RIECKPIL

ENTERPRISE DEVELOPMENT WITH JAVA MADE SIMPLE

GETTING STARTED WITH Eclipse MicroProfile

All you need to know about each MicroProfile specification

Either use it standalone or optimize your Jakarta EE application for a cloud-native architecture

Getting Started with Eclipse MicroProfile E-Book

Philip Riecks
Eclipse MicroProfile Version: 3.3

Table of Contents

Getting Started with Eclipse MicroProfile

This E-Book guides you trough each of the twelve MicroProfile specifications in detail:

- MicroProfile Config
- MicroProfile Metrics
- MicroProfile OpenAPI
- MicroProfile OpenTracing
- MicroProfile Fault Tolerance
- MicroProfile Rest Client
- MicroProfile Health
- MicroProfile JWT Auth
- Contexts and Dependency Injection (CDI)
- Jakarta RESTful Web Services (JAX-RS)
- JSON Binding (JSON-B)
- JSON Processing (JSON-P)

After reading this book you'll ...

- Understand the concept and use of each MicroProfile specification
- Explore how to add missing parts (e.g. OpenAPI, JWT Auth, Resiliency, etc.) to an existing Enterprise Java application
- Discover and implement best practices using MicroProfile
- Avoid vendor-lock when implementing applications using Eclipse MicroProfile
- Discover why MicroProfile is a great spec for building cloud-native applications
- Use the practical know-how in your projects
- Get hands-on screencasts for each specification with further explanations

This Book is for ...

Java Developers that want to start using Eclipse MicroProfile or master dedicated Eclipse MicroProfile specifications. No additional knowledge required except basic experience with Java.

PS: If you find any typo or invalid example, please open an issue on GitHub.

Have fun reading this book & learning more about Eclipse MicroProfile,

Phil

Introduction to Eclipse MicroProfile

The Eclipse MicroProfile initiative was launched at Java One 2016 due to the slow pace of Java EE back at the time. Eclipse MicroProfile is a set of standardized specifications to add the missing parts to Enterprise Java to be cloud-ready: fault-tolerance, tracing, health checks, etc.

With three release each year the MicroProfile project moves fast forward and allows short feedback-cycles for changes to existing specifications or for a new specification.

Available implementations

Check the homepage of the different vendors for the currently supported MicroProfile version.

Application Servers

- Open Liberty
- Payara
- WildFly
- TomEE

Smaller runtimes

- KumuluzEE
- Helidon
- Quarkus
- Thorntail
- Hammock
- meecrowave

Technologies used for the examples in this books

- MicroProfile 3.3
- Open Liberty
- Java 11
- Maven 3.6
- WAD (Watch and Deploy) from Adam Bien (setup)
- JWTENIZR from Adam Bien

Twelve-part video series

For each Eclipse MicroProfile specification I've recorded a screencast to provide hands-on experience.

GETTING STARTED WITH ECLIPSE MICROPROFILE
YOUTUBE SERIES
MICROPROFILE
OPTIMIZING ENTERPRISE JAVA

Get access to the twelve-part video series for Eclipse MicroProfile here.

MicroProfile Config

Injecting configuration properties like JDBC URLs, passwords, usernames or hostnames from external sources is a common requirement for every application.

Inspired by the twelve-factor app principles you should store configuration in the environment (e.g. OS environment variables or config maps in Kubernetes). These external configuration properties can then be replaced for your different stages (dev/prod/test) with ease.

Using **MicroProfile Config** you can achieve this in a simple and extensible way.

Specification profile: MicroProfile Config

- Current version: **1.4** in **MicroProfile 3.3**
- GitHub repository
- Latest specification document
- Basic use case: **Inject configuration properties from external sources** (like property files, environment or system variables)

Injecting configuration properties

At several parts of your application, you might want to inject configuration properties to configure for example the base URL of a JAX-RS `Client`.

With MicroProfile Config you can inject a `Config` object using CDI and fetch a specific property by its key:

```java
public class BasicConfigurationInjection {

    @Inject
    private Config config;

    public void init(@Observes @Initialized(ApplicationScoped.class) Object init) {
        System.out.println(config.getValue("message", String.class));
    }

}
```

In addition, you can inject a property value to a member variable with the `@ConfigProperty` annotation and also specify a default value:

```java
public class BasicConfigurationInjection {

    @Inject
    @ConfigProperty(name = "message", defaultValue = "Hello World")
    private String message;

}
```

If you don't specify a `defaultValue`, and the application can't find a property value in the configured ConfigSources, your application will throw an error during startup:

```
The BackedAnnotatedField @Inject @ConfigProperty private
de.rieckpil.blog.BasicConfigurationInjection.value InjectionPoint dependency was not
resolved. Error: java.util.NoSuchElementException: CWMCG0015E: The property
not.existing.value was not found in the configuration.
at com.ibm.ws.microprofile.config.impl.AbstractConfig.getValue(AbstractConfig.java:175)
at internal classes
```

For a more resilient behaviour, or if the config property is optional, you can wrap the value with Java's `Optional<T>` class and check its existence during runtime:

```
public class BasicConfigurationInjection {

    @Inject
    @ConfigProperty(name = "my.app.password")
    private Optional<String> password;

}
```

Furthermore you can wrap the property with a `Provider<T>` for a more dynmaic injection. This ensure that each invocation of `Provider.get()` resolves the latest value from the underlying `Config` and you are able to change it during runtime.

```
public class BasicConfigurationInjection {

    @Inject
    @ConfigProperty(name = "my.app.timeout")
    private Provider<Long> timeout;

    public void init(@Observes @Initialized(ApplicationScoped.class) Object init) {
        System.out.println(timeout.get());
    }

}
```

For the key of the configuration property you might use the dot notation to prevent conflicts and seperate domains: `my.app.passwords.twitter`.

Configuration sources

The default ConfigSources are the following:

- System property (default ordinal: 400): passed with `-Dmessage=Hello` to the application
- Environment variables (default ordinal: 300): OS variables like `export MESSAGE=Hello`
- Property file (default ordinal: 100): file `META-INF/microprofile-config.properties`

Once the MicroProfile Config runtime finds a property in two places (e.g. property file and environment variable), the value with the higher ordinal source is chosen.

These default configuration sources should cover most of the use cases and support writing cloud-native applications. However, if you need any additional custom `ConfigSource`, you can plug-in your own (e.g. fetch configurations from a database or external service).

To provide you an example for a custom `ConfigSource`, I'm creating a static source which serves just two properties. Therefore you just need to implement the `ConfigSource` interface and its methods

```
public class CustomConfigSource implements ConfigSource {

    public static final String CUSTOM_PASSWORD = "CUSTOM_PASSWORD";
    public static final String MESSAGE = "Hello from custom ConfigSource";

    @Override
    public int getOrdinal() {
        return 500;
    }

    @Override
    public Map<String, String> getProperties() {
        Map<String, String> properties = new HashMap<>();
        properties.put("my.app.password", CUSTOM_PASSWORD);
        properties.put("message", MESSAGE);
        return properties;
    }

    @Override
    public String getValue(String key) {
        if (key.equalsIgnoreCase("my.app.password")) {
            return CUSTOM_PASSWORD;
        } else if (key.equalsIgnoreCase("message")) {
            return MESSAGE;
        }
        return null;
    }

    @Override
    public String getName() {
        return "randomConfigSource";
    }
}
```

To register this new `ConfigSource` you can either bootstrap a custom `Config` object with this source:

```
Config config = ConfigProviderResolver
.instance()
.getBuilder()
.addDefaultSources()
.withSources(new CustomConfigSource())
.addDiscoveredConverters()
.build();
```

or add the fully-qualified name of the class of the configuration source to the `org.eclipse.microprofile.config.spi.ConfigSource` file in `/src/main/resources/META-INF/services`:

```
de.rieckpil.blog.CustomConfigSource
```

Using the file approach, the custom source is now part of the `ConfigSources` by default.

Configuration converters

Internally the mechanism for MicroProfile Config is purely String-based. Type-safety is achieved with `Converter` classes. The specification provides default `Converter` for converting the configuration property into the known Java types: `Integer`, `Long`, `Float`, `Boolean`, `Byte`, `Short`, `Character`, `Double` and their primitive counterparts. Furthermore, you can also define a config value with the type `java.lang.Class`. All these built-in converters have the priority of 1.

In addition, there are built-in providers for converting properties into Arrays, Lists, `Optional<T>` and `Provider<T>`. If the default `Converter` doesn't match your requirements and you want e.g. to convert a property into a domain object, you can plug-in a custom `Converter<T>`. For example, I'll convert a config property into a `Token` instance:

```
public class Token {

    private String name;
    private String payload;

    public Token(String name, String payload) {
        this.name = name;
        this.payload = payload;
    }

    // getter & setter
}
```

The custom converter needs to implement the `Converter<Token>` interface. The converter method accepts a raw string value and returns the custom domain object, in this case, an instance of `Token`:

```
public class CustomConfigConverter implements Converter<Token> {

    @Override
    public Token convert(String value) {
        String[] chunks = value.split(",");
        Token result = new Token(chunks[0], chunks[1]);
        return result;
    }
}
```

To register this converter you can either build your own Config instance and add the converter manually:

```
int PRIORITY = 100;

Config config = ConfigProviderResolver
.instance()
.getBuilder()
.addDefaultSources()
.addDiscoveredConverters()
.withConverter(Token.class, PRIORITY, new CustomConfigConverter())
.build();
```

or you can add the fully-qualified name of the class of the converter to the `org.eclipse.microprofile.config.spi.Converter` file in `/src/main/resources/META-INF/services`:

```
de.rieckpil.blog.CustomConfigConverter
```

Once your converter is registered, you can start using it:

```
my.app.token=TOKEN_1337, SUPER_SECRET_VALUE
```

```
public class BasicConfigurationInjection {

    @Inject
    @ConfigProperty(name = "my.app.token")
    private Token token;

}
```

» For more hands-one experience with Eclipse MicroProfile Config, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

MicroProfile Metrics

Ensuring a stable operation of your application in production requires monitoring. Without monitoring, you have no insights about the internal state and health of your system and have to work with a black-box.

MicroProfile Metrics gives you the ability to not only monitor pre-defined metrics like JVM statistics but also create custom metrics to monitor e.g. key figures of your business. These metrics are then exposed via HTTP and ready to visualize on a dashboard and create appropriate alarms.

