MicroProfile OpenAPI Specification

Arthur De Magalhaes (Spec Lead), Eric Wittmann, Anna Safonov, Matt Gill, Ivan Junckes Filho, Jérémie Bresson, Jana Manoharan, Rui Qi Wang, Tommy Wojtczak, Martin Smithson, Michael Edgar, Andrew Rouse

4.1.1, October 21, 2025: Final

Table of Contents

Specification: MicroProfile OpenAPI Specification

Version: 4.1.1

Status: Final

Release: October 21, 2025

Copyright

Eclipse Foundation Specification License - v1.1

Disclaimers

Chapter 1. Introduction

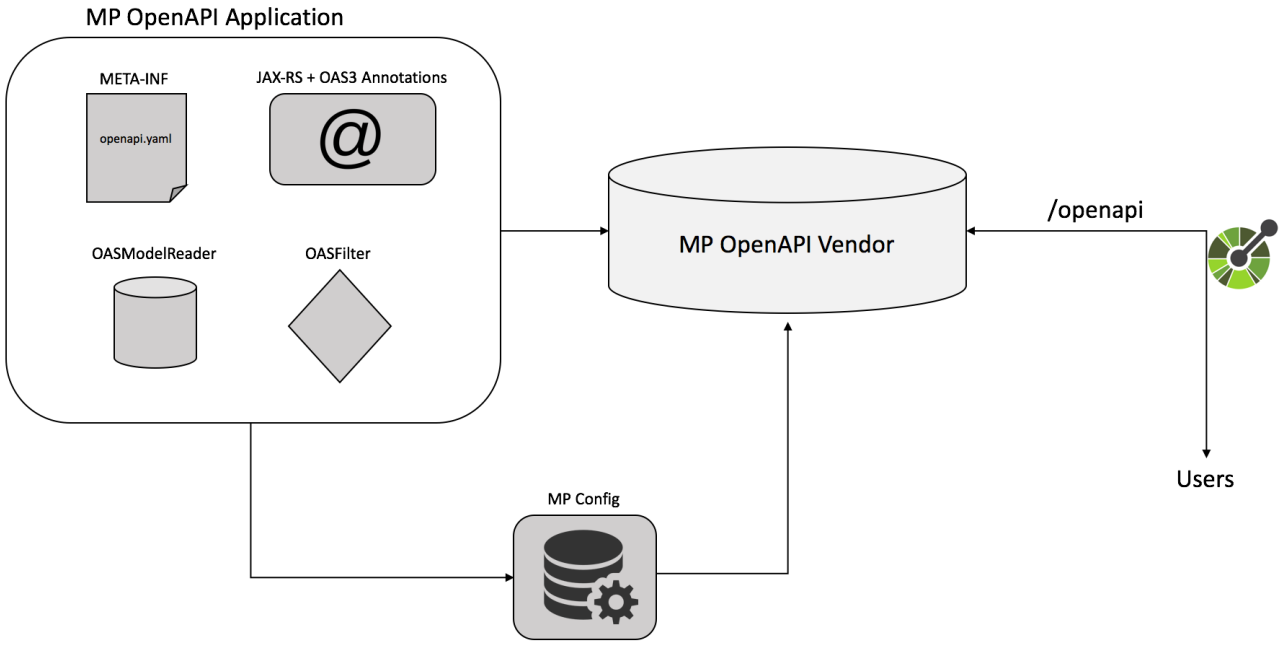
Exposing APIs has become an essential part of all modern applications. At the center of this revolution known as the API Economy we find RESTful APIs, which can transform any application into language agnostic services that can be called from anywhere: on-premises, private cloud, public cloud, etc.

For the clients and providers of these services to connect there needs to be a clear and complete contract. Similar to the WSDL contract for legacy Web Services, the OpenAPI v3.1 specification is the contract for RESTful Services.

This MicroProfile specification, called OpenAPI, aims to provide a set of Java interfaces and programming models which allow Java developers to natively produce OpenAPI v3.1 documents from their applications written using Jakarta RESTful Web Services (Jakarta REST).

Chapter 2. Architecture

There are different ways to augment a Jakarta REST application in order to produce an OpenAPI document, which are described in Documentation Mechanisms. The picture below provides a quick overview of the different types of components that make up the MP OpenAPI specification:



The remaining sections of this specification will go into the details of each component.

Chapter 3. Configuration

Configuration of various parts of this specification is provided via the MicroProfile Config mechanism, which means that vendors implementing the MP OpenAPI specification must also implement the MP Config specification.

There are various ways to inject these configuration values into an MP OpenAPI framework, including the default ConfigSource as well as custom ConfigSource.

Vendors implementing the MP OpenAPI specification can optionally provide additional native ways for these configuration values to be injected into the framework (e.g. via a server configuration file), as long as they also implement the MP Config specification.

3.1. List of configurable items

Vendors must support all the Core configurations of this specification. Optionally, they may also support Vendor extensions that allow the configuration of framework-specific values for configurations that affect implementation behavior.

For convenience of vendors (and application developers using custom ConfigSources), the full list of supported configuration keys is available as constants in the OASConfig class.

3.1.1. Core configurations

The following is a list of configuration values that every vendor must support.

`mp.openapi.model.reader`

Configuration property to specify the fully qualified name of the OASModelReader implementation.

`mp.openapi.filter`

Configuration property to specify the fully qualified name of the OASFilter implementation.

`mp.openapi.scan.disable`

Configuration property to disable annotation scanning. Default value is `false`.

