MVC: Model-View-Controller API

Version 1.0 Early Draft
March 4, 2015

Editors:
Santiago Pericas-Geertsen
Manfred Riem

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

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

JSR-371 MVC (“Specification”)
Version: 1.0
Status: Early Draft
Release: March 4, 2015

TBD

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-based 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 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 framworks like Java Server Faces (JSF) [**?**], 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 [**?**] and Bean Validation [**?**].

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 for standalone implementations of MVC running outside of Java EE.

Non-Goal 3 Support for 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 can run outside of Java EE is a non-goal, this specification shall not intentionally prevent implementations to run in other environments, provided that they include support all the EE technologies required by MVC.

1.3 Additional Information

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

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

The corresponding Javadocs can be found online at:

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

The reference implementation can be obtained from:

http://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 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[**?**].

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

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

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

Note: *This is a note.*

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  }
```

1.5 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 (Individual Member)
- Woong-ki Lee (TmaxSoft, Inc.)
- Paul Nicolucci (IBM)
- Kito D. Mann (Individual Member)

1.6 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.

Chapter 2

Models, Views and Controllers

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

2.1 Controllers

An *MVC controller* is a JAX-RS resource method decorated by an `@Controller` annotation. If this annotation is applied to a class, then all 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 Java Server 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.

The return type of a controller method is restricted to be one of four possible types:

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

String The string returned is interpreted as a path to a view.

Viewable A `Viewable` is a class that encapsulates information about views and their processing.

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

The following class defines equivalent controller methods:

```
1  @Controller
2  @Path("hello")
3  public class HelloController {
4
5      @GET
6      @View("hello.jsp")
7      public void helloVoid() {
8      }
9
10     @GET
11     public String helloString() {
12         return "hello.jsp";
13     }
14
15     @GET
16     public Viewable helloViewable() {
17         return new Viewable("hello.jsp");
18     }
19
20     @GET
21     public Response helloResponse() {
22         return Response.status(Response.Status.OK)
23                        .entity("hello.jsp").build();
24     }
25 }
```

Note that, even though controller methods return types are restricted as explained above, MVC does not impose any restrictions on the parameter types available for 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).

2.1.1 Controller Instances

Unlike in JAX-RS where resource classes can be native (i.e., 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. 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. In particular, proxies may be necessary when, for example, a per-request instance is as a member of a per-application instance. See [**?**] 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 Sections 2.2 and 6. `Viewable` defines traditional constructors for all these objects and it is, therefore, not a CDI-managed class.

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.

2.2 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 the `Models` interface which defines a map between names and objects, and the second is based on CDI `@Named` beans. Support for the `Models` interface is mandatory for all view engines; support for CDI `@Named` beans is OPTIONAL but highly RECOMMENDED. For more information on view engines see Chapter 6.

Let us now revisit our hello-world example, this time also showing how to update a model. Since we intent 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.set("greeting", new Greeting("Hello there!");
11         return "hello.jsp";
12     }
13 }
```

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.

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  <%@ page contentType="text/html;charset=UTF-8" language="java" %>
2  <html>
3  <head>
4      <title>Hello</title>
5  </head>
6  <body>
7  <h1>${greeting.message}</h1>
8  </body>
9  </html>
```

In a JSP, model properties are accessible via EL [**?**]. In the example above, the property `message` is read from the `greeting` model whose name was specified either in the `@Named` annotation or as the first argument to the setter in `Models`, depending on which controller from Section 2.2 triggered this view's processing.

Chapter 3

Applications

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

3.1 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. In what follows, we drop the adjective and refer to them simply as applications.

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 [**?**] 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 or not.

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 MUST use a non-empty path or pattern: i.e., *"/"* or *"/*"* are invalid in MVC applications.

The reason for this is that MVC implementations often forward requests to the Servlet container, and the use of these 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, which this specification also recommends.

3.2 Providers in MVC

Implementations are free 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.

Note: *Should we define the priorities for portability?*

Chapter 4

Exception Handling

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

4.1 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 mapping 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  @Produces("text/html")
4  public class FormController {
5
6      @POST
7      public Response formPost(@Valid @BeanParam FormDataBean form) {
8          return Response.status(OK).entity("data.jsp").build();
9      }
10 }
```

