MVC: Model-View-Controller API

Version 1.0 Early Draft (Second Edition)
July 12, 2016

Editors:
Santiago Pericas-Geertsen
Manfred Riem
Christian Kaltepoth

Comments to: users@mvc-spec.java.net

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

ORACLE AMERICA, INC. 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 Model View Controller API (”Specification”) Version: 1.0
Status: Early Draft Review 2
Release: October 2015

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

NOTICE

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, and Java 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 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

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.

Goal 2 Integrate with CDI [2] and Bean Validation [3].

Goal 3 Define a solid core to build MVC applications without necessarily supporting all the features in its first version.

Goal 4 Explore layering on top of JAX-RS for the purpose of re-using its matching and binding layers.

Goal 5 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://java.net/jira/browse/MVC_SPEC

The corresponding Javadocs can be found online at:

https://mvc-spec.java.net/

The reference implementation can be obtained from:

https://ozark.java.net/

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

users@mvc-spec.java.net

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 Appendix C.

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.*

1.6 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)

1.7 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 MVC architectural pattern: models, views and controllers.

2.1 Controllers

An *MVC controller* is a JAX-RS [5] resource method decorated by `@Controller` ⟦**mvc:controller**⟧. If this annotation is applied to a class, then all resource methods in it are regarded as controllers ⟦**mvc:all-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` ⟦**mvc:void-controllers**⟧.

String A string returned is interpreted as a view path.

Viewable A `Viewable` is a class that encapsulates a view path as well as additional information related to its processing.

(Java Type) The method `toString` is called on other Java types and the result 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:

```
1  @Controller
2  @Path("hello")
3  public class HelloController {
4
5      @GET @Path("void")
6      @View("hello.jsp")
7      public void helloVoid() {
8      }
9
10     @GET @Path("string")
11     public String helloString() {
12         return "hello.jsp";
13     }
14
15     @GET @Path("viewable")
16     public Viewable helloViewable() {
17         return new Viewable("hello.jsp");
18     }
19
20     @GET @Path("response")
21     public Response helloResponse() {
22         return Response.status(Response.Status.OK)
23                        .entity("hello.jsp").build();
24     }
25
26     @GET @Path("myview")
27     public MyView helloMyView() {
28         return new MyView("hello.jsp");    // toString() -> "hello.jsp"
29     }
30 }
```

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 ⟦**mvc:null-controllers**⟧.

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 MVC controllers. Likewise, injection of fields and properties is unrestricted and fully compatible with JAX-RS —modulo the restrictions explained in Section 2.1.1.

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 ⟦**mvc:cdi-beans**⟧. 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* ⟦**mvc:per-request**⟧. That is, an instance of a controller class MUST be instantiated and initialized on every request. Implementations MAY support other lifecycles via CDI; the same caveats that apply to JAX-RS classes in other lifecycles applied to MVC classes. [1] See [5] for more information on lifecycles and their caveats.

2.1.2 Viewable

The `Viewable` class encapsulates information about a view as well as, optionally, information about how it should be processed. More precisely, a `Viewable` instance may include references to `Models` and `ViewEngine` objects —for more information see Section 2.2 and Chapter 7, respectively. `Viewable` defines traditional constructors for all these objects and it is, therefore, not a CDI-managed bean.

The reader is referred to the Javadoc of the `Viewable` class for more information on its semantics.

2.1.3 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.

```
1  @GET
2  @Controller
3  public Response getById(@PathParam("id") String id) {
4      if (id.length() == 0) {
5          return Response.status(Response.Status.BAD_REQUEST)
6                         .entity("error.jsp").build();
7      }
8      ...
9  }
```

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.

2.1.4 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,

```
1  @GET
2  @Controller
3  public Response redirect() {
4      return Response.seeOther(URI.create("see/here")).build();
5  }
```

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 ⟦**mvc:redirect**⟧. Using this prefix, the controller shown above can be re-written as follows:

[1] In particular, CDI may need to create proxies when, for example, a per-request instance is as a member of a per-application instance.

```
1  @GET
2  @Controller
3  public String redirect() {
4      return "redirect:see/here";
5  }
```

In either case, the HTTP status code returned is 302 and relative paths are resolved relative to the application path –for more information please refer to the Javadoc for the `seeOther` method in JAX-RS. 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.