Specification profile: MicroProfile Metrics

- Current version: **2.3** in MicroProfile **3.3**
- GitHub repository
- Latest specification document
- Basic use case: Add custom metrics (e.g. timer or counter) to your application and expose them via HTTP

Default MicroProfile metrics defined in the specification

The specification defines one endpoint with three subresources to collect metrics from a MicroProfile application:

- The endpoint to collect all available metrics: `/metrics`
- Base (pre-defined by the specification) metrics: `/metrics/base`
- Application metrics: `/metrics/application` (optional)
- Vendor-specific metrics: `/metrics/vendor` (optional)

So you can either use the main `/metrics` endpoint and get all available metrics for your application or one of the sub-resources. The default media type for these endpoints is `text/plain` using the OpenMetrics format.

You are also able to get them as JSON if you specify the Accept header in your request as `application/json`. In the specification, you find a list of base metrics every MicroProfile Metrics compliant application server has to offer. These are mainly JVM, GC, memory, and CPU related metrics to monitor the infrastructure. The following output is the required amount of base metrics:

```
{
   "gc.total;name=scavenge":393,
   "gc.time;name=global":386,
   "cpu.systemLoadAverage":0.92,
   "thread.count":85,
   "classloader.loadedClasses.count":11795,
   "classloader.unloadedClasses.total":21,
   "jvm.uptime":985206,
   "memory.committedHeap":63111168,
   "thread.max.count":100,
   "cpu.availableProcessors":12,
   "classloader.loadedClasses.total":11816,
   "thread.daemon.count":82,
   "gc.time;name=scavenge":412,
   "gc.total;name=global":14,
   "memory.maxHeap":4182573056,
   "cpu.processCpuLoad":0.0017964831879557087,
   "memory.usedHeap":34319912
}
```

In addition, you are able to add metadata and tags to your metrics like in the output above for `gc.time` where `name=global` is a tag. You can use these tags to further separate a metric for multiple use cases.

Since Eclipse MicroProfile 3.3 there is now also a new (optional) base metric `REST.request`. This tracks the total count of requests and the total elapsed time spent at your JAX-RS endpoints. As this is an optional metric, it might not be available in every implementation.

Create a custom metric with MicroProfile Metrics

There are two ways for defining a custom metric with MicroProfile Metrics: using annotations or programmatically. The specification offers five different metric types:

- Timer: sampling the time for e.g. a method call
- Counter: monotonically counting e.g. invocations of a method
- Gauges: sample the value of an object e.g. current size of JMS queue
- Meters: tracking the throughput of e.g. a JAX-RS endpoint
- Histogram: calculate the distribution of a value e.g. the variance of incoming user agents

For simple use cases, you can make use of annotations and just add them to a method you want to monitor. Each annotation offers attributes to configure tags and metadata for the metric:

```java
@Counted(name = "bookCommentClientInvocations",
  description = "Counting the invocations of the constructor",
  displayName = "bookCommentClientInvoke",
  tags = {"usecase=simple"})
public BookCommentClient() {
}
```

If your monitoring use case requires a more dynamic configuration, you can programmatically create/update your metrics. For this, you just need to inject the `MetricRegistry` to your class:

```java
public class BookCommentClient {

    @Inject
    @RegistryType(type = MetricRegistry.Type.APPLICATION)
    private MetricRegistry metricRegistry;

    public String getBookCommentByBookId(String bookId) {
        Response response = this.bookCommentsWebTarget.path(bookId).request().get();
        this.metricRegistry.counter("bookCommentApiResponseCode"
           + response.getStatus()).inc();
        return response.readEntity(JsonObject.class).getString("body");
    }
}
```

Create a timer metric

If you want to track and sample the duration for a method call, you can make use of timers. You can add them with the `@Timed` annotation or using the `MetricRegistry`.

A good use case might be tracking the time for a call to an external service:

```java
@Timed(name = "getBookCommentByBookIdDuration")
public String getBookCommentByBookId(String bookId) {
    Response response = this.bookCommentsWebTarget.path(bookId).request().get();
    return response.readEntity(JsonObject.class).getString("body");
}
```

While using the timer metric type you'll also get a count of method invocations and mean/max/min/percentile calculations out-of-the-box:

```json
 "de.rieckpil.blog.BookCommentClient.getBookCommentByBookIdDuration": {
        "fiveMinRate": 0.000004243196464475842,
        "max": 3966817891,
        "count": 13,
        "p50": 737218798,
        "p95": 3966817891,
        "p98": 3966817891,
        "p75": 997698383,
        "p99": 3966817891,
        "min": 371079671,
        "fifteenMinRate": 0.005509550587308515,
        "meanRate": 0.003936521878196718,
        "mean": 1041488167.7031761,
        "p999": 3966817891,
        "oneMinRate": 1.1484886591525709e-24,
        "stddev": 971678361.3592016
}
```

Be aware that you get the result as nanoseconds if you request the JSON result and for the OpenMetrics format, you get seconds:

```
getBookCommentByBookIdDuration_rate_per_second 0.003756880727820997
getBookCommentByBookIdDuration_one_min_rate_per_second 7.980095572816848E-26
getBookCommentByBookIdDuration_five_min_rate_per_second 2.4892551645230856E-6
getBookCommentByBookIdDuration_fifteen_min_rate_per_second 0.004612201440656351
getBookCommentByBookIdDuration_mean_seconds 1.0414881677031762
getBookCommentByBookIdDuration_max_seconds 3.9668178910000003
getBookCommentByBookIdDuration_min_seconds 0.371079671
getBookCommentByBookIdDuration_stddev_seconds 0.9716783613592016
getBookCommentByBookIdDuration_seconds_count 13
getBookCommentByBookIdDuration_seconds{quantile="0.5"} 0.737218798
getBookCommentByBookIdDuration_seconds{quantile="0.75"} 0.997698383
getBookCommentByBookIdDuration_seconds{quantile="0.95"} 3.9668178910000003
getBookCommentByBookIdDuration_seconds{quantile="0.98"} 3.9668178910000003
getBookCommentByBookIdDuration_seconds{quantile="0.99"} 3.9668178910000003
getBookCommentByBookIdDuration_seconds{quantile="0.999"} 3.9668178910000003
```

Create a simple timer

As you saw it in the chapter above, the `@Timed` annotation already calculates throughput and percentile statistics. If you don't need this amount of data to e.g. reduce the bandwith, you can fallback on `@SimplyTimed`.

This annotation is similar to the already mentioned timer, but solely tracks how long an invocation took to complete and does not prepare any statistics for you:

```
@SimplyTimed(name = "getBookCommentByBookIdDuration")
public String getBookCommentByBookId(String bookId) {
    Response response = this.bookCommentsWebTarget.path(bookId).request().get();
    return response.readEntity(JsonObject.class).getString("body");
}
```

Create a counter metric

The next metric type is the simplest one: a counter. With the counter, you can track e.g. the number of invocations of a method:

```
@Counted
public String doFoo() {
    return "Duke";
}
```

In one of the previous MicroProfile Metrics versions, you were able to decrease the counter and have a not monotonic counter. As this caused confusion with the gauge metric type, the current specification version defines this metric type as a monotonic counter which can only increase.

If you use the programmatic approach, you are also able to define the amount of increase for the counter on each invocation:

```
public void checkoutItem(String item, Long amount) {
    this.metricRegistry.counter(item + "Count").inc(amount);
    // further business logic
}
```

Create a metered metric

The meter type is perfect if you want to measure the throughput of something and get the one-, five- and fifteen-minute rates. As an example I'll monitor the throughput of a JAX-RS endpoint:

```
@GET
@Metered(name = "getBookCommentForLatestBookRequest",
    tags = {"spec=JAX-RS", "level=REST"})
@Produces(MediaType.TEXT_PLAIN)
public Response getBookCommentForLatestBookRequest() {
    String latestBookRequestId = bookRequestProcessor.getLatestBookRequestId();
    return Response.ok(this.bookCommentClient.getBookCommentByBookId(latestBookRequestId)
).build();
}
```

After several invocations, the result looks like the following:

```
"de.rieckpil.blog.BookResource.getBookCommentForLatestBookRequest": {
"oneMinRate;level=REST;spec=JAX-RS": 1.1363013189791909e-24,
"fiveMinRate;level=REST;spec=JAX-RS": 0.0000042408326224725166,
"meanRate;level=REST;spec=JAX-RS": 0.003936520624021342,
"fifteenMinRate;level=REST;spec=JAX-RS": 0.0055092085268208186,
"count;level=REST;spec=JAX-RS": 13
}
```

Depending on your implementation provider of MicroProfile Metrics, tracking time and invocations for JAX-RS endpoints might be redundant, as there is now the optional base metric `REST.request`.

Create a gauge metric

To monitor a value which can increase and decrease over time, you should use the gauge metric type. Imagine you want to visualize the current disk size or the remaining messages to process in a queue:

```
@Gauge(unit = "amount")
public Long remainingBookRequestsToProcess() {
    // monitor e.g. current size of a JMS queue
    return ThreadLocalRandom.current().nextLong(0, 1_000_000);
}
```

The `unit` attribute of the annotation is required and has to be explicitly configured. There is a `MetricUnits` class which you can use for common units like seconds or megabytes.

In contrast to all other metrics, the `@Gauge` annotation can only be used in combination with a single instance (e.g. `@ApplicationScoped`) as otherwise, it would be not clear which instance represents the actual value.

There is a `@ConcurrentGauge` if you need to count parallel invocations. The outcome is the current value of the gauge, which might increase or decrease over time:

```
# TYPE application_..._remainingBookRequestsToProcess_amount
application_..._remainingBookRequestsToProcess_amount 990120

// invocation of /metrics 5 minutes later

# TYPE application_..._remainingBookRequestsToProcess_amount
application_..._remainingBookRequestsToProcess_amount 11003
```

» For more hands-one experience with Eclipse MicroProfile Metrics, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

MicroProfile OpenAPI

Exposing REST endpoints usually requires documentation for your clients. This documentation usually includes the following: accepted media types, HTTP method, path variables, query parameters, and the request and response schema.

With the OpenAPI v3 specification we have a standard way to document APIs. You can generate this kind of API documentation from your `JAX-RS` classes using MicroProfile OpenAPI out-of-the-box.

In addition, you can customize the result with additional metadata like detailed description, error codes and their reasons, and further information about the used security mechanism.

Specification profile: MicroProfile OpenAPI

- Current version: **1.1** in MicroProfile **3.3**
- GitHub repository
- Latest specification document
- Basic use case: Provide a unified Java API for the OpenAPI v3 specification to expose API documentation

Customize your API documentation with MicroProfile OpenAPI

Without any additional annotation or configuration, you get your API documentation with MicroProfile OpenAPI out-of-the-box. Therefore your `JAX-RS` classes are scanned for your `@Produces`, `@Consumes`, `@Path`, `@GET` etc. annotations to extract the required information for the documentation. If you have external clients accessing your endpoints, you usually add further metadata for them to understand what each endpoint is about.

