JAKARTA EE

Jakarta MVC Specification

Jakarta EE MVC Team, https://projects.eclipse.org/projects/ee4j.mvc

3.0, April 15, 2025: Final Release

Table of Contents

License

```
Specification: Jakarta MVC Specification

Version: 3.0

Status: Final Release

Release: April 15, 2025
```

Copyright

Eclipse Foundation Specification License - v1.1

Disclaimers

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 Jakarta Server Faces [1], but simply a different approach to building Web applications on the Jakarta EE platform.

1.1. Goals

The following are goals of the API:

Goal 1 Leverage existing Jakarta EE technologies like Jakarta Contexs and Dependency Injecttion [2] and Jakarta 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 Jakarta RESTful Web Services for the purpose of re-using its matching and binding layers.

Goal 4 Provide built-in support for Jakarta Server Pages 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 Jakarta EE.

Non-Goal 3 Support REST services not based on Jakarta RESTful Web Services.

Non-Goal 4 Provide built-in support for view languages that are not part of Jakarta EE.

It is worth noting that, even though a standalone implementation of MVC that runs outside of Jakarta 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 Jakarta EE technologies required by MVC.

1.3. Additional Information

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

https://github.com/eclipse-ee4j/mvc-api/issues

The corresponding Javadocs can be found online at:

https://jakarta.ee/specifications/mvc/1.1/apidocs/

A compatible implementation can be obtained from:

https://projects.eclipse.org/projects/ee4j.krazo

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

https://accounts.eclipse.org/mailing-list/mvc-dev

1.4. Terminology

Most of the terminology used in this specification is borrowed from other specifications such as Jakarta RESTful Web Services and Jakarta Contexts and Dependency Injection. 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].

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

```
package com.example.hello;

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

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. Acknowledgements for version 2.0

The Jakarta MVC 2.0 specification was created by the Jakarta MVC Specification Project with guidance provided by the Jakarta EE Working Group (https://jakarta.ee/).

1.7. Acknowledgements for version 1.1

The Jakarta MVC 1.1 specification was created by the Jakarta MVC Specification Project with guidance provided by the Jakarta EE Working Group (https://jakarta.ee/).

1.8. Acknowledgements for version 1.0

Version 1.0 was developed as part of JSR 371 under the Java Community Process.

1.8.1. Specification Leads

The following table lists the specification leads of the JSR:

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)

1.8.2. Expert Group Members

The following were the expert group members:

Ivar Grimstad (Individual Member)	Neil Griffin (Liferay, Inc)
Joshua Wilson (RedHat)	Rodrigo Turini (Caelum)
Stefan Tilkov (innoQ Deutschland GmbH)	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)

The following were former members of the expert group:

Guilherme de Azevedo Silveira (Individual Member)	

1.8.3. Contributors

The following were the contributors of the specification:

Daniel Dias dos Santos	Phillip Krüger
Andreas Badelt	

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

A *Jakarta MVC controller* is a Jakarta RESTful Web Services [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 Jakarta RESTful Web Services resource methods.

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

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

    @GET
    @Controller
    public String hello(){
        return "hello.jsp";
    }
}
```

In this example, `hello` is a controller method that returns a path to a Jakarta Server Page. The semantics of controller methods differ slightly from Jakarta RESTful Web Services 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 Jakarta RESTful Web Services.

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 Jakarta RESTful Web Services `Response` whose entity's type is one of the above.

The following class defines equivalent controller methods:

```
@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, Jakarta MVC does not impose any restrictions on parameter types available to controller methods: i.e., all parameter types injectable in Jakarta RESTful Web Services resources are also available in controllers. Likewise, injection of fields and properties is unrestricted and fully compatible with Jakarta RESTful Web Services. Note the restrictions explained in Section Controller Instances.

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

2.1.1. Controller Instances

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

Like in Jakarta RESTful Web Services, the default resource class instance lifecycle is *per-request*. Implementations MAY support other lifecycles via CDI; the same caveats that apply to Jakarta RESTful Web Services classes in other lifecycles applied to Jakarta 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; Jakarta RESTful Web Services 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 Jakarta RESTful Web Services 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, Jakarta 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 Jakarta RESTful Web Services 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, Jakarta 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.

Jakarta 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 Jakarta 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:

```
@Controller
@Path("submit")
public class MyController {

    @Inject
    private MyBean myBean;

    @POST
    public String post() {
        myBean.setValue("Redirect about to happen");
        return "redirect:/submit";
    }

    @GET
    public String get() {
        return "mybean.jsp"; // mybean.value accessed in Jakarta Server Page
    }
}
```

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

Jakarta 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. Jakarta MVC provides a view engine for Jakarta Server Pages 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 Jakarta Server Pages 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 Jakarta Server Pages page for the hello-world example:

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

In Jakarta Server Pages, 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.

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, Jakarta 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. Jakarta 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 Jakarta Server Pages and all view engines which support invoking methods on CDI model objects.

Chapter 3. Data Binding

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

3.1. Introduction

Jakarta RESTful Web Services provides support for binding request parameters (like form/query parameters) to resource fields or resource method parameters. With Jakarta RESTful Web Services, 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:

```
@Path("form")
public class FormResource {

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

    @POST
    public Response handlePost() {
        // ...
    }
}
```

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 Jakarta RESTful Web Services binds the submitted data to the field, two types of errors are possible:

Binding Error — This type occurs if Jakarta RESTful Web Services 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, Jakarta RESTful Web Services 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 Jakarta RESTful Web Services data binding mechanism doesn't work well for web applications:

- Both binding and validation errors will cause Jakarta RESTful Web Services to throw an exception which can only be handled by an `ExceptionMapper`. Especially Jakarta RESTful Web Services 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 Jakarta RESTful Web Services 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, Jakarta RESTful Web Services will always assume the US number format.

3.2. @MvcBinding annotation

Jakarta MVC addresses the shortcomings of the standard Jakarta RESTful Web Services data binding by providing a special data binding mode optimized for web applications. You can enable the Jakarta 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 Jakarta MVC binding on a controller field.

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

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

    @POST
    public String processForm() {
        // ...
    }
}
```

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, Jakarta MVC implementations MUST support `@MvcBinding` with all Jakarta RESTful Web Services binding annotations.

The following sections will describe the differences from traditional Jakarta RESTful Web Services data binding in detail.

3.3. Error handling with BindingResult

As mentioned in the first section, Jakarta RESTful Web Services 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.

Jakarta MVC bindings handle such errors in a different way. A Jakarta 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 Jakarta 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.

Jakarta 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 Jakarta RESTful Web Services 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 Jakarta RESTful Web Services is expecting `true` for boolean values).

Jakarta 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 Jakarta MVC bindings. Support for other numeric types is optional. When converting values to these numeric Java types, Jakarta 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 Jakarta MVC request locale.

3.4.2. Boolean type

When a Jakarta 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, Jakarta 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 Jakarta 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.

Jakarta 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 Jakarta Expression Language 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 Jakarta 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. Jakarta 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 `jakarta.mvc.security.CsrfProtection` enables CSRF protection when set to one of the possible values defined in `jakarta.mvc.security.Csrf.CsrfOptions`. The default value of this property is `CsrfOptions.IMPLICIT`. Any other value than `CsrfOptions.OFF` will automatically inject a CSRF token as an HTTP header. The actual name of the header can be configured via the `Csrf.CSRF_HEADER_NAME` configuration property. The default name of the header is `Csrf.DEFAULT_CSRF_HEADER_NAME`.

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
    }
}
```

Jakarta 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 Jakarta MVC implementation MUST throw a `jakarta.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.

Jakarta MVC applications can gain access to encoders through the `MvcContext` object; the methods defined by `jakarta.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 Jakarta MVC applications can be informed of important events that occur while processing a request. This mechanism is based on Jakarta Contexts and Dependency Injection events that can be fired by implementations and observed by applications.

5.1. Observers

The package `jakarta.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.