`mp.openapi.scan.packages`

Configuration property to specify the list of packages to scan. Classes within the package and any subpackages will be scanned for annotations. For example, `mp.openapi.scan.packages=com.xyz.packageA,com.xyz.packageB`

`mp.openapi.scan.classes`

Configuration property to specify the list of classes to scan. For example, `mp.openapi.scan.classes=com.xyz.MyClassA,com.xyz.MyClassB`

`mp.openapi.scan.exclude.packages`

Configuration property to specify the list of packages to exclude from scans. Classes within the package and any subpackages will be excluded from scans. For example,

`mp.openapi.scan.exclude.packages=com.xyz.packageC,com.xyz.packageD`

`mp.openapi.scan.exclude.classes`

Configuration property to specify the list of classes to exclude from scans. For example, `mp.openapi.scan.exclude.classes=com.xyz.MyClassC,com.xyz.MyClassD`

The following rules are used to determine whether a class is scanned for annotations:

1. A class is not scanned if it's listed in `mp.openapi.scan.exclude.classes`
2. A class is scanned if it's listed in `mp.openapi.scan.classes`
3. A class is not scanned if its package, or any of its parent packages are listed in `mp.openapi.scan.exclude.packages`, unless a more complete package or parent package is listed in `mp.openapi.scan.packages`
4. A class is scanned if its package or any of its parent packages are listed in `mp.openapi.scan.packages`
5. A class is scanned if `mp.openapi.scan.classes` and `mp.openapi.scan.packages` are both empty or not set

`mp.openapi.scan.beanvalidation`

Configuration property to enable or disable the scanning and processing of Jakarta Bean Validation annotations. Defaults to `true`.

`mp.openapi.servers`

Configuration property to specify the list of global servers that provide connectivity information. For example, `mp.openapi.servers=https://xyz.com/v1,https://abc.com/v1`

`mp.openapi.servers.path.`

Prefix of the configuration property to specify an alternative list of servers to service all operations in a path. For example, `mp.openapi.servers.path./airlines/bookings/{id}=https://xyz.io/v1`

`mp.openapi.servers.operation.`

Prefix of the configuration property to specify an alternative list of servers to service an operation. Operations that want to specify an alternative list of servers must define an `operationId`, a unique string used to identify the operation. For example, `mp.openapi.servers.operation.getBooking=https://abc.io/v1`

`mp.openapi.schema.`

Prefix of the configuration property to specify a schema for a specific class, in JSON format. The remainder of the property key must be the fully-qualified class name. The value must be a valid OpenAPI schema object, specified in the JSON format. The use of this property is functionally equivalent to the use of the `@Schema` annotation on a Java class, but may be used in cases where the application developer does not have access to the source code of a class.

When a `name` key is provided with a string value, the schema will be added to the `schemas` collection in the `components` object of the resulting OpenAPI document using `name`'s value as the key.

For example, in the case where an application wishes to represent Java `Date`s in epoch milliseconds, the following configuration could be used (line escapes and indentation added for readability):

```
mp.openapi.schema.java.util.Date = { \
  "name": "EpochMillis", \
  "type": "number", \
  "format": "int64", \
  "description": "Milliseconds since January 1, 1970, 00:00:00 GMT" \
}
```

3.1.2. Vendor extensions

Vendors that wish to provide vendor-specific configuration via MP Config (instead of another native configuration framework) must use the prefix `mp.openapi.extensions`.

Chapter 4. Documentation Mechanisms

There are many different ways to provide input for the generation of the resulting OpenAPI document.

The MP OpenAPI specification requires vendors to produce a valid OpenAPI document from pure Jakarta REST applications. This means that vendors must process all the relevant Jakarta REST annotations (such as `@Path` and `@Consumes`) as well as Java objects (POJOs) used as input or output to Jakarta REST operations. This is a good place to start for application developers that are new to OpenAPI: just deploy your existing Jakarta REST application into a MP OpenAPI vendor and check out the output from `/openapi`!

The application developer then has a few choices:

1. Augment those Jakarta REST annotations with the OpenAPI Annotations. Using annotations means developers don't have to re-write the portions of the OpenAPI document that are already covered by the Jakarta REST framework (e.g. the HTTP method of an operation).
2. Take the initial output from `/openapi` as a starting point to document your APIs via Static OpenAPI files. It's worth mentioning that these static files can also be written before any code, which is an approach often adopted by enterprises that want to lock-in the contract of the API. In this case, we refer to the OpenAPI document as the "source of truth", by which the client and provider must abide.
3. Use the Programming model to provide a bootstrap (or complete) OpenAPI model tree.

Additionally, a Filter is described which can update the OpenAPI model after it has been built from the previously described documentation mechanisms.

4.1. Annotations

Many of these annotations were derived from the Swagger Core library, which allows for a mostly-mechanical transformation of applications that are using that library and wish to take advantage to the official MP OpenAPI interfaces.

4.1.1. Quick overview of annotations

The following annotations are found in the org.eclipse.microprofile.openapi.annotations package.

Annotation	Description
@Callback	Represents a callback URL that will be invoked.
@Callbacks	Represents an array of Callback URLs that can be invoked.
@CallbackOperation	Represents an operation that will be invoked during the callback.
@Components	A container that holds various reusable objects for different aspects of the OpenAPI Specification.
@Explode	Enumeration used to define the value of the `explode` property.

