MicroProfile OpenAPI Specification

Version 1.0-RC1, December 07, 2017

Table of Contents

Chapter 1. Introduction

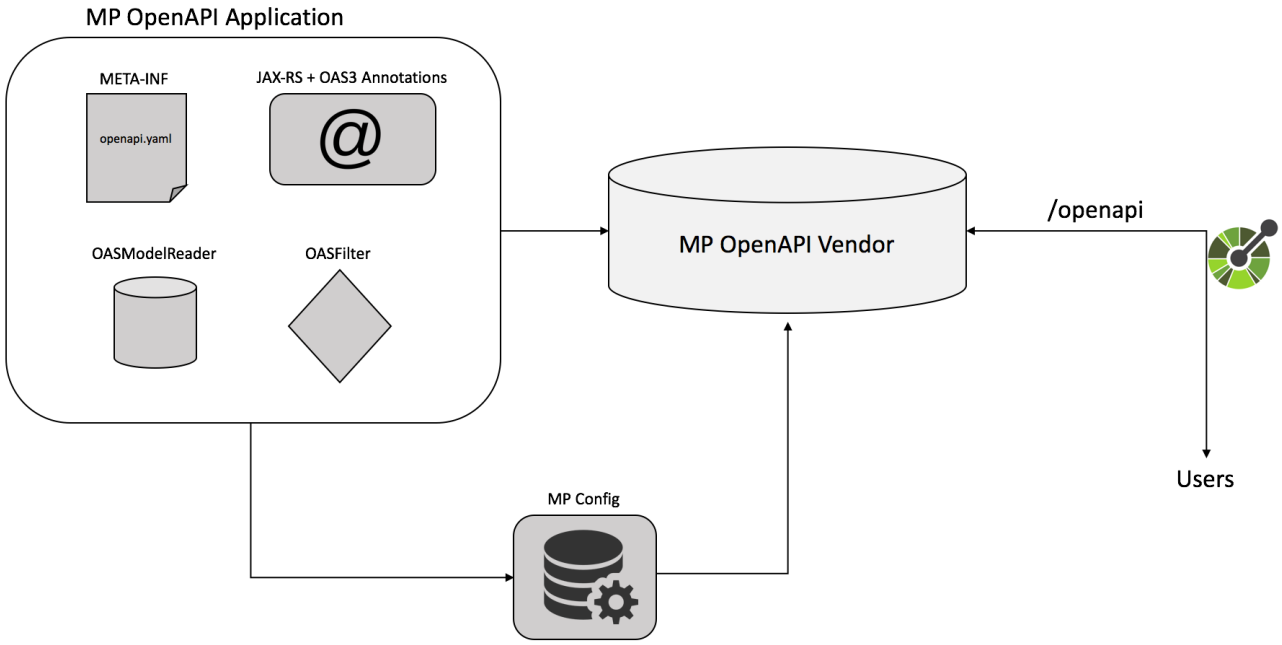
Exposing APIs has become an essential part of all modern applications. At the center of this revolution known as the API Economy we find RESTful APIs, which can transform any application into language agnostic services that can be called from anywhere: on-premises, private cloud, public cloud, etc.

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 specification is the contract for RESTful Services.

This MicroProfile specification, called OpenAPI 1.0, aims to provide a set of Java interfaces and programming models which allow Java developers to natively produce OpenAPI v3 documents from their JAX-RS applications.

Chapter 2. Architecture

There are different ways to augment a JAX-RS 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 also 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.

Config key	Value description
`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. 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. 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`
`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`

Config key	Value description
`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`

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 JAX-RS 2.0 applications. This means that vendors must process all the relevant JAX-RS annotations (such as `@Path` and `@Consumes`) as well as Java objects (POJOs) used as input or output to JAX-RS operations. This is a good place to start for application developers that are new to OpenAPI: just deploy your existing JAX-RS application into a MP OpenAPI vendor and check out the output from `/openapi`!

The application developer then has a few choices:

1. Augment those JAX-RS 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 JAX-RS 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 OpenAPI v3 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.
@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.
@ParameterIn	Enumeration representing the parameter's `in` property.
@ParameterStyle	Enumeration for the parameter's `style` property.

Annotation	Description
@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.
@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.
@Schema	Allows the definition of input and output data types.
@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.
@APIResponse	Describes a single response from an API operation.
@APIResponses	A container for multiple responses 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.

Annotation	Description
@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.

4.1.2. Detailed usage of key annotations

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",
           responses = {
                @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
```

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 methodWithRequestBodyAndTwoParameters(
          @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: methodWithRequestBodyAndTwoParameters
    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
```

Servers

Sample 1 - Extended Server scenarios

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

```
"1", allowableValues = {"1", "2"})
        })
@Server(
        description = "class server 2",
        url = "http://class2",
        variables = {
                @ServerVariable(name = "var1", description = "var 1", defaultValue =
"1", allowableValues = {"1", "2"})
        })
public class ServersResource {

    @GET
    @Path("/")
    @Operation(servers = {
            @Server(
                    description = "operation server 1",
                    url = "http://op1",
                    variables = {
                            @ServerVariable(name = "var1", description = "var 1",
defaultValue = "1", allowableValues = {"1", "2"})
                    })
    })
    @Server(
            description = "method server 1",
            url = "http://method1",
            variables = {
                    @ServerVariable(name = "var1", description = "var 1", defaultValue
= "1", allowableValues = {"1", "2"})
            })
    @Server(
            description = "method server 2",
            url = "http://method2"
    )

    public Response getServers() {
        return Response.ok().entity("ok").build();
    }
}
```

Output for Sample 1

```
openapi: 3.0.0
servers:
- url: http://definition1
  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://class1
        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://class2
        description: class server 2
        variables:
          var1:
            description: var 1
            enum:
            - "1"
            - "2"
            default: "1"
      - url: http://method1
        description: method server 1
        variables:
          var1:
            description: var 1
            enum:
            - "1"
            - "2"
            default: "1"
      - url: http://method2
        description: method server 2
        variables: {}
```

```
      - url: http://op1
        description: operation server 1
        variables:
          var1:
            description: var 1
            enum:
            - "1"
            - "2"
            default: "1"
```

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 with the provided information.
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/MyBooking'
```

For more samples please see the MicroProfile Wiki.

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's `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.

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`. Oherwise 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 all registered filters in the application (0..N) 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).

TODO: Document solution from issue #56 once ready

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

4.5.1. Validation

The MP OpenAPI 1.0 specification does not mandate vendors to validate the resulting OpenAPI v3 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).

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 specific configuration] 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 JAX-RS 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 and valid OpenAPI document must be available at the root URL `/openapi`, as a `HTTP GET` operation.

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

This document represents the result of the applied Processing rules.

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 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 behaviour

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 1.0 version of 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 AND 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).

Chapter 6. Limitations

6.1. Internationalization

The 1.0 version of 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.