Fortunately, the MicroProfile OpenAPI specification defines a bunch of annotations you can use to customize the API documentation. The following example shows a part of the available annotation you can use to add further information:

```
@GET
@Operation(summary = "Get all books", description = "Returns all available books of the
book store XYZ")
@APIResponse(responseCode = "404", description = "No books found")
@APIResponse(responseCode = "418", description = "I'm a teapot")
@APIResponse(responseCode = "500", description = "Server unavailable")
@Tag(name = "BETA", description = "This API is currently in beta state")
@Produces(MediaType.APPLICATION_JSON)
public Response getAllBooks() {
System.out.println("Get all books...");
    return Response.ok(new Book("MicroProfile", "Duke", 1L)).build();
}
```

In this example, I'm adding a summary and description to the endpoint to tell the client what this endpoint is about. Furthermore, you can specify the different response codes this endpoint returns and give them a description if they are somehow different from the HTTP spec.

Another important part of your API documentation is the request and response body schema. With JSON as the current de-facto standard format for exchanging data, you and need to know the expected and accepted formats.

The same is true for the response as your client needs information about the contract of the API to further process the result. This can be achieved with an additional MicroProfile OpenAPI annotation:

```
@GET
@APIResponse(description = "Book",
    content = @Content(mediaType = "application/json",
    schema = @Schema(implementation = Book.class)))
@Path("/{id}")
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getBookById(@PathParam("id") Long id) {
    return Response.ok(new Book("MicroProfile", "Duke", 1L)).build();
}
```

Within the `@APIResponse` annotation we can reference the response object with the `schema` attribute. This can point to your data transfer object class.

Aforementioned Java class can then also have further annotations to specify which field is required and what are example values:

```
@Schema(name = "Book", description = "POJO that represents a book.")
public class Book {

    @Schema(required = true, example = "MicroProfile")
    private String title;

    @Schema(required = true, example = "Duke")
    private String author;

    @Schema(required = true, readOnly = true, example = "1")
    private Long id;

}
```

Access the created documentation

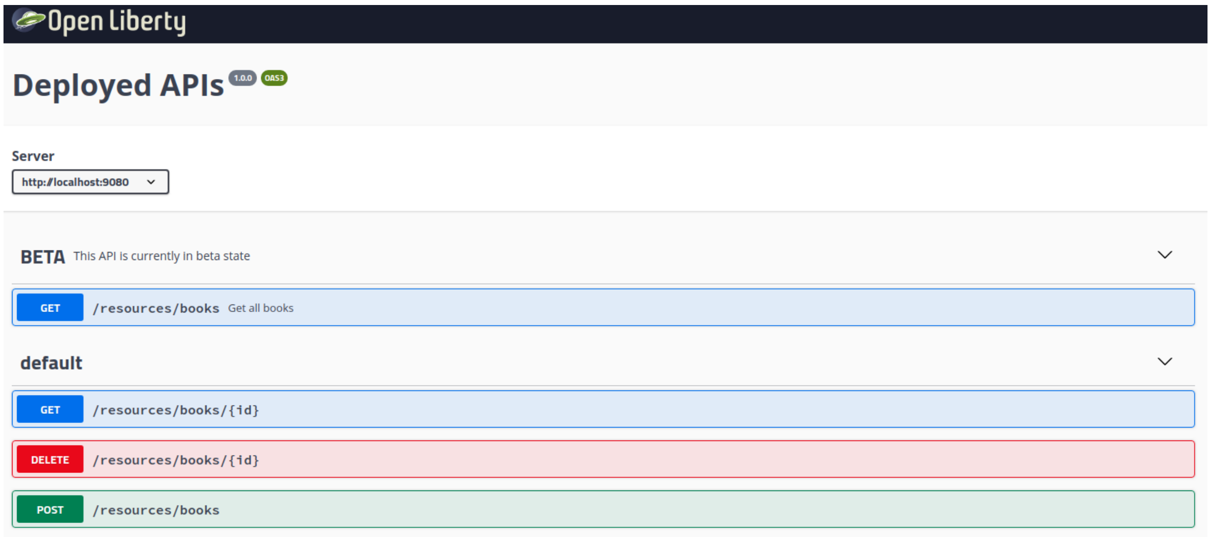
The MicroProfile OpenAPI specification defines a pre-defined endpoint to access the documentation: `/openapi`:

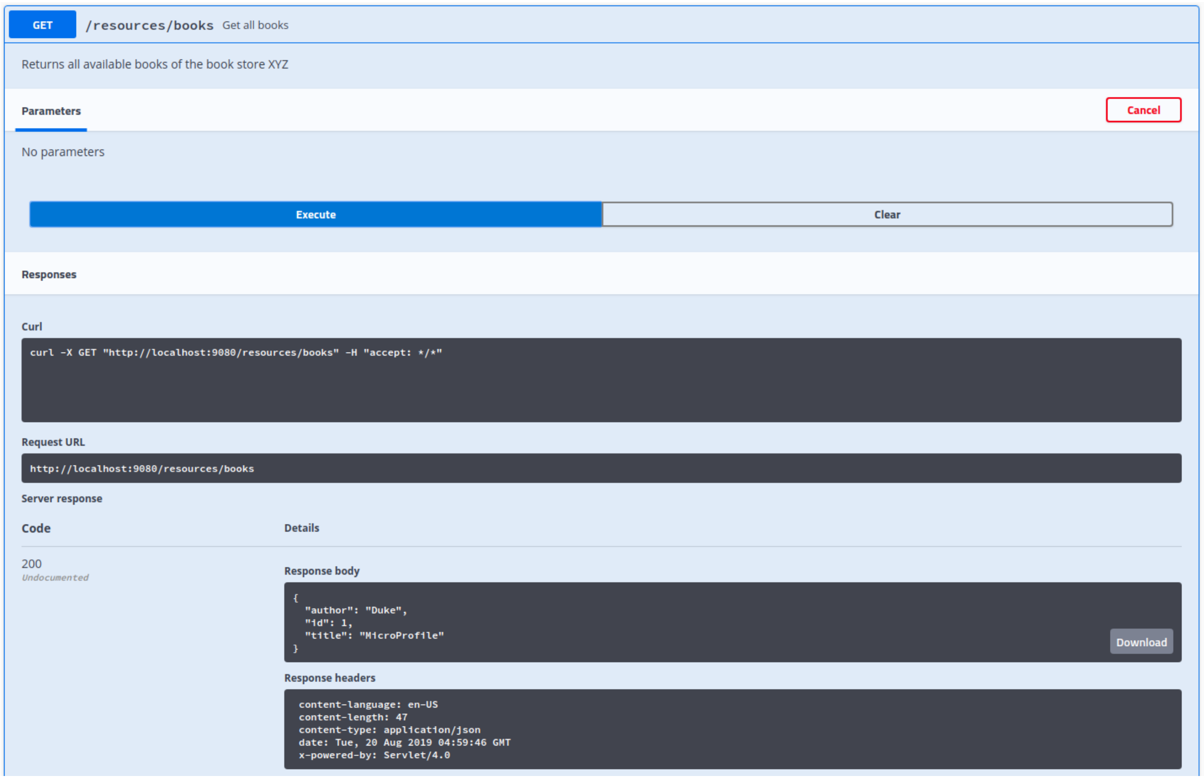
```yaml
info:
  title: Deployed APIs
  version: 1.0.0
servers:
  - url: http://localhost:9080
  - url: https://localhost:9443
tags:
  - name: BETA
    description: This API is currently in beta state
paths:
  /resources/books/{id}:
    get:
      operationId: getBookById
      parameters:
      - name: id
        in: path
        required: true
        schema:
          type: integer
          format: int64
      responses:
        default:
          description: Book
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Book'
```

This endpoint returns your generated API documentation in the OpenAPI v3 specification format as `text/plain`.

Moreover, if you are using Open Liberty you'll get a nice-looking user interface for your API documentation. You can access it at http://localhost:9080/openapi/ui/.

This looks similar to the Swagger UI and offers your client a way to explore your API and also trigger requests to your endpoints via this user interface:


Open Liberty
Deployed APIs
1.0.0
OAS3
Server
http://localhost:9080
BETA This API is currently in beta state
GET /resources/books Get all books
default
GET /resources/books/{id}
DELETE /resources/books/{id}
POST /resources/books


GET
/resources/books Get all books
Returns all available books of the book store XYZ
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl

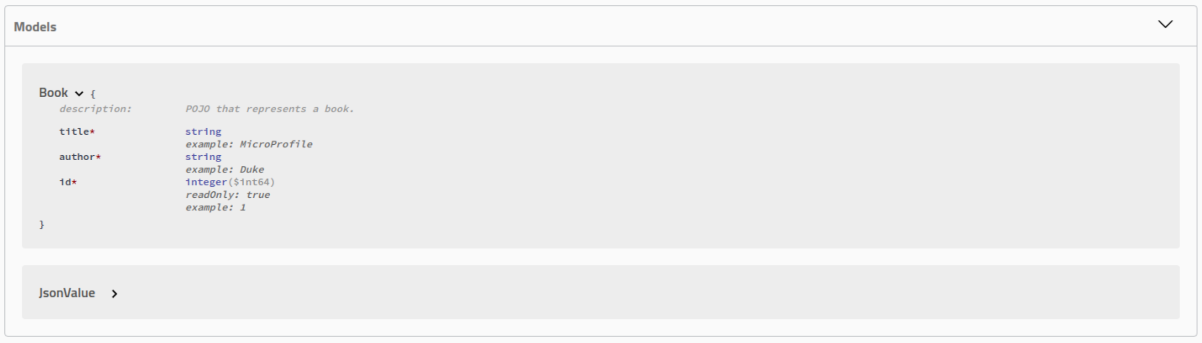
```
curl -X GET "http://localhost:9080/resources/books" -H "accept: */*"
```

Request URL

```
http://localhost:9080/resources/books
```

```
{
  "author": "Duke",
  "id": 1,
  "title": "MicroProfile"
}
```

```
content-language: en-US
content-length: 47
content-type: application/json
date: Tue, 20 Aug 2019 04:59:46 GMT
x-powered-by: Servlet/4.0
```



```
Book {
    description:        POJO that represents a book.
    title*              string
                        example: MicroProfile
    author*             string
                        example: Duke
    id*                 integer($int64)
                        readOnly: true
                        example: 1
}
```

» For more hands-one experience with Eclipse MicroProfile OpenAPI, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

MicroProfile OpenTracing

Tracing method calls in a monolith to identify slow parts is simple. Everything is happening in one application (context) and you can easily add metrics to gather information about e.g. the elapsed time for fetching data from the database.

