# Table of Contents

1. Introduction .................................................................................................................. 2
2. Architecture .................................................................................................................... 3
3. Configuration ................................................................................................................... 4
   3.1. List of configurable items ......................................................................................... 4
      3.1.1. Core configurations ........................................................................................ 4
      3.1.2. Vendor extensions ......................................................................................... 5
4. Documentation Mechanisms ........................................................................................... 6
   4.1. Annotations ............................................................................................................... 6
      4.1.1. Quick overview of annotations ..................................................................... 6
      4.1.2. Detailed usage of key annotations .............................................................. 8
      4.2. Static OpenAPI files ............................................................................................ 8
      4.2.1. Location and formats .................................................................................... 8
   4.3. Programming model ................................................................................................ 10
      4.3.1. OASFactory ................................................................................................. 10
      4.3.2. OASModelReader ........................................................................................ 10
   4.4. Filter ........................................................................................................................ 13
      4.4.1. OASFilter .................................................................................................... 13
   4.5. Processing rules ....................................................................................................... 14
5. OpenAPI Endpoint ......................................................................................................... 14
   5.1. Overview .................................................................................................................. 14
   5.2. Content format ........................................................................................................ 14
   5.3. Query parameters ................................................................................................... 14
   5.4. Context root behavior ............................................................................................ 14
   5.5. Multiple applications .............................................................................................. 14
6. Limitations ...................................................................................................................... 16
   6.1. Internationalization ............................................................................................... 16
   6.2. Validation ................................................................................................................. 16
   6.3. Cross Origin Resource Sharing (CORS) ................................................................. 16
7. Release Notes for MicroProfile OpenAPI 1.0 ............................................................... 17
Specification: MicroProfile OpenAPI Specification

Version: 1.0

Status: Final

Release: December 15, 2017

Copyright (c) 2017 Contributors to the Eclipse Foundation

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
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 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.

<table>
<thead>
<tr>
<th>Config key</th>
<th>Value description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mp.openapi.model.reader</td>
<td>Configuration property to specify the fully qualified name of the OASModelReader implementation.</td>
</tr>
<tr>
<td>mp.openapi.filter</td>
<td>Configuration property to specify the fully qualified name of the OASFilter implementation.</td>
</tr>
<tr>
<td>mp.openapi.scan.disable</td>
<td>Configuration property to disable annotation scanning. Default value is false.</td>
</tr>
<tr>
<td>mp.openapi.scan.packages</td>
<td>Configuration property to specify the list of packages to scan. For example, mp.openapi.scan.packages=com.xyz.PackageA,com.xyz.PackageB</td>
</tr>
<tr>
<td>mp.openapi.scan.classes</td>
<td>Configuration property to specify the list of classes to scan. For example, mp.openapi.scan.classes=com.xyz.MyClassA,com.xyz.MyClassB</td>
</tr>
<tr>
<td>mp.openapi.scan.exclude.packages</td>
<td>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</td>
</tr>
<tr>
<td>mp.openapi.scan.exclude.classes</td>
<td>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</td>
</tr>
<tr>
<td>mp.openapi.servers</td>
<td>Configuration property to specify the list of global servers that provide connectivity information. For example, mp.openapi.servers=<a href="https://xyz.com/v1,https://abc.com/v1">https://xyz.com/v1,https://abc.com/v1</a></td>
</tr>
</tbody>
</table>
### 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`.  