The method `formPost` injects 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  class FormViolationMapper implements
2                            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              final ConstraintViolation<?> cv = set.iterator().next();
12              // fill out ErrorDataBean ...
13
14          }
15          return Response.status(Response.Status.BAD_REQUEST)
16                         .entity("error.jsp").build();
17      }
18  }
```

This exception mapper updates `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 using a single method makes it difficult to provide controller-specific customizations. Moreover, exception mappers do not get access to the (likely partially valid) bound data, or `FormDataBean` in the example above.

4.2 Validation Exceptions

MVC provides an alternative exception handling mechanism that is specific for the use case described in Section 4.1. 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, either directly or indirectly via inheritance, that defines a field or a property of type `javax.mvc.validation.ValidationResult` will have its controller methods called even if a validation 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 MUST be annotated with `@Inject` to guarantee proper bean initialization.

Let us revisit the example from Section 4.1, this time using the controller method as an exception handler.

```
1   @Controller
2   @Path("form")
3   @Produces("text/html")
4   public class FormController {
5
6       @Inject
7       private ValidationResult vr;
8
9       @Inject
```

```
10      private ErrorDataBean error;
11
12      @POST
13      @ValidateOnExecution(type = ExecutableType.NONE)
14      public Response formPost(@Valid @BeanParam FormDataBean form) {
15          if (vr.isFailed()) {
16              final Set<ConstraintViolation<?>> set = vr.getAllViolations();
17              // fill out ErrorDataBean ...
18              return Response.status(BAD_REQUEST).entity("error.jsp").build();
19          }
20          return Response.status(OK).entity("data.jsp").build();
21      }
22  }
```

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

The class `javax.mvc.validation.ValidationResult` 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 `javax.mvc.validation.ValidationResult` are also supported. Here is a modified version of the example in which a property is used instead:

```
1   @Controller
2   @Path("form")
3   @Produces("text/html")
4   public class FormController {
5
6       private ValidationResult vr;
7
8       public ValidationResult getVr() {
9           return vr;
10      }
11
12      @Inject
13      public void setVr(ValidationResult vr) {
14          this.vr = vr;
15      }
16      ...
17  }
```

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.

Editors Note 4.1 *Support for injection of `javax.mvc.validation.ValidationResult` as a method parameter may be re-considered based on any new capabilities available in CDI 2.0.*

[1] The `ValidateOnExecution` annotation is necessary to ensure that CDI calls the method even after a validation failure is encountered. Thus, to ensure the correct semantics, validation must be done by the JAX-RS implementation before the method is called.

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, provide application-level logging, monitor performance, etc. Chapter 6 describes the algorithm used by implementations to select a specific view engine for processing. This information is made available to any application (or framework) that observes the `ViewEngineSelected` event. For example,

```
1  @ApplicationScoped
2  public class EventObserver {
3
4      public void onViewEngineSelected(@Observes ViewEngineSelected event) {
5          ...
6      }
7  }
```

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 method `onViewEngineSelected` is called every time a `ViewEngineSelected` event is fired by the implementation.

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 the view:

```
1  @ApplicationScoped
2  public class EventObserver {
3
```

```
4       @Inject
5       private EventBean eventBean;
6
7       public void onViewEngineSelected(@Observes ViewEngineSelected event) {
8           eventBean.setView(event.getView());
9           eventBean.setEngine(event.getEngine());
10      }
11  }
```

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

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.

Editors Note 5.1 *Synchronous vs. asynchronous event processing should be reviewed based on any new capabilities available in CDI 2.0.*

Chapter 6

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.

6.1 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 and writing the result into a response object.

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 `support` method, discarding the engine if this method returns `false`.

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  }
```

6.2 Selection Algorithm

As explained in Section 2.1.2, a `Viewable` is an encapsulation for information that relates to a view. In particular, a `Viewable` may optionally include a reference to a view engine. Implementations should perform the following steps while trying to find a suitable view engine for a `Viewable`.

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]

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

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`.

The `processView` method has all the information that is required for processing, 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.

Prior to the view render phase, all entries available in `Models` MUST be bound in such a way that they become available to the processing view. 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.

Appendix A

Summary of Annotations

Annotation	Target	Description

Appendix B

Change Log

B.1 Changes Since 1.0 Early Draft

- Section ??. TBD.

Bibliography