Annotation	Description
@ParameterIn	Enumeration representing the parameter's `in` property.
@ParameterStyle	Enumeration for the parameter's `style` property.
@SecuritySchemeIn	Enumeration for the security scheme's `in` property.
@SecuritySchemeType	Enumeration for the security scheme's `type` property.
@Extension	Adds an extension with contained properties.
@Extensions	Adds custom properties to an extension.
@ExternalDocumentation	References an external resource for extended documentation.
@Header	Describes a single header object.
@Contact	Contact information for the exposed API.
@Info	This annotation encapsulates metadata about the API.
@License	License information for the exposed API.
@Link	Represents a design-time link for a response.
@LinkParameter	Represents a parameter to pass to the linked operation.
@Content	Provides schema and examples for a particular media type.
@DependentRequired	Used within `@Schema` to indicate properties that are required if another property is present.
@DependentSchema	Used within `@Schema` to indicate additional rules that are required if a named property is present.
@DiscriminatorMapping	Used to differentiate between other schemas which may satisfy the payload description.
@Encoding	Single encoding definition to be applied to single Schema Object.
@ExampleObject	Illustrates an example of a particular content.
@PatternProperty	Used within `@Schema` to define validation rules for properties whose names match a regular expression.
@Schema	Allows the definition of input and output data types.
@SchemaProperty	Allows the definition of a property nested within a parent @Schema.
@OpenAPIDefinition	General metadata for an OpenAPI definition.
@Operation	Describes an operation or typically a HTTP method against a specific path.
@Parameter	Describes a single operation parameter.
@Parameters	Encapsulates input parameters.
@RequestBody	Describes a single request body.

Annotation	Description
@RequestBodySchema	Describes a single request body with schema implementation class.
@PathItem	Describes a set of operations available at the same location. Mostly only used to document webhooks as the paths for operations within the application can be discovered from Jakarta REST resources.
@PathItemOperation	Used within `@PathItem` to describe an operation.
@APIResponse	Describes a single response from an API operation.
@APIResponses	A container for multiple responses from an API operation.
@APIResponseSchema	Describes a single response with schema implementation class from an API operation.
@OAuthFlow	Configuration details for a supported OAuth Flow.
@OAuthFlows	Allows configuration of the supported OAuth Flows.
@OAuthScope	Represents an OAuth scope.
@SecurityRequirement	Specifies a security requirement for an operation.
@SecurityRequirements	Represents an array of security requirements where only one needs to be satisfied.
@SecurityRequirementsSet	Represents an array of security requirements that need to be satisfied.
@SecurityScheme	Defines a security scheme that can be used by the operations.
@SecuritySchemes	Represents an array of security schemes that can be specified.
@Server	Represents a server used in an operation or used by all operations in an OpenAPI document.
@Servers	A container for multiple server definitions.
@ServerVariable	Represents a server variable for server URL template substitution.
@Tag	Represents a tag for the API endpoint.
@Tags	A container of multiple tags.

4.1.1.1. Overrides

When the same annotation is used on a class and a method, the values from the method instance will take precedence for that particular method. This commonly occurs with the `@Server` and `@Tag` annotations.

In other cases, such as with `@Parameter` and `@RequestBody`, the annotation values from the method's parameters takes precedence over corresponding annotation values from the method itself - in this scenario the combined usage of these annotations is allowed but discouraged, as it is error prone.

The `@Schema` annotation has a complex set of possible combinations. It can placed on POJOs (and

their fields / methods) and referenced from many other annotations. In the event that a `@Schema#implementation` value points to a POJO that also contains a `@Schema` annotation, the values are merged but with precedence given to the referrer annotation (i.e. the one that contains the `implementation` key). This allows POJO models to be reusable and configurable.

4.1.2. Detailed usage of key annotations

4.1.2.1. Operation

Sample 1 - Simple operation description

```
@GET
@Path("/findByStatus")
@Operation(summary = "Finds Pets by status",
           description = "Multiple status values can be provided with comma separated
strings")
public Response findPetsByStatus(...) { ... }
```

Output for Sample 1

```
/pet/findByStatus:
  get:
    summary: Finds Pets by status
    description: Multiple status values can be provided with comma separated strings
    operationId: findPetsByStatus
```

Sample 2 - Operation with different responses

```
@GET
@Path("/{username}")
@Operation(summary = "Get user by user name")
@APIResponse(description = "The user",
             content = @Content(mediaType = "application/json",
                                schema = @Schema(implementation = User.class))),
@APIResponse(responseCode = "400", description = "User not found")
public Response getUserByName(
        @Parameter(description = "The name that needs to be fetched. Use user1 for
testing. ", required = true) @PathParam("username") String username)
 {...}
```

Output for Sample 2

```
/user/{username}:
    get:
      summary: Get user by user name
      operationId: getUserByName
      parameters:
      - name: username
        in: path
```

```
        description: 'The name that needs to be fetched. Use user1 for testing. '
        required: true
        schema:
          type: string
      responses:
        default:
          description: The user
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
        400:
          description: User not found
```

4.1.2.2. RequestBody

Sample 1 - Simple RequestBody

```
@POST
@Path("/user")
@Operation(summary = "Create user",
           description = "This can only be done by the logged in user.")
public Response methodWithRequestBody(
          @RequestBody(description = "Created user object", required = true,
                       content = @Content(schema = @Schema(implementation = User.
class))) User user,
          @QueryParam("name") String name, @QueryParam("code") String code)
          { ... }
```

Output for Sample 1

```
post:
    summary: Create user
      description: This can only be done by the logged in user.
      operationId: methodWithRequestBody
    parameters:
    - name: name
      in: query
      schema:
      type: string
    - name: code
      in: query
      schema:
      type: string
    requestBody:
      description: Created user object
      content:
        '*/*':
          schema:
            $ref: '#/components/schemas/User'
```

```
      required: true
    responses:
      default:
        description: no description
```