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. Support for the `Models` interface is mandatory for all view engines; support for CDI `@Named` beans is OPTIONAL but highly RECOMMENDED. Application developers are encouraged to use CDI-based models whenever supported, and thus take advantage of the existing CDI and EL integration on the platform.

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:

```
1  @Named("greeting")
2  @RequestScoped
3  public class Greeting {
4
5      private String message;
6
7      public String getMessage() { return message; }
8      public void setMessage(String message) { this.message = message; }
9      ...
10 }
```

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:

```
1  @Path("hello")
2  public class HelloController {
3
4      @Inject
5      private Greeting greeting;
6
7      @GET
8      @Controller
9      public String hello() {
10         greeting.setMessage("Hello there!");
11         return "hello.jsp";
12     }
13 }
```

If the view engine that processes the view returned by the controller is not CDI enabled, then controllers can use the `Models` map instead:

```
1  @Path("hello")
2  public class HelloController {
3
4      @Inject
5      private Models models;
6
7      @GET
8      @Controller
9      public String hello() {
10         models.put("greeting", new Greeting("Hello there!");
11         return "hello.jsp";
12     }
13 }
```

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

As stated above, the use of typed CDI `@Named` beans is recommended over the `Models` map, but support for the latter may be necessary to integrate view engines that are not CDI aware. For more information about view engines see Chapter 7.

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:

```
1  <html>
2  <head>
3      <title>Hello</title>
4  </head>
5  <body>
6  <h1>${greeting.message}</h1>
7  </body>
8  </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 Section 2.2 triggered this view's processing.

Chapter 3

Exception Handling

This chapter discusses exception handling in the MVC API. Exception handling in MVC is based on the underlying mechanism provided by JAX-RS, but with additional support for handling binding and validation exceptions that are common in MVC frameworks.

3.1 Exception Mappers

The general exception handling mechanism in MVC controllers is identical to that defined for resource methods in the JAX-RS specification. In a nutshell, applications can implement exception mapping providers for the purpose of converting exceptions to responses. If an exception mapper is not found for a particular exception type, default rules apply that describe how to process the exception depending on whether it is checked or unchecked, and using additional rules for the special case of a `WebApplicationException` that includes a response. The reader is referred to the JAX-RS specification for more information.

Let us consider the case of a `ConstraintViolationException` that is thrown as a result of a bean validation failure:

```
1  @Controller
2  @Path("form")
3  public class FormController {
4
5      @POST
6      public Response formPost(@Valid @BeanParam FormDataBean form) {
7          return Response.status(OK).entity("data.jsp").build();
8      }
9  }
```

The method `formPost` defines a bean parameter of type `FormDataBean` which, for the sake of the example, we assume includes validation constraints such as `@Min(18)`, `@Size(min=1)`, etc. The presence of `@Valid` triggers validation of the bean on every HTML form post; if validation fails, a `ConstraintViolationException` (a subclass of `ValidationException`) is thrown.

An application can handle the exception by including an exception mapper as follows:

```
1  public class FormViolationMapper
2                implements ExceptionMapper<ConstraintViolationException> {
3
```

```
4       @Inject
5       private ErrorDataBean error;
6
7       @Override
8       public Response toResponse(ConstraintViolationException e) {
9           final Set<ConstraintViolation<?>> set = e.getConstraintViolations();
10          if (!set.isEmpty()) {
11              // fill out ErrorDataBean ...
12          }
13          return Response.status(Response.Status.BAD_REQUEST)
14                         .entity("error.jsp").build();
15      }
16  }
```

This exception mapper updates an instance of `ErrorDataBean` and returns the `error.jsp` view (wrapped in a response as required by the method signature) with the intent to provide a human-friendly description of the exception.

Even though using exception mappers is a convenient way to handle exceptions in general, there are cases in which finer control is necessary. The mapper defined above will be invoked for all instances of `ConstraintViolationException` thrown in an application. Given that applications may include several form-post controllers, handling all exceptions in a single location makes it difficult to provide controller-specific customizations. Moreover, exception mappers do not get access to the (partially valid) bound data, or `FormDataBean` in the example above.

3.2 Validation Exceptions

MVC provides an alternative exception handling mechanism that is specific for the use case described in Section 3.1. Rather than funnelling exception handling into a single location while providing no access to the bound data, controller methods may opt to act as exception handlers as well. In other words, controller methods can get called even if parameter validation fails as long as they declare themselves capable of handling errors.

A controller class that, either directly or indirectly via inheritance, defines a field or a property of type `javax.mvc.binding.BindingResult` will have its controller methods called even if an error is encountered while validating parameters. Implementations MUST introspect the controller bean for a field or a property of this type to determine the correct semantics; fields and property setters of this type MUST be annotated with `@Inject` to guarantee proper bean initialization ⟦**mvc:validation-result**⟧.

Let us revisit the example from Section 3.1, this time using the controller method as an exception handler:

```
1   @Controller
2   @Path("form")
3   public class FormController {
4
5       @Inject
6       private BindingResult br;
7
8       @Inject
9       private ErrorDataBean error;
10
11      @POST
```

```
12      @ValidateOnExecution(type = ExecutableType.NONE)
13      public Response formPost(@Valid @BeanParam FormDataBean form) {
14          if (br.isFailed()) {
15              // fill out ErrorDataBean ...
16              return Response.status(BAD_REQUEST).entity("error.jsp").build();
17          }
18          return Response.status(OK).entity("data.jsp").build();
19      }
20  }
```

The presence of the injection target for the field `br` indicates to an implementation that controller methods in this class can handle errors. As a result, methods in this class that validate parameters should call `br.isFailed()` to verify if validation errors were found. [1]

The class `BindingResult` provides methods to get detailed information about any violations found during validation. Instances of this class are always in request scope; the reader is referred to the Javadoc for more information.

As previously stated, properties of type `BindingResult` are also supported. Here is a modified version of the example in which a property is used instead:

```
1   @Controller
2   @Path("form")
3   public class FormController {
4
5       private BindingResult br;
6
7       public BindingResult getBr() {
8           return br;
9       }
10
11      @Inject
12      public void setBr(BindingResult br) {
13          this.br = br;
14      }
15      ...
16  }
```

Note that the `@Inject` annotation has been moved from the field to the setter, thus ensuring the bean is properly initialized by CDI when it is created. Implementations MUST give precedence to a property (calling its getter and setter) over a field if both are present in the same class.

3.3 Binding Exceptions

As suggested by its name, instances of `BindingResult` also track any binding errors that occur while mapping request parameters to Java types. Binding errors are discovered even before validation takes place. An example of a binding error is that of a query parameter bound to an `int` whose value cannot be converted to that type.

[1] The `ValidateOnExecution` annotation is necessary to ensure that CDI and BV do not abort the invocation upon detecting a violation. Thus, to ensure the correct semantics, validation must be performed by the JAX-RS implementation before the method is called.

JAX-RS uses the notion of a parameter converter to provide extension points for these conversions; if none are specified for the type at hand, a set of default parameter converters is available. Regardless of where the parameter converter is coming from, a failure to carry out a conversion results in an `IllegalArgumentException` thrown and, typically, a 500 error code returned to the client. As explained before, applications can provide an exception mapper for `IllegalArgumentException` but this may be insufficient when error recovery using controller-specific logic is required.

Controllers can call the same `isFailed` method to check for binding errors —the method returns true if at least one error of either kind is found. Additional methods in the `BindingResult` type allow to get specific information related to binding errors. See the Javadoc for more information.

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">
```

which 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 `@CsrfValid` 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 Section 6.2.

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

```
1  <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.OFF`. Setting it to any other value 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 `@CsrfValid` to manually enable validation ⟦**mvc:csrf-options**⟧ as shown in the following example.

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

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

    @POST
    @CsrfValid                 // 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` ⟦**mvc:csrf-support**⟧; other media types and scenarios may also be supported but are OPTIONAL.

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 ⟦**mvc:event-firing**⟧. 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; applications can monitor these events using an observer as shown next.