Once you have a microservice environment with service-to-service communication, tracing needs more effort. If a business operation requires your service to call other services (which might then also call others) to gather data, identifying the source of a bottleneck is hard. Over the past years, several vendors evolved to tackle this issue of distributed tracing (e.g. Jaeger, Zipkin etc.).

As the different solutions did not rely on a single, standard mechanism for trace description and propagation, a vendor-neutral standard for distributed tracing was due: OpenTracing. With Eclipse MicroProfile we get a dedicated specification to make use of this standard: MicroProfile OpenTracing.

Specification profile: MicroProfile OpenTracing

- Current version: **1.3** in MicroProfile **3.3**
- GitHub repository
- Latest specification document
- Basic use case: Provide distributed tracing for your JAX-RS application using the OpenTracing standard

Basics about distributed tracing

Once the flow of a request touches multiple service boundaries, you need to somehow correlate each incoming call with the same business flow. To accomplish this with distributed tracing, each service is instrumented to log messages with a correlation id that may have been propagated from an upstream service.

These messages are then collected in a storage system and aggregated as they share the same correlation id. A so-called trace represents the full journey of a request containing multiple spans. A span contains a single operation within the request with both start and end-time information.

The distributed tracing systems (e.g. Jager or Zipkin) then usually provide a visual timeline representation for a given trace with its spans.

Enabling distributed tracing with MicroProfile OpenTracing

The MicroProfile OpenTracing specification does not address the problem of defining, implementing or configuring the underlying distributed tracing system. It assumes an environment where all services use a common OpenTracing implementation.

The MicroProfile specification defines two operation modes:

- Without instrumentation of application code (distributed tracing is enabled for JAX-RS applications by default)
- With explicit code instrumentation (using the `@Traced` annotation)

So once a request arrives at a JAX-RS endpoint, the Tracer instance extracts the SpanContext (if given) from the inbound request and starts a new span. If there is no SpanContext yet, e.g. the request is coming from a frontend application, the MicroProfile application has to create one.

Every outgoing request (with either the JAX-RS Client or the MicroProfile Rest Client) then needs to contain the SpanContext and propagate it downstream. Tracing for the JAX-RS Client might need to be explicitly enabled (depending on the implementation), for the MicroProfile Rest Client it is globally enabled by default.

Besides the no instrumentation mode, you can add the `@Traced` annotation to a class or method to explicitly start a new span at the beginning of a method.

Sample application setup for MicroProfile OpenTracing

To provide you an example, I'm using the following two services to simulate a *microservice architecture* setup: *book-store* and *book-store-client*.

Both are MicroProfile applications and have no further dependencies. The *book-store-client* has one public endpoint to retrieve books together with their price:

```java
@Path("books")
public class BookResource {

    @Inject
    private BookProvider bookProvider;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getBooks() {
        return Response.ok(bookProvider.getBooksFromBookStore()).build();
    }

}
```

For gathering information about the book and its price, the book-store-client communicates with the *book-store*:

```
@RequestScoped
public class BookProvider {

    @Inject
    private PriceCalculator priceCalculator;

    private WebTarget bookStoreTarget;

    @PostConstruct
    public void setup() {
        Client client = ClientBuilder
                .newBuilder()
                .connectTimeout(2, TimeUnit.SECONDS)
                .readTimeout(2, TimeUnit.SECONDS)
                .build();

        this.bookStoreTarget = client.target("http://book-store:9080/resources/books");
    }

    public JsonArray getBooksFromBookStore() {

        JsonArray books = this.bookStoreTarget
                .request()
                .get()
                .readEntity(JsonArray.class);

        List<JsonObject> result = new ArrayList();

        for (JsonObject book : books.getValuesAs(JsonValue::asJsonObject)) {
            result.add(Json.createObjectBuilder()
                    .add("title", book.getString("title"))
                    .add("price", priceCalculator.getPriceForBook(book.getInt("id")))
                    .build());
        }

        return result
                .stream()
                .collect(JsonCollectors.toJsonArray());
    }
}
```

So there will be at least on outgoing call to fetch all available books and for each book and additional request to get the price of the book:

```
@RequestScoped
public class PriceCalculator {

    private WebTarget bookStorePriceTarget;
    private Double discount = 1.5;

    @PostConstruct
    public void setUp() {
        Client client = ClientBuilder
                .newBuilder()
                .connectTimeout(2, TimeUnit.SECONDS)
                .readTimeout(2, TimeUnit.SECONDS)
                .build();

        this.bookStorePriceTarget = client.target("http://book-
store:9080/resources/prices");
    }

    public Double getPriceForBook(int id) {
        Double bookPrice = this.bookStorePriceTarget
            .path(String.valueOf(id))
            .request()
            .get()
            .readEntity(Double.class);
        return Math.round((bookPrice - discount) * 100.0) / 100.0;
    }

}
```

On the *book-store* side, when fetching the prices, there is a random `Thread.sleep()`, so we can later see different traces. Without further instrumentations on both sides, we are ready for distributed tracing.

We could add additional `@Traced` annotations to the involved methods, to create a span for each method call and narrow down the tracing.

Using the Zipkin implementation on Open Liberty

For this example, I'm using Open Liberty to deploy both applications. With Open Liberty we have to add a feature for the OpenTracing implementation to the server and configure it in `server.xml`:

```
FROM open-liberty:kernel-java11
COPY --chown=1001:0  target/microprofile-open-tracing-server.war /config/dropins/
COPY --chown=1001:0  server.xml /config/
COPY --chown=1001:0  extension /opt/ol/wlp/usr/extension
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<server description="new server">

    <featureManager>
        <feature>microProfile-3.3</feature>
        <feature>usr:opentracingZipkin-0.31</feature>
    </featureManager>

    <opentracingZipkin host="zipkin" port="9411"/>

    <mpMetrics authentication="false"/>

    <ssl id="defaultSSLConfig" keyStoreRef="defaultKeyStore" trustStoreRef="jdkTrustStore
"/>
    <keyStore id="jdkTrustStore" location="${java.home}/lib/security/cacerts" password=
"changeit"/>

    <httpEndpoint id="defaultHttpEndpoint" httpPort="9080" httpsPort="9443" />
</server>
```

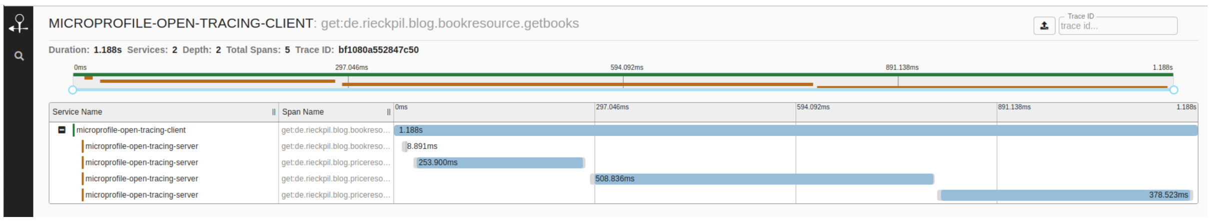
The OpenTracing Zipkin implementation is provided by IBM and can be downloaded at the following tutorial.

For the *book-store* DNS resolution, you saw in the previous code snippets and to start Zipkin as the distributed tracing system, I'm using `docker-compose`:

```yaml
services:
  book-store-client:
    build: book-store-client/
    ports:
      - "9080:9080"
      - "9443:9443"
    links:
      - zipkin
      - book-store
  book-store:
    build: book-store/
    links:
      - zipkin
  zipkin:
    image: openzipkin/zipkin
    ports:
      - "9411:9411"
```

Once both services and Zipkin is running, you can visit http://localhost:9080/resources/books to fetch all available books from the *book-store-client* application.

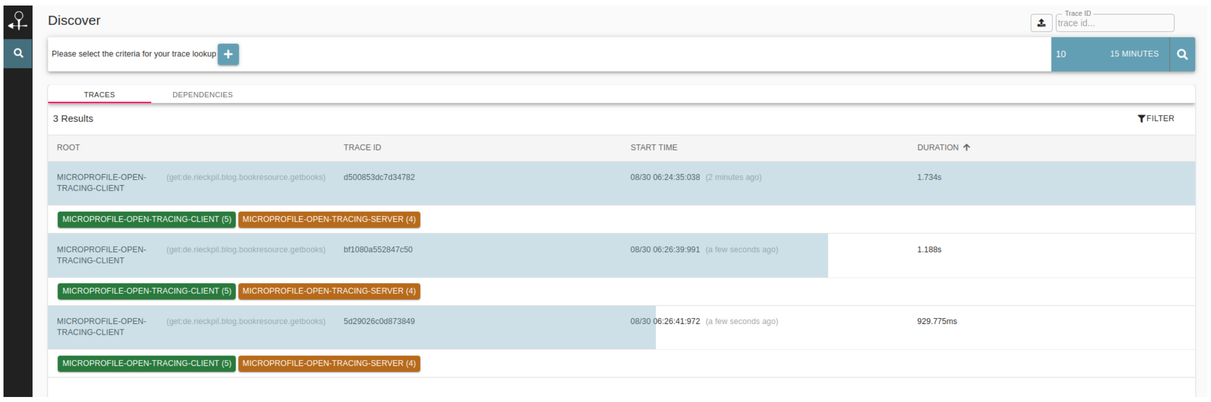
You can now hit this endpoint several times and then switch to http://localhost:9411/zipkin/ and query for all available traces:


MICROPROFILE-OPEN-TRACING-CLIENT: get:de.rieckpil.blog.bookresource.getbooks

Duration: **1.188s** Services: **2** Depth: **2** Total Spans: **5** Trace ID: **bf1080a552847c50**

Service Name	Span Name	Duration
microprofile-open-tracing-client	get:de.rieckpil.blog.bookreso...	1.188s
microprofile-open-tracing-server	get:de.rieckpil.blog.bookreso...	8.891ms
microprofile-open-tracing-server	get:de.rieckpil.blog.pricereso...	253.900ms
microprofile-open-tracing-server	get:de.rieckpil.blog.pricereso...	508.836ms
microprofile-open-tracing-server	get:de.rieckpil.blog.pricereso...	378.523ms

Once you click on a specific trace, you'll get a timeline to see what operation took the most time:


Discover

Trace ID
trace id...