4.1.2.3. Servers

Sample 1 - Extended Server scenarios

```
@OpenAPIDefinition(
    servers = {
        @Server(
            description = "definition server 1",
            url = "http://{var1}.definition1/{var2}",
            variables = {
                @ServerVariable(name = "var1",
                                description = "var 1",
                                defaultValue = "1",
                                enumeration = {"1", "2"}),
                @ServerVariable(name = "var2",
                                description = "var 2",
                                defaultValue = "1",
                                enumeration = {"1", "2"})})})
@Server(
    description = "class server 1",
    url = "http://{var1}.class1/{var2}",
    variables = {
            @ServerVariable(
                      name = "var1",
                      description = "var 1",
                      defaultValue = "1",
                      enumeration = {"1", "2"}),
            @ServerVariable(
                      name = "var2",
                      description = "var 2",
                      defaultValue = "1",
                      enumeration = {"1", "2"})})
@Server(
    description = "class server 2",
    url = "http://{var1}.class2",
    variables = {
            @ServerVariable(
                       name = "var1",
                       description = "var 1",
                       defaultValue = "1",
                       enumeration = {"1", "2"})})
public class ServersResource {

    @GET
    @Path("/")
```

```
    @Server(
        description = "method server 1",
        url = "http://{var1}.method1",
        variables = {
            @ServerVariable(
                      name = "var1",
                      description = "var 1",
                      defaultValue = "1",
                      enumeration = {"1", "2"})})
    @Server(
        description = "method server 2",
        url = "http://method2"
    )
    public Response getServers() {
        return Response.ok().entity("ok").build();
    }
}
```

Output for Sample 1

```
openapi: 3.1.0
servers:
- url: http://{var1}.definition1/{var2}
  description: definition server 1
  variables:
    var1:
      description: var 1
      enum:
      - "1"
      - "2"
      default: "1"
    var2:
      description: var 2
      enum:
      - "1"
      - "2"
      default: "1"
paths:
  /:
    get:
      operationId: getServers
      responses:
        default:
          description: default response
      servers:
      - url: http://{var1}.class1/{var2}
        description: class server 1
        variables:
          var1:
            description: var 1
```

```
            enum:
            - "1"
            - "2"
            default: "1"
          var2:
            description: var 2
            enum:
            - "1"
            - "2"
            default: "1"
      - url: http://{var1}.class2
        description: class server 2
        variables:
          var1:
            description: var 1
            enum:
            - "1"
            - "2"
            default: "1"
      - url: http://{var1}.method1
        description: method server 1
        variables:
          var1:
            description: var 1
            enum:
            - "1"
            - "2"
            default: "1"
      - url: http://method2
        description: method server 2
        variables: {}
```

4.1.2.4. Schema

Sample 1 - Schema POJO

```
@Schema(name="MyBooking", description="POJO that represents a booking.")
public class Booking {
    @Schema(required = true, example = "32126319")
    private String airMiles;

    @Schema(required = true, example = "window")
    private String seatPreference;
}
```

Output for Sample 1

```
components:
  schemas:
    MyBooking:
```

```
      description: POJO that represents a booking.
      required:
      - airMiles
      - seatPreference
      type: object
      properties:
        airMiles:
          type: string
          example: "32126319"
        seatPreference:
          type: string
          example: window
```

Sample 2 - Schema POJO reference

```
@POST
public Response createBooking(
     @RequestBody(description = "Create a new booking.",
                  content = @Content(mediaType = "application/json",
                                     schema = @Schema(implementation = Booking.class)
)) Booking booking) {
```

Output for Sample 2

```
post:
  operationId: createBooking
  requestBody:
    description: Create a new booking.
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/MyBooking'
```

For more samples please see the MicroProfile Wiki.

4.1.3. Jakarta Bean Validation Annotations

In some cases, additional schema restrictions can be inferred from Jakarta Bean Validation annotations and used to enhance the generated OpenAPI document.

If an implementation includes support for the Jakarta Bean Validation specification, then it must also process Jakarta Bean Validation annotations when creating OpenAPI schemas. Such implementations must add the properties listed in the table below to the schema model when:

- the annotation is applied to to an element for which a schema is generated and
- the annotation and generated schema type are listed together in the table below and
- the annotation has a `group` attribute which is empty or includes `jakarta.validation.groups.Default` and

- the user has not set any of the relevant property values using other annotations and
- processing of bean validation annotations has not been disabled via configuration

Annotation	Schema type	Schema properties to set
`@NotEmpty`	`string`	`minLength = 1`
`@NotEmpty`	`array`	`minItems = 1`
`@NotEmpty`	`object`	`minProperties = 1`
`@NotBlank`	`string`	`pattern = \S`
`@Size(min = a, max = b)`	`string`	`minLength = a` `maxLenth = b`
`@Size(min = a, max = b)`	`array`	`minItems = a` `maxItems = b`
`@Size(min = a, max = b)`	`object`	`minProperties = a` `maxProperties = b`
`@DecimalMax(value = a)`	`number` or `integer`	`maximum = a`
`@DecimalMax(value = a, inclusive = false)`	`number` or `integer`	`exclusiveMaximum = a`
`@DecimalMin(value = a)`	`number` or `integer`	`minimum = a`
`@DecimalMin(value = a, inclusive = false)`	`number` or `integer`	`exclusiveMinimum = a`
`@Max(a)`	`number` or `integer`	`maximum = a`
`@Min(a)`	`number` or `integer`	`minimum = a`
`@Negative`	`number` or `integer`	`exclusiveMaximum = 0`
`@NegativeOrZero`	`number` or `integer`	`maximum = 0`
`@Positive`	`number` or `integer`	`exclusiveMinimum = 0`
`@PositiveOrZero`	`number` or `integer`	`minimum = 0`

4.2. Static OpenAPI files

Application developers may wish to include a pre-generated OpenAPI document that was written separately from the code (e.g. with an editor such as this).

