     * @see javax.ws.rs.core.Application#getProperties()
     */
    String VIEW_FOLDER = "javax.mvc.engine.ViewEngine.viewFolder";

```

```
37      /**
38       * Default value for property {@link #VIEW_FOLDER}.
39       */
40      String DEFAULT_VIEW_FOLDER = "/WEB-INF/views/";
41
42      /**
43       * Priority for all built-in view engines.
44       */
45      int PRIORITY_BUILTIN = 1000;
46
47      /**
48       * Recommended priority for all view engines provided by frameworks built
49       * on top of MVC implementations.
50       */
51      int PRIORITY_FRAMEWORK = 2000;
52
53      /**
54       * Recommended priority for all application-provided view engines (default).
55       */
56      int PRIORITY_APPLICATION = 3000;
57
58      /**
59       * Returns <code>true</code> if this engine can process the view or <code>false</code>
60       * otherwise.
61       *
62       * @param view the view.
63       * @return outcome of supports test.
64       */
65      boolean supports(String view);
66
67      /**
68       * <p>Process a view given a {@link javax.mvc.engine.ViewEngineContext}. Processing
69       * a view involves <i>merging</i> the model and template data and writing
70       * the result to an output stream.</p>
71       *
72       * <p>Following the Java EE threading model, the underlying view engine implementation
73       * must support this method being called by different threads. Any resources allocated
74       * during view processing must be released before the method returns.</p>
75       *
76       * @param context the context needed for processing.
77       * @throws ViewEngineException if an error occurs during processing.
78       */
79      void processView(ViewEngineContext context) throws ViewEngineException;
80  }
```

7.2. Selection Algorithm

Implementations should perform the following steps while trying to find a suitable view engine for a view.

1. Lookup all instances of `javax.mvc.engine.ViewEngine` available via CDI.
2. Call `supports` on every view engine found in the previous step, discarding those that return `false`.
3. If the resulting set is empty, return `null`.
4. Otherwise, sort the resulting set in descending order of priority using the integer value from the `@Priority` annotation decorating the view engine class or the default value

`ViewEngine.PRIORITY_APPLICATION` if the annotation is not present.

5. Return the first element in the resulting sorted set, that is, the view engine with the highest priority that supports the given view.

If a view engine that can process a view is not found, implementations are REQUIRED to throw a `ViewEngineException`.

The `processView` method has all the information necessary for processing in the `ViewEngineContext`, including the view, a reference to `Models`, as well as the underlying `OutputStream` that can be used to send the result to the client. Implementations MUST catch exceptions thrown during the execution of `processView` and re-throw them as `ViewEngineException`'s.

Prior to the view render phase, all entries available in `Models` MUST be bound in such a way that they become available to the view being processed. The exact mechanism for this depends on the actual view engine implementation. In the case of the built-in view engines for JSPs and Facelets, entries in `Models` must be bound by calling `HttpServletRequest.setAttribute(String, Object)`. Calling this method ensures access to the named models from EL expressions.

A view returned by a controller method represents a path within an application archive. If the path is relative, does not start with `/`, implementations MUST resolve view paths relative to the view folder, which defaults to `/WEB-INF/views/`. If the path is absolute, no further processing is required. It is recommended to use relative paths and a location under `WEB-INF` to prevent direct access to views as static resources.

7.3. FacesServlet

Because Facelets support is not enabled by default, MVC applications that use Facelets are required to package a `web.xml` deployment descriptor with the following entry mapping the extension `*.xhtml` as shown next:

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
</servlet-mapping>
```

Alternatively to a `web.xml` deployment descriptor an empty `faces-config.xml` file can be placed in the `WEB-INF` folder to enable Facelets support.

It is worth noting that if you opt to use Facelets as a view technology for your MVC application, regular JSF post-backs will not be processed by the MVC runtime. The usage of `<h:form />` and depending form components like `<h:inputText />` is not recommended as they would be the entry point to a real JSF application.

Chapter 8. Internationalization

This chapter introduces the notion of a *request locale* and describes how MVC handles internationalization and localization.

8.1. Introduction

Internationalization and localization are very important concepts for any web application framework. Therefore MVC has been designed to make supporting multiple languages and regional differences in applications very easy.

MVC defines the term *request locale* as the locale which is used for any locale-dependent operation within the lifecycle of a request. The request locale MUST be resolved exactly once for each request using the resolving algorithm described in the Resolving Algorithm section.

These locale-dependent operations include, but are not limited to:

1. Data type conversion as part of the data binding mechanism.
2. Formatting of data when rendering it to the view.
3. Generating binding and validation error messages in the specific language.

The request locale is available from `MvcContext` and can be used by controllers, view engines and other components to perform operations which depend on the current locale. The example below shows a controller that uses the request locale to create a `NumberFormat` instance.

```
1 @Controller
2 @Path("/foobar")
3 public class MyController {
4
5     @Inject
6     private MvcContext mvc;
7
8     @GET
9     public String get() {
10        Locale locale = mvc.getLocale();
11        NumberFormat format = NumberFormat.getInstance(locale);
12    }
13 }
```

The following sections will explain the locale resolving algorithm and the default resolver provided by the MVC implementation.