Please select the criteria for your trace lookup

10 15 MINUTES

TRACES DEPENDENCIES

3 Results FILTER

ROOT		TRACE ID	START TIME	DURATION
MICROPROFILE-OPEN-TRACING-CLIENT	(get:de.rieckpil.blog.bookresource.getbooks)	d500853dc7d34782	08/30 06:24:35:038 (2 minutes ago)	1.734s
MICROPROFILE-OPEN-TRACING-CLIENT (5)	MICROPROFILE-OPEN-TRACING-SERVER (4)			
MICROPROFILE-OPEN-TRACING-CLIENT	(get:de.rieckpil.blog.bookresource.getbooks)	bf1080a552847c50	08/30 06:26:39:991 (a few seconds ago)	1.188s
MICROPROFILE-OPEN-TRACING-CLIENT (5)	MICROPROFILE-OPEN-TRACING-SERVER (4)			
MICROPROFILE-OPEN-TRACING-CLIENT	(get:de.rieckpil.blog.bookresource.getbooks)	5d29026c0d873849	08/30 06:26:41:972 (a few seconds ago)	929.775ms
MICROPROFILE-OPEN-TRACING-CLIENT (5)	MICROPROFILE-OPEN-TRACING-SERVER (4)			

» For more hands-one experience with Eclipse MicroProfile Open Tracing, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

MicroProfile Fault Tolerance

With the current trend to build distributed-systems, it is increasingly important to build fault-tolerant services. Fault tolerance is about using different strategies to handle failures in a distributed system.

Moreover, the services should be resilient and be able to operate further if a failure occurs in an external service and not cascade the failure and bring the system down. There is a set of common patterns to achieve fault tolerance within your system. These patterns are all available within the MicroProfile Fault Tolerance specification.

This example covers all available interceptor bindings as defined in the specification:

- Fallback
- Timeout
- Retry
- CircuitBreaker
- Asynchronous
- Bulkhead

Specification profile: MicroProfile Fault Tolerance

- Current version: **2.1** in MicroProfile **3.3**
- GitHub repository
- Latest specification document
- Basic use case: Provide a set of strategies to build resilient and fault-tolerant services

Provide a fallback method

First, let’s cover the `@Fallback` interceptor binding of the MicroProfile Fault Tolerance specification. With this annotation, you can provide a fallback behavior of your method in case of an exception.

Assume your service fetches data from other microservices and the call might fail due to network issues or downtime of the target. In case your service could recover from the failure and you can provide meaningful fallback behavior for your domain, the `@Fallback` annotation saves you.

A good example might be the checkout process of your webshop where you rely on a third-party service for handling e.g. credit card payments. If this service fails, you might fall back to a default payment provider and recover gracefully from the failure.

For a simple example, I’ll demonstrate it with a JAX-RS client request to a placeholder REST API and provide a fallback method:

```
@Fallback(fallbackMethod = "getDefaultPost")
public JsonObject getPostById(Long id) {
    return this.webTarget
            .path(String.valueOf(id))
            .request()
            .accept(MediaType.APPLICATION_JSON)
            .get(JsonObject.class);
}

public JsonObject getDefaultPost(Long id) {
    return Json.createObjectBuilder()
            .add("comment", "Lorem ipsum")
            .add("postId", id)
            .build();
}
```

With the `@Fallback` annotation you can specify the method name of the fallback method which must share the same response type and method arguments as the annotated method. In addition, you can also specify a dedicated class to handle the fallback.

This class is required to implement the `FallbackHandler<T>` interface where `T` is the response type of the targeted method:

```
@Fallback(PlaceHolderApiFallback.class)
public JsonObject getPostById(Long id) {
    return this.webTarget
        .path(String.valueOf(id))
        .request()
        .accept(MediaType.APPLICATION_JSON)
        .get(JsonObject.class);
}
```

```
public class PlaceHolderApiFallback implements FallbackHandler<JsonObject> {

    @Override
    public JsonObject handle(ExecutionContext context) {
        return Json.createObjectBuilder()
                .add("comment", "Lorem ipsum")
                .add("postId", Long.valueOf(context.getParameters()[0].toString()))
                .build();
    }
}
```

As you'll see it in the upcoming chapters, the `@Fallback` annotation can be used in combination with other MicroProfile Fault Tolerance interceptor bindings.

Furthermore, you can instruct the fallback annotation to apply only when a specific exception is thrown. You can also include different exceptions from not triggering the fallback behaviour:

```
@Fallback(value = PlaceHolderApiFallback.class,
          applyOn = {MyCustomException.class, MySevereException.class},
          skipOn = NumberFormatException.class)
```

By default the fallback occurs on every exception extending `Throwable` and does not skip on any exception.

Add timeouts to limit the duration of a method execution

For some operations in your system, you might have a strict response time target. If you make use of the JAX-RS client or the client of MicroProfile Rest Client you can specify read and connect timeouts to avoid long-running requests.

But what about use cases where you can't declare timeouts easily? The MicroProfile Fault Tolerance specification defines the `@Timeout` annotation for such problems. With this interceptor binding, you can specify the maximum duration of a method. If the computation time within the method exceeds the limit, a `TimeoutException` is thrown.

```
@Timeout(4000)
@Fallback(fallbackMethod = "getFallbackData")
public String getDataFromLongRunningTask() throws InterruptedException {
    Thread.sleep(4500);
    return "duke";
}
```

The default unit is milliseconds, but you can configure a different `ChronoUnit`:

```
@Timeout(value = 4, unit = ChronoUnit.SECONDS)
@Fallback(fallbackMethod = "getFallbackData")
public String getDataFromLongRunningTask() throws InterruptedException {
    Thread.sleep(4500);
    return "duke";
}
```

Define retry policies for method calls

A valid fallback behavior for an external system call might be just to retry it. With the `@Retry` annotation, we can achieve such a behavior. Directly retrying to execute the request might not always be the best solution.

Similarily you want to add delay for the next retry and maybe add some randomness. We can configure such a requirement with the `@Retry` annotation:

```
@Retry(maxDuration = 5000, maxRetries = 3, delay = 500, jitter = 200)
@Fallback(fallbackMethod = "getFallbackData")
public String accessFlakyService() {

    System.out.println("Trying to access flaky service at " + LocalTime.now());

    if (ThreadLocalRandom.current().nextLong(1000) < 50) {
        return "flaky duke";
    } else {
        throw new RuntimeException("Flaky service not accessible");
    }
}
```

In this example, we would try to execute the method three times with a delay of 500 milliseconds and 200 milliseconds of randomness (called `jitter`).

The effective delay is the following: [delay - jitter, delay + jitter] (in our example 300 to 700 milliseconds).

Furthermore, endless retrying might also be counter-productive. That's why we can specify the `maxDuration` which is quite similar to the `@Timeout` annotation above. If the whole retrying takes more than 5 seconds, it will fail with a `TimeoutException`.

Similar to the `@Fallback` annotation, we can specify the type of exceptions to trigger and not trigger a retry:

```
@Retry(maxRetries = 3,
       retryOn = {RuntimeException.class},
       abortOn = {NumberFormatException.class})
```

Add a Circuit Breaker to fail fast

Once an external system you call is down or returning 503 as it is currently unavailable to process further requests, you might not want to access it for a given timeframe again.

This might help the other system to recover and your methods can fail fast as you already know the expected response from requests in the past. For this scenario, the Circuit Breaker pattern comes into place.

The Circuit Breaker offers a way to fail fast by directly failing the method execution to prevent further overloading of the target system and indefinite wait or timeouts. With MicroProfile Fault Tolerance we have an annotation to achieve this with ease: `@CircuitBreaker`

There are three different states a Circuit Breaker can have: closed, opened, half-open.

In the closed state, the operation is executed as expected. If a failure occurs while e.g. calling an external service, the Circuit Breaker records such an event. If a particular threshold of failures is met, it will switch to the open state. Once the Circuit Breaker enters the open state, further calls will fail immediately.

After a given delay the circuit enters the half-open state. Within the half-open state, trial executions will happen. Once such a trial execution fails, the circuit transitions to the open state again.

When a predefined number of these trial executions succeed, the circuit enters the original closed state. Let's have a look at the following example:

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 5, failureRatio = 0.5,
delay = 500)
@Fallback(fallbackMethod = "getFallbackData")
public String getRandomData() {
  if (ThreadLocalRandom.current().nextLong(1000) < 300) {
    return "random duke";
  } else {
    throw new RuntimeException("Random data not available");
  }
}
```

In the example above I define a Circuit Breaker which enters the open state once 50% (`failureRatio=0.5`) of five consecutive executions (`requestVolumeThreshold=5`) fail. After a delay of 500 milliseconds in the open state, the circuit transitions to half-open. Once ten trial executions (`successThreshold=10`) in the half-open state succeed, the circuit will be back in the closed state.

This annotation also allows defining the exception types to skip and to fail on:

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 5, delay = 500,
    skipOn = {NumberFormatException.class},
    failOn = {RuntimeException.class})
```

Execute a method asynchronously

Some use cases of your system might not require synchronous and in-order execution of different tasks. For instance, you can fetch data for a customer (purchased orders, contact information, invoices) from different services in parallel.

The MicroProfile Fault Tolerance specification offers a convenient way for achieving such asynchronous method executions: `@Asynchronous`:

```
@Asynchronous
public Future<String> getConcurrentServiceData(String name) {
    System.out.println(name + " is accessing the concurrent service");
    return CompletableFuture.completedFuture("concurrent duke");
}
```

With this annotation, the execution will be on a separate thread and the method has to return either a `Future` or a `CompletionStage`

Apply Bulkheads to limit the number of concurrent calls

The Bulkhead pattern is a way of isolating failures in your system while the rest can still function.

It’s named after the sectioned parts (bulkheads) of a ship. If one bulkhead of a ship is damaged and filled with water, the other bulkheads aren’t affected, which prevents the ship from sinking.

Imagine a scenario where all your threads are occupied for a request to a (slowresponding) external system and your application can't process other tasks. To prevent such a scenario, we can apply the `@Bulkhead` annotation and limit concurrent calls:

```
@Bulkhead(5)
@Asynchronous
public Future<String> getConcurrentServiceData(String name) throws InterruptedException {
    Thread.sleep(1000);
    System.out.println(name + " is accessing the concurrent service");
    return CompletableFuture.completedFuture("concurrent duke");
}
```

In this example, only five concurrent calls can enter this method and further have to wait. If this annotation is used together with `@Asynchronous`, as in the example above, it means thread isolation.

In addition and only for asynchronous methods we can specify the length of the waiting queue with the attribute `waitingTaksQueue`.

For non-async methods, the specification defines to utilize semaphores for isolation.

MicroProfile Fault Tolerance integration with MicroProfile Config

Above all, the MicroProfile Fault Tolerance specification provides tight integration with the config spec.