Depending on the scenario, the document may be fully complete or partially complete. If a document is fully complete then the application developer will want to set the `mp.openapi.scan.disable` configuration property to `true`. If a document is partially complete, then the application developer will need to augment the OpenAPI snippet with annotations, programming model, or via the filter.

4.2.1. Location and formats

Vendors are required to fetch a single document named `openapi` with an extension of `yml`, `yaml` or `json`, inside the application module's root `META-INF` folder. If there is more than one document found that matches one of these extensions the behavior of which file is chosen is undefined (i.e. each

vendor may implement their own logic), which means that application developers should only place a single `openapi` document into that folder.

For convenience, you may also place your `microprofile-config.properties` in the root `META-INF` folder, if you wish to keep both documents in the same directory. This is in addition to the default locations defined by MicroProfile Config.

4.3. Programming model

Application developers are able to provide OpenAPI elements via Java POJOs. The complete set of models are found in the org.eclipse.microprofile.openapi.models package.

4.3.1. OASFactory

The OASFactory is used to create all of the elements of an OpenAPI tree.

For example, the following snippet creates a simple Info element that contains a title, description, and version.

```
OASFactory.createObject(Info.class).title("Airlines").description("Airlines APIs")
.version("1.0.0");
```

4.3.2. OASModelReader

The OASModelReader interface allows application developers to bootstrap the OpenAPI model tree used by the processing framework. To use it, simply create an implementation of this interface and register it using the `mp.openapi.model.reader` configuration key, where the value is the fully qualified name of the reader class.

Sample META-INF/microprofile-config.properties

```
mp.openapi.model.reader=com.mypackage.MyModelReader
```

Similar to static files, the model reader can be used to provide either complete or partial model trees. If providing a complete OpenAPI model tree, application developers should set the `mp.openapi.scan.disable` configuration to `true`. Otherwise this partial model will be used as the base model during the processing of the other Documentation Mechanisms.

Vendors are required to call the OASReader a single time, in the order defined by the Processing rules section. Only a single OASReader instance is allowed per application.

4.4. Filter

There are many scenarios where application developers may wish to update or remove certain elements and fields of the OpenAPI document. This is done via a filter, which is called once after all other documentation mechanisms have completed.

4.4.1. OASFilter

The OASFilter interface allows application developers to receive callbacks for various key OpenAPI elements. The interface has a default implementation for every method, which allows application developers to only override the methods they care about. To use it, simply create an implementation of this interface and register it using the `mp.openapi.filter` configuration key, where the value is the fully qualified name of the filter class.

Sample META-INF/microprofile-config.properties

```
mp.openapi.filter=com.mypackage.MyFilter
```

Vendors are required to call the registered filter once for each filtered element. For example, the method `filterPathItem` is called **for each** corresponding `PathItem` element in the model tree. This allows application developers to filter the element and any of its descendants.

The order of filter methods called is undefined, with two exceptions:

1. All filterable descendant elements of a filtered element must be called before its ancestor.
2. The `filterOpenAPI` method must be the **last** method called on a filter (which is just a specialization of the first exception).

4.5. Processing rules

The processed document available from the OpenAPI Endpoint is built from a variety of sources, which were outlined in the sub-headings of Documentation Mechanisms. Vendors are required to process these different sources in the following order:

1. Fetch configuration values from `mp.openapi` namespace
2. Call OASModelReader
3. Fetch static OpenAPI file
4. Process annotations
5. Filter model via OASFilter

Example processing:

- A vendor starts by fetching all available Configuration. If an `OASModelReader` was specified in that configuration list, its `buildModel` method is called to form the starting OpenAPI model tree for this application.
- Any Vendor extensions are added on top of that starting model (overriding conflicts), or create a new model if an `OASModelReader` was not registered.
- The vendor searches for a file as defined in the section Static OpenAPI files. If found, it will read that document and merge with the model produced by previous processing steps (if any), where conflicting elements from the static file will override the values from the original model.
- If annotation scanning was not disabled, the Jakarta REST and OpenAPI annotations from the application will be processed, further overriding any conflicting elements from the current

model.

- The final model is filtered by walking the model tree and invoking all registered OASFilter classes.

Chapter 5. OpenAPI Endpoint

5.1. Overview

A fully processed OpenAPI document must be served from the root URL `/openapi` in response to an `HTTP GET` request if any of the following conditions are met:

- an `OASModelReader` has been configured with `mp.openapi.model.reader`
- an `OASFilter` has been configured with `mp.openapi.filter`
- one of the allowed static files is present, i.e. `META-INF/openapi.(json|yaml|yml)`
- the application uses Jakarta REST

For example, `GET http://myHost:myPort/openapi`.

This document represents the result of the applied Processing rules.

The protocol required is `http`. Vendors are encouraged, but not required, to support the `https` protocol as well, to enable a secure connection to the OpenAPI endpoint.

5.2. Content format

The default format of the `/openapi` endpoint is `YAML`.

Vendors must also support the `JSON` format if the request contains an `Accept` header with a value of `application/json`, in which case the response must contain a `Content-Type` header with a value of `application/json`.

5.3. Query parameters

No query parameters are required for the `/openapi` endpoint. However, one suggested but optional query parameter for vendors to support is `format`, where the value can be either `JSON` or `YAML`, to facilitate the toggle between the default `YAML` format and `JSON` format.