```
1  @ApplicationScoped
2  public class EventObserver {
3
4      public void onBeforeController(@Observes BeforeControllerEvent e) {
5          System.out.println("URI: " + e.getUriInfo().getRequestURI());
6      }
7
8      public void onAfterController(@Observes AfterControllerEvent e) {
9          System.out.println("Controller: " +
10             e.getResourceInfo().getResourceMethod());
11     }
12 }
```

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.

Chapter 7 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 and can be observed in a similar manner:

```
1  @ApplicationScoped
2  public class EventObserver {
3
4      public void onBeforeProcessView(@Observes BeforeProcessViewEvent e) {
5          ...
6      }
7
8      public void onAfterProcessView(@Observes AfterProcessViewEvent e) {
9          ...
10     }
11 }
```

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:

```
1  @ApplicationScoped
2  public class EventObserver {
3
4      @Inject
5      private EventBean eventBean;
6
7      public void onBeforeProcessView(@Observes BeforeProcessViewEvent e) {
8          eventBean.setView(e.getView());
9          eventBean.setEngine(e.getEngine());
10     }
11
12     ...
13 }
```

For more information about the interaction between views and models, the reader is referred to Section 2.2.

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.

All the rules described in the Servlet section of the JAX-RS Specification [5] apply to MVC as well. This section recommends the use of the Servlet 3 framework pluggability mechanism and describes its semantics for the cases in which an `Application` subclass is present and absent.

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. Most JAX-RS applications avoid using these values, and many use `"/resources"` or `"/resources/*"` by convention. For consistency, it is recommended for MVC applications to use these patterns as well.

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 application

scope ⟦**mvc:mvc-context**⟧. For convenience, the `MvcContext` instance is also available using the name `mvc` in EL.

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

```
1   <link rel="stylesheet" type="text/css" href="${mvc.contextPath}/my.css">
```

For more information on security see Chapter 4; 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 ⟦**mvc:annotation-inheritance**⟧. 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.

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 ⟦**mvc:builtin-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` ⟦**mvc:extension-engines**⟧.

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

```
1  public interface ViewEngine {
2
3      boolean supports(String view);
4
5      void processView(ViewEngineContext context) throws ViewEngineException;
6  }
```

7.2 Selection Algorithm

As explained in Section 2.1.2, a `Viewable` is an encapsulation for information that relates to a view. Every possible return type from a controller method is either a `Viewable` or can be turned into one by calling a constructor. Thus, the following algorithm assumes only `Viewable` as input.

Implementations should perform the following steps while trying to find a suitable view engine for a `Viewable` ⟦**mvc:selection-algorithm**⟧.

1. If calling `getViewEngine` on the `Viewable` returns a non-null value, return that view engine.

2. Otherwise, lookup all instances of `javax.mvc.engine.ViewEngine` available via CDI. [1]
3. Call `supports` on every view engine found in the previous step, discarding those that return `false`.
4. If the resulting set is empty, return `null`.
5. 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 `Priorities.DEFAULT` if the annotation is not present.
6. Return the first element in the resulting sorted set, that is, the view engine with the highest priority that supports the given `Viewable`.

If a view engine that can process a `Viewable` is not found, as a fall-back attempt to process the view by other means, implementations are REQUIRED to forward the request-response pair back to the Servlet container using a `RequestDispatcher` ⟦**mvc:request-forward**⟧.

The `processView` method has all the information necessary for processing in the `ViewEngineContext`, including the view, a reference to `Models`, as well as the HTTP request and response from the underlying the Servlet container. Implementations MUST catch exceptions thrown during the execution of `processView` and re-throw them as `ViewEngineException`'s ⟦**mvc:exception-wrap**⟧.

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 value of `ViewEngine.DEFAULT_VIEW_FOLDER`, which is set to `/WEB-INF/views/`. If the path is absolute, no further processing is required ⟦**mvc:view-resolution**⟧. 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:

```
1     <servlet>
2         <servlet-name>Faces Servlet</servlet-name>
3         <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
4         <load-on-startup>1</load-on-startup>
5     </servlet>
6     <servlet-mapping>
7         <servlet-name>Faces Servlet</servlet-name>
8         <url-pattern>*.xhtml</url-pattern>
9     </servlet-mapping>
```

[1] The `@Any` annotation in CDI can be used for this purpose.

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.

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 Section 8.2.

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 ⟦**mvc:request-locale-context**⟧. 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
11         Locale locale = mvc.getLocale();
```

```
12            NumberFormat format = NumberFormat.getInstance(locale);
13
14        }
15
16    }
```

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:

```
1   public interface LocaleResolver {
2
3       Locale resolveLocale(LocaleResolverContext context);
4
5   }
```

There may be more than one locale resolver for a MVC application. Locale resolvers are discovered using CDI ⟦**mvc:extension-resolvers**⟧. 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 ⟦**mvc:resolve-algorithm**⟧:

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 Section 8.3 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 ⟦**mvc:default-locale-resolver**⟧:

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. Check if there is an *application default locale* specified in the JAX-RS configuration. The application default locale can be configured using the key `LocaleResolver.DEFAULT_LOCALE` and a `java.util.Locale` as the value. If such a default locale is found, return it as the result.
3. If the previous steps were 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 Section 8.2 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.