<table>
<thead>
<tr>
<th>Config key</th>
<th>Value description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mp.openapi.servers.path.</code></td>
<td>Prefix of the configuration property to specify an alternative list of servers to service all operations in a path. For example, <code>mp.openapi.servers.path./airlines/bookings/{id}=https://xyz.io/v1</code></td>
</tr>
<tr>
<td><code>mp.openapi.servers.operation.</code></td>
<td>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 <code>operationId</code>, a unique string used to identify the operation. For example, <code>mp.openapi.servers.operation.getBooking=https://abc.io/v1</code></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Callback</td>
<td>Represents a callback URL that will be invoked.</td>
</tr>
<tr>
<td>@Callbacks</td>
<td>Represents an array of Callback URLs that can be invoked.</td>
</tr>
<tr>
<td>@CallbackOperation</td>
<td>Represents an operation that will be invoked during the callback.</td>
</tr>
<tr>
<td>@Components</td>
<td>A container that holds various reusable objects for different aspects of the OpenAPI Specification.</td>
</tr>
<tr>
<td>@Explode</td>
<td>Enumeration used to define the value of the explode property.</td>
</tr>
<tr>
<td>Annotation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@ParameterIn</td>
<td>Enumeration representing the parameter’s <em>in</em> property.</td>
</tr>
<tr>
<td>@ParameterStyle</td>
<td>Enumeration for the parameter’s <em>style</em> property.</td>
</tr>
<tr>
<td>@SecuritySchemeIn</td>
<td>Enumeration for the security scheme’s <em>in</em> property.</td>
</tr>
<tr>
<td>@SecuritySchemeType</td>
<td>Enumeration for the security scheme’s <em>type</em> property.</td>
</tr>
<tr>
<td>@Extension</td>
<td>Adds an extension with contained properties.</td>
</tr>
<tr>
<td>@Extensions</td>
<td>Adds custom properties to an extension.</td>
</tr>
<tr>
<td>@ExternalDocumentation</td>
<td>References an external resource for extended documentation.</td>
</tr>
<tr>
<td>@Header</td>
<td>Describes a single header object.</td>
</tr>
<tr>
<td>@Contact</td>
<td>Contact information for the exposed API.</td>
</tr>
<tr>
<td>@Info</td>
<td>This annotation encapsulates metadata about the API.</td>
</tr>
<tr>
<td>@License</td>
<td>License information for the exposed API.</td>
</tr>
<tr>
<td>@Link</td>
<td>Represents a design-time link for a response.</td>
</tr>
<tr>
<td>@LinkParameter</td>
<td>Represents a parameter to pass to the linked operation.</td>
</tr>
<tr>
<td>@Content</td>
<td>Provides schema and examples for a particular media type.</td>
</tr>
<tr>
<td>@DiscriminatorMapping</td>
<td>Used to differentiate between other schemas which may satisfy the payload description.</td>
</tr>
<tr>
<td>@Encoding</td>
<td>Single encoding definition to be applied to single Schema Object.</td>
</tr>
<tr>
<td>@ExampleObject</td>
<td>Illustrates an example of a particular content.</td>
</tr>
<tr>
<td>@Schema</td>
<td>Allows the definition of input and output data types.</td>
</tr>
<tr>
<td>@OpenAPIDefinition</td>
<td>General metadata for an OpenAPI definition.</td>
</tr>
<tr>
<td>@Operation</td>
<td>Describes an operation or typically a HTTP method against a specific path.</td>
</tr>
<tr>
<td>@Parameter</td>
<td>Describes a single operation parameter.</td>
</tr>
<tr>
<td>@Parameters</td>
<td>Encapsulates input parameters.</td>
</tr>
<tr>
<td>@RequestBody</td>
<td>Describes a single request body.</td>
</tr>
<tr>
<td>@APIResponse</td>
<td>Describes a single response from an API operation.</td>
</tr>
<tr>
<td>@APIResponses</td>
<td>A container for multiple responses from an API operation.</td>
</tr>
<tr>
<td>@OAuthFlow</td>
<td>Configuration details for a supported OAuth Flow.</td>
</tr>
<tr>
<td>@OAuthFlows</td>
<td>Allows configuration of the supported OAuth Flows.</td>
</tr>
<tr>
<td>@OAuthScope</td>
<td>Represents an OAuth scope.</td>
</tr>
<tr>
<td>@SecurityRequirement</td>
<td>Specifies a security requirement for an operation.</td>
</tr>
<tr>
<td>@SecurityRequirements</td>
<td>Represents an array of security requirements where only one needs to be satisfied.</td>
</tr>
<tr>
<td>Annotation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@SecurityRequirementsSet</td>
<td>Represents an array of security requirements that need to be satisfied.</td>
</tr>
<tr>
<td>@SecurityScheme</td>
<td>Defines a security scheme that can be used by the operations.</td>
</tr>
<tr>
<td>@SecuritySchemes</td>
<td>Represents an array of security schemes that can be specified.</td>
</tr>
<tr>
<td>@Server</td>
<td>Represents a server used in an operation or used by all operations in an OpenAPI document.</td>
</tr>
<tr>
<td>@Servers</td>
<td>A container for multiple server definitions.</td>
</tr>
<tr>
<td>@ServerVariable</td>
<td>Represents a server variable for server URL template substitution.</td>
</tr>
<tr>
<td>@Tag</td>
<td>Represents a tag for the API endpoint.</td>
</tr>
<tr>
<td>@Tags</td>
<td>A container of multiple tags.</td>
</tr>
</tbody>
</table>

**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

**Operation**

**Sample 1 - Simple operation description**

```java
@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
```

Output for Sample 2

```
@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) {
...
```

```
/users/{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
```
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

Servers

Sample 1 - Extended Server scenarios

@OpenAPIDefinition
  servers = {
    @Server(
      description = "definition server 1",
      url = "http://{var1}.definition1/{var2}",
      variables = {
        @ServerVariable(name = "var1",
..."}
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"})
    }
)
public Response getServers() {
    return Response.ok().entity("ok").build();
}
}
openapi: 3.0.0
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: get_servers
      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: {}

Schema

Sample 1 - Schema POJO

```java
@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

```json
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

```java
@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

```json
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.

4.2. 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 (i.e. WAR artifact) 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.

```java
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

```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

```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 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 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.

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 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 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).
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.

6.2. 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).

6.3. Cross Origin Resource Sharing (CORS)

The MP OpenAPI 1.0 specification does not require vendors to support CORS for the /openapi endpoint. The behavior of CORS requests is implementation dependent.
Chapter 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.