5.4. Context root behavior

Vendors are required to ensure that the combination of each global server element and pathItem element resolve to the absolute backend URL of that particular path. If that `pathItem` contains a `servers` element , then this list of operation-level `server` elements replaces the global list of servers for that particular `pathItem`.

For example: an application may have an `ApplicationPath` annotation with the value of `/`, but is assigned the context root of `/myApp` during deployment. In this case, the `server` elements (either global or operation-level) must either end with `/myApp` or a corresponding proxy. Alternatively it is valid, but discouraged, to add that context root (`/myApp`) to every `pathItem` defined in that application.

5.5. Multiple applications

The MicroProfile OpenAPI specification does not define how the `/openapi` endpoint may be partitioned in the event that the MicroProfile runtime supports deployment of multiple applications. If an implementation wishes to support multiple applications within a MicroProfile runtime, the semantics of the `/openapi` endpoint are expected to be the logical union of all the applications in the runtime, which would imply merging multiple OpenAPI documents into a single valid document (handling conflicting IDs and unique names).

5.6. User Interface

Vendors may provide a separate interface to allow users to vizualize or browse the contents of the OpenAPI document. If such a user interface is provided, it should be made available at `/openapi/ui`.

Chapter 6. Integration with other MicroProfile specifications

This section will outline specific integrations between MicroProfile OpenAPI and other MicroProfile specifications.

6.1. MicroProfile Rest Client

It is common that a microservice (A) using MicroProfile OpenAPI will also use MicroProfile Rest Client to make outbound calls into another microservice (B). In this case, we do not want the interface for microservice (B) to appear in microservice (A)'s OAS3 document.

Therefore, vendors are required to exclude from the final OAS3 document any interface annotated with org.eclipse.microprofile.rest.client.inject.RegisterRestClient.

Chapter 7. Limitations

7.1. Internationalization

The MicroProfile OpenAPI spec does not require vendors to support multiple languages based on the `Accept-Language`. One reasonable approach is for vendors to support unique keys (instead of hardcoded text) via the various Documentation Mechanisms, so that the implementing framework can perform a global replacement of the keys with the language-specific text that matches the `Accept-Language` request for the `/openapi` endpoint. A cache of processed languages can be kept to improve performance.

7.2. Validation

The MP OpenAPI specification does not mandate vendors to validate the resulting OpenAPI v3.1 model (after processing the 5 steps previously mentioned), which means that the behavior of invalid models is vendor specific (i.e. vendors may choose to ignore, reject, or pass-through invalid inputs).

7.3. Cross Origin Resource Sharing (CORS)

The MP OpenAPI specification does not mandate but recommends vendors support CORS for the `/openapi` endpoint. Without CORS support, tools such as Swagger-UI might experience some errors. However, the behavior of CORS requests is implementation dependent.

Chapter 8. Release Notes

8.1. Release Notes for MicroProfile OpenAPI 4.1

A full list of changes delivered in the 4.1 release can be found at MicroProfile OpenAPI 4.1 Milestone

8.1.1. API/SPI changes

- Model API changes
  - New `OpenAPI` property `jsonSchemaDialect` (660)
  - New methods added to `Extensible`: `getExtension(String)` and `hasExtension(String)` (666)
- Add `@Target` to `@DependentRequired`, `@DependentSchema` and `@SchemaProperty` where it was missing (676)

8.2. Release Notes for MicroProfile OpenAPI 4.0

A full list of changes delivered in the 4.0 release can be found at MicroProfile OpenAPI 4.0 Milestone

8.2.1. Incompatible Changes

- `/openapi` endpoint now serves documentation in OpenAPI v3.1 format (333)
- Incompatible changes to the `Schema` model API, reflecting changes in the OpenAPI v3.1 document format (584)
  - `type` property type changed from `SchemaType` to `List<SchemaType>`
  - `exclusiveMinimum` and `exclusiveMaximum` property types changed from `Boolean` to `BigDecimal`
  - `nullable` property removed (replaced by the addition of `NULL` to `SchemaType`)
- Default value of `@RequestBody.required` changed to `true` to reflect that this is the much more common case where a RESTful resource method accepts a request body (349)
- Minimum Java version increased to 11

8.2.2. API/SPI changes

- Model API changes, reflecting changes in the OpenAPI v3.1 document format
  - New `OpenAPI` property: `webhooks` (583)
  - New `Components` property: `pathItems` (437)
  - New `Info` property: `summary` (435)
  - New `License` property: `identifier` (436)
  - New `Schema` properties: `booleanSchema`, `comment`, `constValue`, `contains`, `contentEncoding`, `contentMediaType`, `contentSchema`, `dependentRequired`, `dependentSchemas`, `elseSchema`, `examples`,

ifSchema, maxContains, minContains, patternProperties, prefixItems, propertyNames, schemaDialect, thenSchema, unevaluatedItems, unevaluatedProperties (584), (567)

- New Schema methods for working with custom properties: set(String, Object), get(String), setAll(Map<String, ?>), getAll() (584)
- New Schema.SchemaType enum value: NULL (584)
- New SecuritySchema.Type enum value: MUTUALTLS (582)