You can configure every attribute of the different interceptor bindings with an external config source like the `microprofile-config.properties` file. The pattern for external configuration is the following:

```
`<classname>/<methodname>/<annotation>/<parameter>`
```

```
de.rieckpil.blog.RandomDataProvider/accessFlakyService/Retry/maxRetries=10
de.rieckpil.blog.RandomDataProvider/accessFlakyService/Retry/delay=300
de.rieckpil.blog.RandomDataProvider/accessFlakyService/Retry/maxDuration=5000
```

» For more hands-one experience with Eclipse MicroProfile Fault Tolerance, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

MicroProfile Rest Client

In a distributed system your services usually communicate via HTTP and expose REST APIs. External clients or other services in your system consume these endpoints on a regular basis to e.g. fetch data from a different part of the domain. If you are using Java EE you can utilize the JAX-RS `WebTarget` and `Client` for this kind of communication.

With the MicroProfile Rest Client specification, you'll get a more advanced and simpler way of creating these RESTful clients. You just declare interfaces and use a more declarative approach (like you might already know it from the Feign library).

Specification profile: MicroProfile Rest Client

- Current version: **1.4** in MicroProfile **3.3**
- GitHub repository
- Latest specification document
- Basic use case: Provide a type-safe approach to invoke RESTful services over HTTP.

Defining the RESTful client

For defining the Rest Client you just need a Java interface and model the remote REST API using JAX-RS annotations:

```
public interface JSONPlaceholderClient {

    @GET
    @Path("/posts")
    JsonArray getAllPosts();

    @POST
    @Path("/posts")
    Response createPost(JsonObject post);

}
```

You can specify the response type with a specific POJO (JSON-B will then try to deserialize the HTTP response body) or use the generic `Response` class of JAX-RS.

Furthermore, you can indicate an asynchronous execution, if you use `CompletionStage<T>` as the method return type:

```
@GET
@Path("/posts/{id}")
CompletionStage<JsonObject> getPostById(@PathParam("id") String id);
```

Path variables and query parameters for the remote endpoint can be specified with `@PathParam` and `@QueryParam`:

```
@GET
@Path("/posts")
JsonArray getAllPosts(@QueryParam("orderBy") String orderDirection);

@GET
@Path("/posts/{id}/comments")
JsonArray getCommentsForPostByPostId(@PathParam("id") String id);
```

You can define the media type of the request and the expected media type of the response on either interface level or for each method separately:

```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public interface JSONPlaceholderClient {

    @GET
    // overrides the JSON media type only for this method
    @Produces(MediaType.APPLICATION_XML)
    @Path("/posts/{id}")
    CompletionStage<JsonObject> getPostById(@PathParam("id") String id);

}
```

If you have to declare specific HTTP headers (e.g. for authentication), you can pass them either to the method with `@HeaderParam` or define them with `@ClientHeaderParam` (static value or refer to a method):

```
@ClientHeaderParam(name = "X-Application-Name", value = "MP-blog")
public interface JSONPlaceholderClient {

    @PUT
    @ClientHeaderParam(name = "Authorization", value = "{generateAuthHeader}")
    @Path("/posts/{id}")
    Response updatePostById(@PathParam("id") String id, JsonObject post,
                            @HeaderParam("X-Request-Id") String requestIdHeader);

    default String generateAuthHeader() {
        return "Basic "+ new String(Base64
            .getEncoder().encode("duke:SECRET".getBytes()));
    }

}
```

Specifying multiple HTTP headers

If you want to generate multiple HTTP headers or **propagate** HTTP headers from an incoming JAX-RS request (e.g. pass the Authorization header to a downstream system), you can use the `ClientHeadersFactory`.

This interface specifies one method that returns the final HTTP headers for an outgoing client call. The headers might still be manipulated by a filter or any other mechanism before sending the client request.

While implementing this method you get two arguments passed to the `update` method. First, yo get passed incoming headers if the Rest Client is used as part of a JAX-RS request. These might be empty. Furthermore, you have access to the HTTP headers you specified already at your interface while using e.g.

`@ClientHeaderParam`:

```
public interface ClientHeadersFactory {
    MultivaluedMap<String, String> update(
         MultivaluedMap<String, String> incomingHeaders,
         MultivaluedMap<String, String> clientOutgoingHeaders);
}
```

Inside your implementation, you can now define logic for the outgoing HTTP headers. As an example I'm merging the incoming headers with the client outgoing headers and add three more headers manually:

```
@ApplicationScoped
public class GlobalClientHeaders implements ClientHeadersFactory {

  @Inject
  @ConfigProperty(name = "secrets.value")
  private String secretValue;

  @Override
  public MultivaluedMap<String, String> update(
    MultivaluedMap<String, String> incomingHeaders,
    MultivaluedMap<String, String> clientOutgoingHeaders) {

    System.out.println("--- Incoming headers of the JAX-RS environment");
    incomingHeaders.forEach((k, v) -> System.out.println(k + ":" + v));

    System.out.println("--- Specified outgoing headers of the Rest Client");
    clientOutgoingHeaders.forEach((k, v) -> System.out.println(k + ":" + v));

    MultivaluedMap<String, String> resultHeader = new MultivaluedHashMap();
    resultHeader.putAll(incomingHeaders);
    resultHeader.putAll(clientOutgoingHeaders);

    resultHeader.add("X-Secret-Header", secretValue);
    resultHeader.add("X-Global-Header", "duke");
    resultHeader.add("X-Special-Header", "MicroProfile");

    System.out.println("--- Header of the Rest Client after merging");
    resultHeader.forEach((k, v) -> System.out.println(k + ":" + v));

    return resultHeader;
  }
}
```

Besides the benefit of propagating HTTP headers of a JAX-RS request, you can use `@Inject` here if your implementation is managed by CDI. This allows you to inject secrets for example or any other CDI bean to calculate a header value.

Finally, you have to register your factory implementation using `@RegisterClientHeaders(NameOfFactoryImpl.class)` on your Rest Client interface.

```
@RegisterRestClient
@RegisterClientHeaders(GlobalClientHeaders.class)
public interface JSONPlaceholderClient {
}
```

You can get further information on using the `ClientHeadersFactory` interface in the MicroProfile Rest Client 1.4 update video.

Using the client interface

Once you define your Rest Client interface you have two ways of using them. First, you can make use of the programmatic approach using the `RestClientBuilder`.

With this builder we can set the base URI, define timeouts and register JAX-RS features/provider like `ClientResponseFilter`, `MessageBodyReader`, `ReaderInterceptor` etc. :

```
JSONPlaceholderClient jsonApiClient = RestClientBuilder.newBuilder()
    .baseUri(new URI("https://jsonplaceholder.typicode.com"))
    .register(ResponseLoggingFilter.class)
    .connectTimeout(2, TimeUnit.SECONDS),
    .readTimeout(2, TimeUnit.SECONDS)
    .build(JSONPlaceholderClient.class);

jsonApiClient.getPostById("1").thenAccept(System.out::println);
```

In addition to this, we can use CDI to inject the Rest Client. To register the interface as a CDI managed bean during runtime, the interface requires the `@RegisterRestClient` annotation:

```
@RegisterRestClient
@RegisterProvider(ResponseLoggingFilter.class)
public interface JSONPlaceholderClient {

}
```

With the `@RegisterProvider` you can register further JAX-RS provider and features as you've seen it in the programmatic approach. If you don't specify any scope for the interface, the `@Dependent` scope will be used by default. With this scope, your Rest Client bean is bound (dependent) to the lifecycle of the injector class.

You can now use it as any other CDI bean and inject it to your classes. Make sure to add the CDI qualifier `@RestClient` to the injection point:

```
@ApplicationScoped
public class PostService {

    @Inject
    @RestClient
    JSONPlaceholderClient jsonPlaceholderClient;

}
```

Further configuration for the Rest Client

If you use the CDI approach, you can make use of MicroProfile Config to further configure the Rest Client. You can specify the following properties with MicroProfile Config:

- Base URL (…/`mp-rest/url`)
- Base URI (…/`mp-rest/uri`)
- The CDI scope of the client as a fully qualified class name (…/`mp-rest/scope`)
- JAX-RS provider as a comma-separated list of fully qualified class names (`../mp-rest/providers`)
- The priority of a registered provider (…/`mp-rest/providers/com.acme.MyProvider/priority`)
- Connect and read timeouts (…/`mp-rest/connectTimeout` and …/`mp-rest/readTimeout`)

You can specify these properties for each client individually as you have to specify the fully qualified class name of the Rest Client for each property:

```
de.rieckpil.blog.JSONPlaceholderClient/mp-rest/url=https://jsonplaceholder.typicode.com
de.rieckpil.blog.JSONPlaceholderClient/mp-rest/connectTimeout=3000
de.rieckpil.blog.JSONPlaceholderClient/mp-rest/readTimeout=3000
```

» For more hands-one experience with Eclipse MicroProfile Rest Client, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

MicroProfile Health

Once your application is deployed to production you want to ensure it's up- and running . To determine the health and status of your application you can use monitoring based on different metrics, but this requires further knowledge and takes time.

Usually, you just want a quick answer to the question: Is my application up? The same is true if your application is running e.g. in a Kubernetes cluster, where the cluster regularly performs health probes to terminate unhealthy pods.

With MicroProfile Health you can write both readiness and liveness checks and expose them via an HTTP endpoint with ease.

Specification profile: MicroProfile Health

- Current version: **2.2** in MicroProfile **3.3**
- GitHub repository
- Latest specification document
- Basic use case: Add liveness and readiness checks to determine the application's health

Determine the application's health with MicroProfile Health

With MicroProfile Health you get three new endpoints to determine both the readiness and liveness of your application:

- `/health/ready`: Returns the result of all readiness checks and determines whether or not your application can process requests
- `/health/live`: Returns the result of all liveness checks and determines whether or not your application is up- and running
- `/health` : As in previous versions of MicroProfile Health there was no distinction between readiness and liveness, this is active for downwards compatibility. This endpoint returns the result of both health check types.

To determine your readiness and liveness you can have multiple checks. The overall status is constructed with a logical AND of all your checks of that specific type (liveness or readiness). If e.g. on liveness check fails, the overall liveness status is DOWN and the HTTP status is 503:

```
$ curl -v http://localhost:9080/health/live

< HTTP/1.1 503 Service Unavailable
< X-Powered-By: Servlet/4.0
< Content-Type: application/json; charset=UTF-8
< Content-Language: en-US

{"checks":[...],"status":"DOWN"}
```

In case of an overall UP status, you'll receive the HTTP status 200:

```
$ curl -v http://localhost:9080/health/ready

< HTTP/1.1 200 OK
< X-Powered-By: Servlet/4.0
< Content-Type: application/json; charset=UTF-8
< Content-Language: en-US

{"checks":[...],"status":"UP"}
```

Create a readiness check

To create a readiness check you have to implement the `HealthCheck` interface and add `@Readiness` to your class:

```
@Readiness
public class ReadinessCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.builder()
                .name("readiness")
                .up()
                .build();
    }
}
```

As you can add multiple checks, you need to give every check a dedicated name. In general, all your readiness checks should determine whether your application is ready to accept traffic or not.