```
/**
 * <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
 * @author Ivar Grimstad
 * @see jakarta.enterprise.event.Observes
 * @since 1.0
 */
public interface BeforeControllerEvent extends MvcEvent {

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

    /**
     * Access to the current request controller information.
     *
     * @return resources info.
```

```
     * @see jakarta.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 jakarta.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
 * @author Ivar Grimstad
 * @see jakarta.enterprise.event.Observes
 * @since 1.0
 */
public interface AfterControllerEvent extends MvcEvent {

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

    /**
     * Access to the current request controller information.
     *
     * @return resources info.
     * @see jakarta.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 Jakarta Contexts and Dependency Injection are defined using the `@Observes` annotation on a parameter position. The class `EventObserver` is a Jakarta Contexts and Dependency Injection bean in application scope whose methods `onBeforeController` and `onAfterController` are called before and after a controller is called.

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 jakarta.mvc.engine.ViewEngine#processView(jakarta.mvc.engine.ViewEngineContext)}
 * method is called. Must be fired after {@link jakarta.mvc.event.ControllerRedirectEvent},
 * or if that event is not fired, after {@link jakarta.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
 * @author Ivar Grimstad
 * @see jakarta.enterprise.event.Observes
 * @since 1.0
 */
public interface BeforeProcessViewEvent extends MvcEvent {

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

    /**
```

```
     * Returns the {@link jakarta.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 jakarta.mvc.engine.ViewEngine#processView(jakarta.mvc.engine.ViewEngineContext)}
 * returns. This event is always fired, even if the view engine fails with an exception.
 * Must be fired after {@link jakarta.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
 * @author Ivar Grimstad
 * @see jakarta.enterprise.event.Observes
 * @since 1.0
 */
public interface AfterProcessViewEvent extends MvcEvent {

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

    /**
     * Returns the {@link jakarta.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 Jakarta MVC is `ControllerRedirectEvent`, which is fired just before the Jakarta MVC implementation returns a redirect status code. Please note that this event MUST be fired after `AfterControllerEvent`.

```
/**
 * <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 jakarta.ws.rs.core.Response#seeOther(java.net.URI)}} and
 * {@link jakarta.ws.rs.core.Response#temporaryRedirect(java.net.URI)}}
 * use the status codes to 303 and 307, respectively. Must be
 * fired after {@link jakarta.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
 * @author Ivar Grimstad
 * @see jakarta.enterprise.event.Observes
 * @since 1.0
 */
public interface ControllerRedirectEvent extends MvcEvent {

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

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

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

Jakarta Contexts and Dependency Injection 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 `jakarta.mvc.event` package.

Event reporting requires the Jakarta 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 a Jakarta MVC application and explains how it relates to a Jakarta RESTful Web Services application.

6.1. MVC Applications

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

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

The path in the application's URL space in which Jakarta 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. Jakarta MVC applications SHOULD use a non-empty path or pattern: i.e., "/" or "/*" should be avoided whenever possible. The reason for this is that Jakarta 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 Jakarta RESTful Web Services 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 Jakarta RESTful Web Services 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

Jakarta MVC applications MUST follow the annotation inheritance rules defined by Jakarta RESTful Web Services. Namely, Jakarta 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 Jakarta MVC or Jakarta RESTful Web Services annotations of its own: i.e., if a subclass or implementation method has any Jakarta MVC or Jakarta RESTful Web Services 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 Jakarta RESTful Web Services rules, inheritance of class or interface annotations is not supported.

6.5. Configuration in MVC

Implementations MUST support configuration via the native Jakarta RESTful Web Services configuration mechanism but MAY support other configuration sources.

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

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

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`:

```
@ApplicationPath("resources")
public class MyApplication extends Application {

    @Override
    public Map<String, Object> getProperties() {
        final Map<String, Object> map = new HashMap<>();
        map.put(ViewEngine.VIEW_FOLDER, "/jsp/");
        return map;
    }
}
```

Chapter 7. View Engines

This chapter introduces the notion of a view engine as the mechanism by which views are processed in Jakarta MVC. The set of available view engines is extensible via Jakarta Contexts and Dependency Injection, 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 Jakarta Server Pages. Additional engines may be supported via an extension mechanism based on Jakarta Contexts and Dependency Injection. Namely, any Jakarta Contexts and Dependency Injection bean that implements the `jakarta.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 Jakarta MVC view engines:

```
/**
 * <p>View engines are responsible for processing views and are discovered
 * using Jakarta Contexts and Dependency Injection. 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 jakarta.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 jakarta.mvc.event.BeforeProcessViewEvent} event.</li>
 *     <li>Call method {@link #processView(ViewEngineContext)} to process view.</li>
 *     <li>Fire a {@link jakarta.mvc.event.AfterProcessViewEvent} event.</li>
 * </ol>
 * <p>The default view engine for Jakarta Server Pages uses file extensions to determine
 * support. Namely, the default Jakarta Server Pages view engine supports views with extensions <code>jsp</code>
 * and <code>jspx</code>.</p>
 *
 * @author Santiago Pericas-Geertsen
 * @author Ivar Grimstad
 * @see jakarta.annotation.Priority
 * @see jakarta.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 jakarta.ws.rs.core.Application#getProperties()
     */
    String VIEW_FOLDER = "jakarta.mvc.engine.ViewEngine.viewFolder";
```

```
    /**
     * The default view file extension that is used to fetch templates. For example, if this setting is set to
<code>.jsp</code>,
     * instead of returning <code>index.jsp</code> only <code>index</code> is required.
     *
     * @see jakarta.ws.rs.core.Application#getProperties()
     */
    String VIEW_EXTENSION = "jakarta.mvc.engine.ViewEngine.defaultViewFileExtension";

    /**
     * Default value for property {@link #VIEW_FOLDER}.
     */
    String DEFAULT_VIEW_FOLDER = "/WEB-INF/views/";

    /**
     * Priority for all built-in view engines.
     */
    int PRIORITY_BUILTIN = 1000;

    /**
     * Recommended priority for all view engines provided by frameworks built
     * on top of MVC implementations.
     */
    int PRIORITY_FRAMEWORK = 2000;

    /**
     * Recommended priority for all application-provided view engines (default).
     */
    int PRIORITY_APPLICATION = 3000;

    /**
     * Returns <code>true</code> if this engine can process the view or <code>false</code>
     * otherwise.
     *
     * @param view the view.
     * @return outcome of supports test.
     */
    boolean supports(String view);

    /**
     * <p>Process a view given a {@link jakarta.mvc.engine.ViewEngineContext}. Processing
     * a view involves <i>merging</i> the model and template data and writing
     * the result to an output stream.</p>
     *
     * <p>Following the Jakarta EE threading model, the underlying view engine implementation
     * must support this method being called by different threads. Any resources allocated
     * during view processing must be released before the method returns.</p>
     *
     * @param context the context needed for processing.
     * @throws ViewEngineException if an error occurs during processing.
     */
    void processView(ViewEngineContext context) throws ViewEngineException;
}
```

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 `jakarta.mvc.engine.ViewEngine` available via Jakarta Contexts and Dependency Injection.

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 SHOULD throw a corresponding exception and stop to process the request.

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.

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, 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.

Chapter 8. Internationalization

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

8.1. Introduction

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

Jakarta 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.

```
@Controller
@Path("/foobar")
public class MyController {

    @Inject
    private MvcContext mvc;

    @GET
    public String get() {
        Locale locale = mvc.getLocale();
        NumberFormat format = NumberFormat.getInstance(locale);
    }
}
```

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

8.2. Resolving Algorithm

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

```
/**
 * <p>Locale resolvers are used to determine the locale of the current request and are discovered
 * using Jakarta Contexts and Dependency Injection.</p>
 *
 * <p>The Jakarta 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 jakarta.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
 * @author Ivar Grimstad
 * @see jakarta.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 Jakarta MVC application. Locale resolvers are discovered using Jakarta Contexts and Dependency Injection. Every Jakarta Contexts and Dependency Injection 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 Jakarta Contexts and Dependency Injection 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 Jakarta 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.

Chapter 9. Form method override

This chapter introduces the notion of *form method overwrite* and describes how Jakarta MVC supports HTTP methods besides `GET` and `POST` when using HTML forms.

9.1. Introduction

The HTML `<form>` is per default only capable of handling the HTTP `GET` and `POST` verbs. Anyway, more complex applications maybe want to use Jakarta MVC to support HTML as one type of resource representation and need the power of other HTTP verbs like `PATCH` or `DELETE` too. Therefore Jakarta MVC supports overwriting the HTTP method by providing an easy to use and configurable mechanism.