- Annotation API changes, reflecting changes in the OpenAPI v3.1 document format
  - New @OpenAPIDefinition property: webhooks (583)
  - New @Components property: pathItems (437)
  - New annotation @PathItem (437)
  - New annotation @PathItemOperation (437)
  - New @Callback property: pathItemRef (437)
  - New @Info property: summary (435)
  - New @License property: identifier (436)
  - New @Schema properties: comment, constValue, contains, contentEncoding, contentMediaType, contentSchema, dependentRequired, dependentSchemas, elseSchema, examples, ifSchema, maxContains, minContains, patternProperties, prefixItems, propertyNames, thenSchema (584), (567)
  - New @SchemaProperty properties: additionalProperties, comment, constValue, contains, contentEncoding, contentMediaType, contentSchema, dependentRequired, dependentSchemas ,elseSchema, examples, ifSchema, maxContains, minContains, patternProperties, prefixItems, propertyNames, thenSchema (584)
  - New annotations supporting the new @Schema properties: @DependentRequired, @DependentSchema, @PatternProperty (584), (567)
  - New SecuritySchemeType enum value: MUTUALTLS (582)
- Added module-info to the API jar (577)

8.2.3. Other changes

- Update references to the OpenAPI spec to point to v3.1 (606)
- Update documentation and TCKs to reflect changes in OpenAPI v3.1 which don't affect the model API
  - All security schemes may define required roles (590)
  - Summary and description are now valid when $ref is set (589)
  - Operation.requestBody permitted for HTTP methods which don't allow a request body (591)
  - Only one of Paths, Components, or Webhooks is required (592)
  - New encoding options for multipart/form-data (587)
  - New parameter style values valid for object type (586)
  - Operation no longer requires responses (585)

- Replace references to "JAX-RS" with "Jakarta RESTful Web Services" (574)

8.3. Release Notes for MicroProfile OpenAPI 3.1

A full list of changes delivered in the 3.1 release can be found at MicroProfile OpenAPI 3.1 Milestone.

8.3.1. API/SPI Changes

- Add `extensions` attribute to most annotations (387)
- Improvements to the definition of security requirements (483, 468)
  - Define behavior of `@SecurityRequirementsSet` and make it repeatable
  - Clarify that a individual `@SecurityRequirement` annotation applied to a class or method is equivalent to a `@SecurityRequirementsSet` annotation containing that `@SecurityRequirement` annotation
  - Add `securitySets` attribute to `@OpenAPIDefinition` and `@CallbackOperation`
- Add `additionalProperties` attribute to `@Schema` (423)
- Allow `@APIResponse` to be applied to a class, indicating that every resource method on that class has that response (417)

8.3.2. Other Changes

- Add processing of some Jakarta Bean Validation annotations (482)
- Define the precedence of the `mp.openapi.scan.*` config properties (422)
- Clarify that the `name` attribute of `@Extension` must include the `x-` prefix (339)
- Only require that the `/openapi` endpoint is made available if there is documentation to show (413)
- Recommend a standard endpoint for implementations which provide a user interface (334)
- Recommend that implementations provide a way to serve CORS headers on the `/openapi` endpoint (416)

8.4. Release Notes for MicroProfile OpenAPI 3.0

A full list of changes delivered in the 3.0 release can be found at MicroProfile OpenAPI 3.0 Milestone.

8.4.1. Incompatible Changes

This release aligns with Jakarta EE 9.1 (487), so it won't work with earlier versions of Jakarta or Java EE.

8.4.1.1. API/SPI Changes

There are no functional changes introduced in this release, except the dependency updating from javax to jakarta.

8.4.1.2. Other Changes

- Negative Test Scenario - @SchemaProperty Precedence Behaviour (466)
- Use MediaType.APPLICATION_JSON instead of application/json in some TCKs (471)
- TCK Tag Collection Test contains() side effect (453)
- TestNG 7.4.0 Assert.assertNotSame has a bug which causes ModelConstructionTest TCK to fail (494)

8.5. Release Notes for MicroProfile OpenAPI 2.0

A full list of changes delivered in the 2.0 release can be found at MicroProfile OpenAPI 2.0 Milestone.

8.5.1. Incompatible Changes

- Model interfaces that were deprecated in 1.1 have been removed:
  - `Scopes` - this interface was replaced with `Map<String, ServerVariable>` because it did not need to be extensible (328)
  - `ServerVariables` - this interface was replaced with `Map<String, ServerVariable>` because it did not need to be extensible (245)
- Model interfaces that are not extensible no longer extend `java.util.Map`:
  - `APIResponses` (248)
  - `Callback` (248)
  - `Content` (248)
  - `Path` (248)
  - `SecurityRequirement` (248)
- Methods on model interfaces that were deprecated) in 1.1 have been removed:
  - `APIResponses`
    - `addApiResponse(String name, APIResponse apiResponse)` - use `addAPIResponse(String, APIResponse)` instead (229)
    - `get(Object key)` - use `getAPIResponse(String)` instead (248)
    - `containsKey(Object key)` - use `hasAPIResponse(String)` instead (248)
    - `put(String key, PathItem value)` - use `addAPIResponse(String, APIResponse)` instead (248)
    - `putAll(Map<? extends String, ? extends PathItem> m)` - use `setAPIResponses(Map)` instead (248)
    - `remove(Object key)` - use `removeAPIResponse(String)` instead (248)