Therefore a quick response is preferable. If your application is about exposing and accepting data using REST endpoints and does not rely on other services to work, the readiness check above should be good enough as it returns 200 once the JAX-RS runtime is up- and running:

```
{
   "checks":[
      {
         "data":{

         },
         "name":"readiness",
         "status":"UP"
      }
   ],
   "status":"UP"
}
```

Furthermore, once `/health/ready` returns 200, the readiness is identified and from now on the `/health/live` is used and no more readiness checks are required.

Create liveness checks

Creating liveness checks is as simple as creating readiness checks. The only difference is the `@Livness` annotation at class level:

```
@Liveness
public class DiskSizeCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {

        File file = new File("/");
        long freeSpace = file.getFreeSpace() / 1024 / 1024;

        return responseBuilder = HealthCheckResponse.builder()
                .name("disk")
                .withData("remainingSpace", freeSpace)
                .state(freeSpace > 100)
                .build();

}
```

In this example, I'm checking for free disk space as a service might rely on storage to persist e.g. files.

With the `.withData()` method of the `HealthCheckResponseBuilder` you can add further metadata to your response. In addition, you can also combine the `@Readiness` and `@Liveness` annotation and reuse a health check class for both checks:

```
@Readiness
@Liveness
public class MultipleHealthCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse
                .builder()
                .name("generalCheck")
                .withData("foo", "bar")
                .withData("uptime", 42)
                .withData("isReady", true)
                .up()
                .build();
    }
}
```

This check now appears for `/health/ready` and `/health/live`:

```
{
   "checks":[
      {
         "data":{
            "remainingSpace":447522
         },
         "name":"disk",
         "status":"UP"
      },
      {
         "data":{

         },
         "name":"liveness",
         "status":"UP"
      },
      {
         "data":{
            "foo":"bar",
            "isReady":true,
            "uptime":42
         },
         "name":"generalCheck",
         "status":"UP"
      }
   ],
   "status":"UP"
}
```

Other possible liveness checks might be: checking for active JDBC connections, connections to queues, CPU usage, or custom metrics (with the help of MicroProfile Metrics).

If you want to see how to utilize this specification to deploy an Eclipse MicroProfile/Jakarta EE application to Kubernetes, consider watching the following video.

» For more hands-one experience with Eclipse MicroProfile Health, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

MicroProfile JWT Auth

In today's microservice architectures security is usually based on the following protocols: OAuth2, OpenID Connect, and SAML. These main security protocols use security tokens to propagate the security state from client to server. This stateless approach is usually achieved by passing a JWT token alongside every client request.

For convenient use of this kind of token-based authentication, the MicroProfile JWT Auth evolved. The specification ensures, that the security token is extracted from the request, validated and a security context is created out of the extracted information.

Specification profile: MicroProfile JWT Auth

- Current version: **1.1** in MicroProfile **3.3**
- GitHub repository
- Latest specification document
- Basic use case: Provide JWT token-based authentication for your application

Securing a JAX-RS application

First, we have to instruct our JAX-RS application, that we'll use the JWTs for authentication and authorization. You can configure this with the `@LoginConfig` annotation:

```java
@ApplicationPath("resources")
@LoginConfig(authMethod = "MP-JWT")
public class JAXRSConfiguration extends Application {
}
```

Once an incoming request has a valid JWT within the HTTP Bearer header, the groups in the JWT are mapped to roles.

We can now limit the access for a resource to specific roles and achieve authorization with the Common Security Annotations (JSR-250) (`@RolesAllowed`, `@PermitAll`, `@DenyAll`):

```java
@GET
@RolesAllowed("admin")
public Response getBook() {

    JsonObject secretBook = Json.createObjectBuilder()
        .add("title", "secret")
        .add("author", "duke")
        .build();

    return Response.ok(secretBook).build();
}
```

Furthermore, we can inject the actual JWT token (alongside the `Principal`) with CDI and inject any claim of the JWT in addition:

```
@Path("books")
@RequestScoped
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {

    @Inject
    private Principal principal;

    @Inject
    private JsonWebToken jsonWebToken;

    @Inject
    @Claim("administrator_id")
    private JsonNumber administrator_id;

    @GET
    @RolesAllowed("admin")
    public Response getBook() {

        System.out.println("Secret book for " + principal.getName()
                + " with roles " + jsonWebToken.getGroups());
        System.out.println("Administrator level: "
                + jsonWebToken.getClaim("administrator_level").toString());
        System.out.println("Administrator id: " + administrator_id);

        JsonObject secretBook = Json.createObjectBuilder()
                .add("title", "secret")
                .add("author", "duke")
                .build();

        return Response.ok(secretBook).build();
    }

}
```

In this example, I'm injecting the claim `administrator_id` and access the claim `administrator_level` via the JWT token. These are not part of the standard JWT claims but you can add any additional metadata in your token.

Always make sure to only inject the JWT token and the claims to `@RequestScoped` CDI beans, as you'll get a `DeploymentExcpetion` otherwise:

```
javax.enterprise.inject.spi.DeploymentException: CWWKS5603E: The claim cannot be injected
into the [BackedAnnotatedField] @Inject @Claim private
de.rieckpil.blog.BookResource.administrator_id injection point for the ApplicationScoped
or SessionScoped scopes.
at
com.ibm.ws.security.mp.jwt.cdi.JwtCDIExtension.processInjectionTarget(JwtCDIExtension.java
:92)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
```

HINT: Depending on the application server you'll deploy this example, you might have to first declare the available roles with `@DeclareRoles({"admin", "chief", "duke"})`.

Required configuration for MicroProfile JWT Auth

Achieving validation of the JWT signature requires the public key. Since MicroProfile JWT Auth 1.1, we can configure this with MicroProfile Config (previously it was vendor-specific). The JWT Auth specification allows the following public key formats:

- PKCS#8 (Public Key Cryptography Standards #8 PEM)
- JWK (JSON Web Key)
- JWKS (JSON Web Key Set)
- JWK Base64 URL encoded
- JWKS Base64 URL encoded

For this example, I'm using the PKCS#8 format and specify the path of the `.pem` file containing the public key in the `microprofile-config.properties` file:

```
mp.jwt.verify.publickey.location=/META-INF/publicKey.pem
mp.jwt.verify.issuer=rieckpil
```

The configuration of the `issuer` is also required and has to match the `iss` claim in the JWT. A valid `publicKey.pem` file might look like the following:

```
-----BEGIN RSA PUBLIC KEY-----
YOUR_PUBLIC_KEY
-----END RSA PUBLIC KEY-----
```

Using JWTEnizer to create tokens for testing

Usually, the JWT is issued by an identity provider (e.g. Keycloak). For quick testing, we can use the JWTenizer tool from Adam Bien.

This provides a simple way to create valid JWT token and generates the corresponding public and private key. Once you downloaded the `jwtenizer.jar` you can run it for the first time with the following command:

```
java -jar jwtenizer.jar
```

This will now create a `jwt-token.json` file in the folder you executed the command above. We can adjust this `.json` file to our needs and model a sample JWT token:

```
{
   "iss":"rieckpil",
   "jti":"42",
   "sub":"duke",
   "upn":"duke",
   "groups":[
      "chief",
      "hacker",
      "admin"
   ],
   "administrator_id":42,
   "administrator_level":"HIGH"
}
```

Once you adjusted the raw `jwt-token.json`, you can run `java -jar jwtenizer.jar` again and this second run will now pick the existing `.json` file for creating the JWT.

Alongside the JWT token, the tool generates a `microprofile-config.properties` file, from which we can copy the public key and paste it to our `publicKey.pem` file. Furthermore the shell output of running `jwtenizer.jar` contains a cURL command we can use to hit our resources:

```
curl -i -H'Authorization: Bearer GENERATED_JWT' http://localhost:9080/resources/books
```

With a valid Bearer header you should get the following response from the backend:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/4.0
Content-Type: application/json
Date: Fri, 06 Sep 2019 03:24:16 GMT
Content-Language: en-US
Content-Length: 34

{"title":"secret","author":"duke"}
```

You can now adjust the `jwt-token.json` again and remove the `admin` group and generate a new JWT. With this generated token you shouldn't be able to get a response from the backend and receive 403 Forbidden, as you are authenticated but don't have the correct role.

For further instructions on how to use this tool, have a look at the README on GitHub or the following video of Adam Bien.

» For more hands-one experience with Eclipse MicroProfile JWT Auth, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

Contexts and Dependency Injection (CDI)

Dependency Injection (DI) is one of the central techniques in today's applications and targets Separation of concerns. Not only makes this testing easier, but you are also not in charge to know how to construct the instance of a requested class.

With Java/Jakarta EE we have a specification which (besides other topics) covers this: Contexts and Dependency Injection (short CDI). CDI is also part of the Eclipse MicroProfile project and many other Java/Jakarta EE specifications already use it internally or plan to use it. Please note that I won't cover every aspect of this spec and rather concentrate on the most important parts. For more in-depth knowledge, have a look at the following book.

Specification profile: Contexts and Dependency Injection (CDI)

- Current version: **2.0** in Java/Jakarta EE 8 and MicroProfile **3.3**
- GitHub repository
- Specification homepage
- Basic use case: provide a typesafe dependency injection mechanism

Basic dependency injection with CDI

The main use case for CDI is to provide a typesafe dependency injection mechanism. To make a Java class injectable and managed by the CDI container, you just need a default no-args constructor or a constructor with a `@Inject` annotation.

If you use no further annotations, you have to tell CDI to scan your project for all available beans. You can achieve this which a `beans.xml` file inside `src/main/resources/webapp/WEB-INF` using the `bean-discovery-mode`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans\_1\_1.xsd"
    bean-discovery-mode="all">
</beans>
```

Using this setup, the following `BookService` can inject an instance of the `IsbnValidator` class:

```java
public class IsbnValidator {
    public boolean validateIsbn(String isbn) {
        return isbn.replace("-", "").length() < 13);
    }
}
```

```
public class BookService {

    @Inject
    private IsbnValidator isbnValidator;

    // work with the instance
}
```

You can inject beans via either field-, setter-, constructor-injection or request a bean manually from the CDI runtime:

```
public void storeBook(String bookName, String isbn) {
    if (CDI.current().select(IsbnValidator.class).get().validateIsbn(isbn)) {
        logger.info("Store book with name: " + bookName);
    }
}
```

Once CDI manages a bean, the instances have a well-defined lifecycle and are bound to a scope. You can interact with the lifecycle of a bean while using e.g. `@PostConstruct` or `@PreDestroy`. The default scope, if you don't specify any (like in the example above), is the pseudo-scope `@Dependent`.