Jakarta MVC defines the term *form method overwrite* as the mechanism being responsible for changing the HTTP request's method to something different than `GET` or `POST`.

The *form method overwrite* MUST happen exactly once per request as described in Resolving Algorithm before the controller is resolved.

To have control over the form method handling, Jakarta MVC provides two properties with constants for easier usage in the class `jakarta.mvc.form.FormMethodOverwriter`:

```
public final class FormMethodOverwriter {

    /**
     * Property that can be used to enable the Form method overwrite mechanism for an application.
     * Values of this property must be of type {@link FormMethodOverwriter.Options}.
     */
    public static final String FORM_METHOD_OVERWRITE = "jakarta.mvc.form.FormMethodOverwrite";

    /**
     * Property that can be used to configure the name of the hidden form input to get the targeted HTTP method.
     */
    public static final String HIDDEN_FIELD_NAME = "jakarta.mvc.form.HiddenFieldName";

    /**
     * The default name of the hidden form field used to overwrite the HTTP method.
     */
    public static final String DEFAULT_HIDDEN_FIELD_NAME = "_method";

    /**
     * Options for property {@link FormMethodOverwriter#FORM_METHOD_OVERWRITE}.
     */
    public enum Options {
        /**
         * Form method overwrite is not enabled.
         */
        DISABLED,

        /**
         * Form method overwrite is enabled. Each request will be checked for potential overwrites.
         */
        ENABLED
    }
```

```
}
```

- `jakarta.mvc.form.FormMethodOverwrite` which can be either `ENABLED` or `DISABLED`. The legal options for this property are defined in `jakarta.mvc.form.FormMethodOverwriter.Options`. Its default value is `ENABLED`.
- `jakarta.mvc.form.HiddenFieldName` which defines the name of the hidden input field containing the HTTP method which shall be used instead of the original one. The default value `_method` is defined in `jakarta.mvc.form.FormMethodOverwriter#HIDDEN_FIELD_NAME_DEFAULT`.

The following sections will explain the form method overwrite resolving algorithm provided by the Jakarta MVC implementation.

9.2. Resolving Algorithm

The *form method overwriter* is responsible to detect if the HTTP method shall be overwritten and perform the overwrite if necessary. The specification won't provide an interface for this task, as there are a lot of possibilities provided by the specifications MVC is based on, like `HttpServletFilter` from Jakarta Servlet or Jakarta RESTful's `ContainerRequestFilter`.

Implementations MUST use the following algorithm to overwrite the HTTP method for each request:

1. Check if the following preconditions are `true`:
    a. The configuration property `jakarta.mvc.form.FormMethodOverwrite` is set to `FormMethodOverwriter.Options#ENABLED`
    b. The request is a `POST` request.
    c. A form field with the name like it's configured in `jakarta.mvc.form.HiddenFieldName` is available
2. If all conditions are resolved to true:
    a. Overwrite the HTTP method to the value provided by the hidden form field.
3. If any of these preconditions evaluates to `false`:
    a. End the procedure without changing the request's HTTP method

Applications can either rely on the form method overwriter algorithm which is described in this section or provide a custom form method overwriter which implements some other strategy.

Appendix A: Summary of Annotations

Annotation	Target	Description
`Controller`	Type or method	Defines a resource method as a Jakarta 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`.

Appendix B: Revision History

3.0

~

Remove requirement for Facelets support

2.1

~

Specified `FormMethodOverwrite` for extended HTTP method support

2.0

~

Changed namespace from `javax.mvc` to `jakarta.mvc`.

1.1

~

No API changes since 1.0.

Bibliography

[1]

Jakarta Server Faces 2.3, August 2019
https://jakarta.ee/specifications/faces/2.3/

[2]

Jakarta Context Dependency Injection 2.0, August 2019
https://jakarta.ee/specifications/cdi/2.0/

[3]

Jakarta Bean Validation 2.0, August 2019
https://jakarta.ee/specifications/bean-validation/2.0/

[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]

Jakarta RESTful Web Services 2.1, August 2019
https://jakarta.ee/specifications/restful-ws/2.1/

[6]

Jakarta Expression Language 3.0, August 2019
https://jakarta.ee/specifications/expression-language/3.0/