- `Callback`
  - `get(Object key)` - use `getPathItem(String)` instead (248)
  - `containsKey(Object key)` - use `hasPathItem(String)` instead (248)
  - `put(String key, PathItem value)` - use `addPathItem(String, PathItem)` instead (248)
  - `putAll(Map<? extends String, ? extends PathItem> m)` - use `setPathItems(Map)` instead (248)
  - `remove(Object key)` - use `removePathItem(String)` instead (248)
- `Content`
  - `get(Object key)` - use `getMediaType(String)` instead (248)
  - `containsKey(Object key)` - use `hasMediaType(String)` instead (248)
  - `put(String key, PathItem value)` - use `addMediaType(String, MediaType)` instead (248)
  - `putAll(Map<? extends String, ? extends PathItem> m)` - use `setMediaTypes(Map)` instead (248)
  - `remove(Object key)` - use `removeMediaType(String)` instead (248)
- `OASFactory`
  - `createScopes` - use `Map<String, String>` for scopes instead (328)
  - `createServerVariables` - use use `Map<String, ServerVariable>` for server variables instead (245)
- `OAuthFlow`
  - `setScopes(Scopes scopes)` - use `setScopes(Map)` instead (328)
  - `scopes(Scopes scopes)` - use `scopes(Map)` instead (328)
- `OpenAPI`
  - `path(String name, PathItem path)` - use `Paths#addPathItem(String, PathItem)` on `OpenAPI#getPaths` instead (247)
- `Path`
  - `get(Object key)` - use `getPathItem(String)` instead (248)
  - `containsKey(Object key)` - use `hasPathItem(String)` instead (248)
  - `put(String key, PathItem value)` - use `addPathItem(String, PathItem)` instead (248)
  - `putAll(Map<? extends String, ? extends PathItem> m)` - use `setPathItems(Map)` instead (248)
  - `remove(Object key)` - use `removePathItem(String)` instead (248)
- `PathItem`
  - `readOperations` - use `Map#values()` on `PathItem#getOperations()` instead (256)
  - `readOperationsMap` - use `getOperations()` instead (256)
- `Schema`
  - `getAdditionalProperties` - use `getAdditionalPropertiesSchema()` or

`getAdditionalPropertiesBoolean()` instead (257, 281)

- `setAdditionalProperties(Schema additionalProperties)` - use `setAdditionalPropertiesSchema(Schema)` instead (257, 281)
- `setAdditionalProperties(Boolean additionalProperties)` - use `setAdditionalPropertiesBoolean(Boolean)` instead (257, 281)
- `additionalProperties(Schema additionalProperties)` - use `additionalPropertiesSchema(Schema)` instead (257, 281)
- `additionalProperties(Boolean additionalProperties)` - use `additionalPropertiesBoolean(Boolean)` instead (257, 281)

◦ `SecurityRequirement`

- `get(Object key)` - use `getScheme(String)` instead (248)
- `containsKey(Object key)` - use `hasScheme(String)` instead (248)
- `put(String key, PathItem value)` - use `addScheme(String, List)` instead (248)
- `putAll(Map<? extends String, ? extends PathItem> m)` - use `setSchemes(Map)` instead (248)
- `remove(Object key)` - use `removeScheme(String)` instead (248)

◦ `Server`

- `setVariables(ServerVariables variables)` - use `setVariables(Map)` instead (245)
- `variables(ServerVariables variables)` - use `variables(Map)` instead (245)

8.5.2. API/SPI Changes

- The `@SchemaProperty` annotation has been added to allow the properties for a schema to be defined inline. (360). For example:

```
    @Schema(properties={
        @SchemaProperty(name="creditCard", required=true),
        @SchemaProperty(name="departureFlight", description="The departure flight
information."),
        @SchemaProperty(name="returningFlight")
    })
```

- The `@RequestBodySchema` annotation has been added to provide a shorthand mechanism to specify the schema for a request body (363). For example:

```
    @RequestBodySchema(MyRequestObject.class)
```

- The `@APIResponseSchema` annotation has been added to provide a shorthand mechanism to specify the schema for a response body (363). For example:

```
    @APIResponseSchema(MyResponseObject.class)
```

- The `mp.openapi.schema.*` MicroProfile Config property has been added to allow the schema for a specific class to be specified. This property would typically be used in cases where the application developer does not have access to the source code of a class (364). For example:

```
mp.openapi.schema.java.time.Instant = { \
    "name": "EpochSeconds", \
    "type": "number", \
    "format": "int64", \
    "title": "Epoch Seconds", \
    "description": "Number of seconds from the epoch of 1970-01-01T00:00:00Z" \
}
```

8.5.3. Functional Changes

- Getter methods on model interfaces that return a list or map now return a copy of the list/map containing the same items. This list/map CAN be immutable. (240)
- Setter methods on model interfaces that take a list or a map as a parameter MUST not use the list/map instance directly (284)

8.5.4. Other Changes

- JavaDoc updates to clarify the behaviour of getter methods on model interfaces that return a list or map ((240), 288)
- TCK updates to verify that getter methods on model interfaces return a list or map, return a copy of underlying collection ((240), 288)

8.6. Release Notes for MicroProfile OpenAPI 1.1

Changes include:

- the addition of the JAXRS 2.1 `PATCH` method
- automatic hide MicroProfile Rest Client interfaces
- `OASFactoryResolver` is now a proper `SPI` artifact
- builder methods now have default implementations
- `@Content` now supports a singular `example` field
- `@Extension` now has a `parseValue` field for complex values
- TCK updated to support newer `3.0.x` versions
- overall Javadoc enhancements (classes and packages)
- various other minor improvements to the annotations, models and TCK
    - bug fixes, documentation updates, more convenience methods, deprecations, etc.

8.7. Release Notes for MicroProfile OpenAPI 1.0

First official release of MP OpenAPI. Highlights of the release:

- set of annotations that covers the entire OpenAPI v3 specification when combined with JAX-RS annotations.
- set of OpenAPI v3 models covering the entire OpenAPI v3 specification, with corresponding APIs to provide a bootstrap or complete model tree.
- configuration injected via MicroProfile Config specification.
- ability to provide static (partial or complete) OpenAPI v3 files.
- definition of an HTTP endpoint, `/openapi`, that provides YAML and JSON representations of the generated OpenAPI v3 document.