Appendix B

Change Log

B.1 Changes Since 1.0 Early Draft

- Section 3.3 New section related to the use of `BindingResult` to handle binding errors.
- Section 3.2 The type `ValidationResult` renamed to `BindingResult` after extending its scope to binding errors as well.
- Section 2.1.4 Introduce the redirect scope and related annotation.
- Section 6.2: New section about injectable `MvcContext`.
- Chapter 4: New chapter about security.
- Section 7.3: New section about `FacesServlet` and its configuration.
- Chapter 5: Updated based on changes to `javax.mvc.event` package.
- Section 6.4: New section on annotation inheritance rules.
- Section 2.1.4: New section about HTTP redirects.
- Section 2.1: Allow `@View` to be used for controller methods returning a `null` value.
- Section 2.1: Controller methods can return arbitrary Java types on which `toString` is called, interpreting the result as a view path.
- Section 2.1: Updated return type sample using unique paths.
- Section 8: New chapter about internationalization.

Appendix C

Summary of Assertions

⟦**mvc:controller**⟧ Controller methods are JAX-RS resource methods annotated with `@Controller`.

⟦**mvc:all-controllers**⟧ All resource methods in a class annotated with `@Controller` must be controllers.

⟦**mvc:void-controllers**⟧ Controller methods that return void must be annotated with `@View`.

⟦**mvc:cdi-beans**⟧ MVC beans are managed by CDI.

⟦**mvc:per-request**⟧ Default scope for MVC beans is request scope.

⟦**mvc:validation-result**⟧ If validation fails, controller methods must still be called if a `ValidationResult` field or property is defined.

⟦**mvc:event-firing**⟧ All events in `javax.mvc.event` must be fired. See Javadoc for more information on each event in that package.

⟦**mvc:builtin-engines**⟧ Implementations must provide support for JSPs and Facelets.

⟦**mvc:extension-engines**⟧ CDI beans that implement `javax.mvc.engine.ViewEngine` provide an extension mechanism for view engines.

⟦**mvc:selection-algorithm**⟧ Implementations must use algorithm in Section 7.2 to select view engines.

⟦**mvc:request-forward**⟧ Forward request for which no view engine is found.

⟦**mvc:exception-wrap**⟧ Exceptions thrown during view processing must be wrapped.

⟦**mvc:view-resolution**⟧ Relative paths to views must be resolved as explained in Section 7.2.

⟦**mvc:null-controllers**⟧ The `@View` annotation is treated as a default value for any controller method that returns a `null` value.

⟦**mvc:redirect**⟧ Support HTTP redirects using the `redirect:` prefix and a controller return type of `String`.

⟦**mvc:annotation-inheritance**⟧ Annotation inheritance is derived from JAX-RS and extended to MVC annotations.

⟦**mvc:csrf-options**⟧ CSRF support for configuration options defined by `Csrf.CsrfOptions`.

⟦**mvc:csrf-support**⟧ CSRF validation required only for controllers annotated by `@POST` and consuming the media type `x-www-form-urlencoded`.

⟦**mvc:mvc-context**⟧ Application-scoped `MvcContext` available for injection and as `mvc` in EL.

⟦**mvc:request-locale-context**⟧ The `MvcContext` must provide access to the current request locale.

⟦**mvc:extension-resolvers**⟧ CDI beans implementing `javax.mvc.locale.LocaleResolver` provide an extension mechanism for the request locale resolving algorithm.

⟦**mvc:resolve-algorithm**⟧ The request locale must be resolved as described in Section 8.2.

⟦**mvc:default-locale-resolver**⟧ Implementations must provide a default locale resolver as described in Section 8.3.

Bibliography

[1] Edward Burns. JavaServer Faces 2.2. JSR, JCP, May 2013. See 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. See http://jcp.org/en/jsr/detail?id=346.

[3] Emmanuel Bernard. Bean Validation 1.1. JSR, JCP, March 2013. See 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. See 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. See http://jcp.org/en/jsr/detail?id=339.

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