8.2. Resolving Algorithm

The *locale resolver* is responsible to detect the request locale for each request processed by the MVC runtime. A locale resolver MUST implement the `javax.mvc.locale.LocaleResolver` interface which is defined like this:

```java
/**
 * <p>Locale resolvers are used to determine the locale of the current request and are discovered
 * using CDI.</p>
 *
 * <p>The MVC implementation is required to resolve the locale for each request following this
 * algorithm:</p>
 *
 * <ol>
 * <li>Gather the set of all implementations of this interface available for injection via
 * CDI.</li>
 * <li>Sort the set of implementations using priorities in descending order. Locale resolvers
 * can be decorated with {@link javax.annotation.Priority} to indicate their priority. If no
 * priority is explicitly defined, the priority is assumed to be <code>1000</code>.</li>
 * <li>Call the method {@link #resolveLocale(LocaleResolverContext)}. If the resolver returns
 * a valid locale, use this locale as the request locale. If the resolver returns
 * <code>null</code>, proceed with the next resolver in the ordered set.</li>
 * </ol>
 *
 * <p>Controllers, view engines and other components can access the resolved locale by calling
 * {@link MvcContext#getLocale()}.</p>
 *
 * <p>The MVC implementation is required to provide a default locale resolver with a priority
 * of <code>0</code> which uses the <code>Accept-Language</code> request header to obtain the
 * locale. If resolving the locale this way isn't possible, the default resolver must return
 * {@link Locale#getDefault()}.</p>
 *
 * @author Christian Kaltepoth
 * @see javax.mvc.locale.LocaleResolverContext
 * @see MvcContext#getLocale()
 * @see java.util.Locale
 * @since 1.0
 */
public interface LocaleResolver {

    /**
     * <p>Resolve the locale of the current request given a {@link LocaleResolverContext}.</p>
     *
     * <p>If the implementation is able to resolve the locale for the request, the corresponding
     * locale must be returned. If the implementation cannot resolve the locale, it must return
     * <code>null</code>. In this case the resolving process will continue with the next
     * resolver.</p>
     *
     * @param context the context needed for processing.
     * @return The resolved locale or <code>null</code>.
     */
    Locale resolveLocale(LocaleResolverContext context);

}
```

There may be more than one locale resolver for a MVC application. Locale resolvers are discovered using CDI. Every CDI bean implementing the `LocaleResolver` interface and visible to the application participates in the locale resolving algorithm.

Implementations MUST use the following algorithm to resolve the request locale for each request:

1. Obtain a list of all CDI beans implementing the `LocaleResolver` interface visible to the application's `BeanManager`.
2. Sort the list of locale resolvers in descending order of priority using the integer value from the `@Priority` annotation decorating the resolver class.
   If no `@Priority` annotation is present, assume a default priority of `1000`.
3. Call `resolveLocale()` on the first resolver in the list. If the resolver returns `null`, continue with the next resolver in the list.
   If a resolver returns a non-null result, stop the algorithm and use the returned locale as the request locale.

Applications can either rely on the default locale resolver which is described in the Default Locale Resolver section or provide a custom resolver which implements some other strategy for resolving the request locale. A custom strategy could for example track the locale using the session, a query parameter or the server's hostname.

8.3. Default Locale Resolver

Every MVC implementation MUST provide a default locale resolver with a priority of `0` which resolves the request locale according to the following algorithm:

1. First check whether the client provided an `Accept-Language` request header. If this is the case, the locale with the highest quality factor is returned as the result.
2. If the previous step was not successful, return the system default locale of the server.

Please note that applications can customize the locale resolving process by providing a custom locale resolver with a priority higher than `0`. See the Resolving Algorithm section for details.

Appendix A: Summary of Annotations

Annotation	Target	Description
`Controller`	Type or method	Defines a resource method as an MVC controller. If specified at the type level, it defines all methods in a class as controllers.
`View`	Type or method	Declares a view for a controller method that returns void. If specified at the type level, it applies to all controller methods that return void in a class.
`CsrfValid`	Method	States that a CSRF token must be validated before invoking the controller. Failure to validate the CSRF token results in a `ForbiddenException` thrown.
`RedirectScoped`	Type, method or field	Specifies that a certain bean is in redirect scope.
`UriRef`	Method	Defines a symbolic name for a controller method.
`MvcBinding`	Field, method or parameter	Declares that constraint violations will be handled by a controller through `BindingResult` instead of triggering a `ConstraintViolationException`.

Bibliography

[1]

Edward Burns. JavaServer Faces 2.2. JSR, JCP, May 2013
http://jcp.org/en/jsr/detail?id=344

[2]

Pete Muir. Context and Dependency Injection for Java EE 1.1 MR. JSR, JCP, April 2014
http://jcp.org/en/jsr/detail?id=346

[3]

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

[4]

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

[5]

Santiago Pericas-Geertsen and Marek Potociar. The Java API for RESTful Web Services 2.0 MR. JSR, JCP, October 2014
http://jcp.org/en/jsr/detail?id=339

[6]

Kin man Chung. Expression Language 3.0. JSR, JCP, May 2013
http://jcp.org/en/jsr/detail?id=341