With this scope, an instance of your bean is bound to the scope of the bean it gets injected to and won't be shared. However, you can specify the scope of your bean using the available scopes in CDI:

- `@RequestScoped` - bound to an HTTP request
- `@SessionScoped` - bound to the HTTP session of a user
- `@ApplicationScoped` - like a Singleton, one instance per application
- `@ConversationScoped` - bound to a conversation context e.g. wizard-like web app

If you need a more dynamic approach for creating a bean that is managed by CDI you can use the `@Produces` annotation. This gives you access to the `InjectionPoint` which contains metadata about the class who requested an instance:

```
public class LoggerProducer {

    @Produces
    public Logger produceLogger(InjectionPoint injectionPoint) {
        return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName());
    }
}
```

Using qualifiers to specify beans

In the previous chapter, we looked at the simplest scenario where we just have one possible bean to inject. Imagine the following scenario where have multiple implementations of an interface:

```java
public interface BookDistributor {
    void distributeBook(String bookName);
}
```

```java
public class BookPlaneDistributor implements BookDistributor {

    @Override
    public void distributeBook(String bookName) {
        System.out.println("Distributing book by plane");
    }
}
```

```java
public class BookShipDistributor implements BookDistributor {

    @Override
    public void distributeBook(String bookName) {
        System.out.println("Distributing book by ship");
    }
}
```

If we now request a bean of the type BookDistributor, which instance do we get? The `BookPlaneDistributor` or an instance of `BookShipDistributor`?

```java
public class BookStorage {

    @Inject // this will fail
    private BookDistributor bookDistributor;

}
```

well, we get nothing but an exception, as the CDI runtime doesn't know which implementation to inject:

```
WELD-001409: Ambiguous dependencies for type BookDistributor with qualifiers @Default
at injection point \[BackedAnnotatedField\] @Inject private
de.rieckpil.blog.qualifiers.BookStorage.bookDistributors
at de.rieckpil.blog.qualifiers.BookStorage.bookDistributors(BookStorage.java:0)
Possible dependencies:
- Managed Bean \[class de.rieckpil.blog.qualifiers.BookShipDistributor\] with qualifiers
\[@Any @Default\],
- Managed Bean \[class de.rieckpil.blog.qualifiers.BookPlaneDistributor\] with qualifiers
\[@Any @Default\]
```

The stack trace contains an important hint on how to fix such a scenario. If we don't further qualify a bean our beans have the default qualifiers @Any and `@Default`. In the scenario above the BookStorage class requests for a `BookDistributor` and also does not specify anything else, meaning it will get the `@Default` bean.

As there are two beans with this default behavior, dependency injection is not possible (without further adjustments) here. To fix the error above, we have to introduce qualifiers and further specify which concrete bean we want. A qualifier is a Java annotation including `@Qualifier`:

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface PlaneDistributor {
}
```

Once we have this annotation, we can use it both for the implementation and at the injection point:

```
@PlaneDistributor
public class BookPlaneDistributor implements BookDistributor {

    @Override
    public void distributeBook(String bookName) {
        System.out.println("Distributing book by plane");
    }
}
```

```
@Inject
@PlaneDistributor
private BookDistributor bookPlaneDistributor;
```

and now have a proper injection of our requested bean. Above all, you can also always request for all instances matching a Java type using the `Instance<T>` wrapper class:

```
public class BookStorage {

    @Inject
    private Instance<BookDistributor> bookDistributors;

    public void distributeBookToCustomer(String bookName) {
        bookDistributors.forEach(b -> b.distributeBook(bookName));
    }
}
```

Enrich functionality with decorators & interceptors

With CDI we have two mechanisms to enrich/extend the functionality of a class without changing the implementation: Decorators and Interceptors. Decorators allow a type-safe way to *decorate* your actual implementation. Given the following example of an `Account` interface and one implementation:

```
public interface Account {
    Double getBalance();
    void withdrawMoney(Double amount);
}
```

```java
public class CustomerAccount implements Account {

    @Override
    public Double getBalance() {
        return 42.0;
    }

    @Override
    public void withdrawMoney(Double amount) {
        System.out.println("Withdraw money from customer: " + amount);
    }
}
```

We can now write a decorator to make special checks if the amount of money to withdraw meets a threshold:

```java
@Decorator
public abstract class LargeWithdrawDecorator implements Account {

    @Inject
    @Delegate
    private Account account;

    @Override
    public void withdrawMoney(Double amount) {
        if (amount >= 100.0) {
            System.out.println("A large amount of money gets withdrawn!!!");
            // e.g. do further checks
        }
        account.withdrawMoney(amount);
    }
}
```

With interceptors, we get a more generic approach and don't have the same method signature as the intercepted class, rather an `InvocationContext`. This offers more flexibility as we can reuse our interceptor on multiple classes/methods. A lot of cross-cutting logic in Java/Jakarta EE like transactions and security is actually achieved with interceptors.

For an example on how to write interceptors, have a look at one of my previous blog posts. Both decorators and interceptors are inactive by default. To activate them, you either have to specify them in your `beans.xml` file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans\_1\_1.xsd"
bean-discovery-mode="all">
    <decorators>
        <class>de.rieckpil.blog.decorators.LargeWithdrawDecorator</class>
    </decorators>
</beans>
```

or using the `@Priority` annotation and specify the priority value:

```
@Decorator
@Priority(100)
public abstract class LargeWithdrawDecorator implements Account {
}
```

Decouple components with CDI events

Last but not least, the CDI specification provides a sophisticated event notification model. You can use this to decouple your components and use the Observer pattern to notify all listeners once a new event is available.

The event notification in CDI is available both in a synchronous and asynchronous way. The payload of the event can be any Java class and you can use qualifiers to further specialize an event. Firing an event is as simple as the following:

```
public class BookRequestPublisher {

    @Inject
    private Event<BookRequest> bookRequestEvent;

    public void publishNewRequest() {
        this.bookRequestEvent.fire(new BookRequest("MicroProfile 3.0", 1));
   }
}
```

Observing such an event requires the `@Observes` annotation on the receiver-side:

```
public class BookRequestListener {
    public void onBookRequest(@Observes BookRequest bookRequest) {
        System.out.println("New book request incoming: " + bookRequest.toString());
    }
}
```

Using the asynchronous way, you receive a `CompletionStage<T>` as a result and can add further processing steps or handle errors:

```
public class BookRequestPublisher {

    @Inject
    private Event<BookRequest> bookRequestEvent;

    public void publishNewRequest() {

        this.bookRequestEvent
                .fireAsync(new BookRequest("MicroProfile 3.0", 1))
                .handle((request, error) -> {
                    if (error == null) {
                        System.out.println("Successfully fired async event");
                        return request;
                    } else {
                        System.out.println("Error occured during async event");
                        return null;
                    }
                })
                .thenAccept(r -> System.out.println(r));
    }

}
```

Listening to async events requires the `@ObservesAsync` annotation instead of `@Observes`:

```
public void onBookRequestAsync(@ObservesAsync BookRequest bookRequest) {
    System.out.println("New book request incoming async: " + bookRequest.toString());
}
```

» For more hands-one experience with CDI, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

If you are looking for resources to learn more advanced CDI concepts in-depth, have a look at this book.

Jakarta RESTful Web Services (JAX-RS)

The REST architectural pattern is widely adopted when it comes to creating web services. The term was first introduced by Roy Fielding in his dissertation and describes a way for clients to query and manipulate the resources of a server.

With Jakarta RESTful Web Services (JAX-RS), formerly known as Java API for RESTful Web Services, we have a standardized approach to create such web services. This specification is also part of the MicroProfile project since day one. Please note that I won't cover every aspect of this spec (as it is quite large) and rather concentrate on the most important parts.

Specification profile: Jakarta RESTful Web Services (JAX-RS)

- Current version: **2.1** in Java/Jakarta EE 8 and MicroProfile **3.3**
- GitHub repository
- Specification homepage
- Basic use case: develop web services following the Representational State Transfer (REST) pattern

Bootstrap a JAX-RS application

Bootstrapping a JAX-RS application is simple. The main mechanism is to provide a subclass of `javax.ws.rs.core.Application` on your classpath:

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

With `@ApplicationPath` you can specify the path prefix all of your REST endpoints should share. This might be `/api` or `/resources`. Furthermore, you can override the methods of `Application` and register for example all your resources classes, providers and features manually (`getClasses()` method), but you don't have to.

Create REST endpoints

Most of the time you'll use JAX-RS to expose resources of your server on a given path and for a specific HTTP method. The specification provides an annotation to map each HTTP method (GET, PUT, POST, DELETE …) to a Java method. Using the `@Path` annotation you can specify which path to map and also specify path variables:

```
@Path("books")
@Produces(MediaType.APPLICATION\_JSON)
@Consumes(MediaType.APPLICATION\_JSON)
public class BookResource {


    @GET
    @Path("/{id}")
    public Response getBookById(@PathParam("id") Long id,
                                @QueryParam("title") @DefaultValue("") String title) {
       // ...
    }

    @POST
    public Response getBookById(Book bookToStore, @Context UriInfo uriInfo) {
        // ...
    }

    @DELETE
    @Path("/{id}")
    public Response deleteBook(@PathParam("id") Long id,
                               @HeaderParam("User-Agent") String userAgent) {
       // ...
    }

}
```

In the example above you see that the whole class is mapped to the path `/books` with different HTTP methods. `@PathParam` is used to get the value of a path variable and `@QueryParam` for retrieving query parameters of a URL (e.g. `?order=DESC`).

In addition, you can inject further classes into your JAX-RS method and get access e.g. to the `HttpServlet`, `UriInfo` and HTTP headers of the request (`@HeaderParam("nameOfHeader")`).

Next, JAX-RS offers annotations for content-negotiation: `@Consumes` and `@Produces`. In the example above, I'm adding these annotations on class-level, so all methods (which don't specify their own `@Produces`/`@Consumes`) inherit the rules to accept only JSON requests and produces only JSON responses.

In the case, your client sends a payload in the HTTP body (e.g. creating a new book - `@POST` in the example above), you can map the payload to a Java POJO. For JSON payloads, JSON-B is used in the background and for not default payload types (e.g. binary protobuf payload to POJO) you have to register your own `MessageBodyReader` and `MessageBodyWriter`.

The specification defines a standard set of entity providers, which are supported out-of-the-box (e.g. `String` for `text/plain`, `byte[]` for `/`, `File` for `/`, `MultivaluedMap<String, String>` for `application/x-www-form-urlencoded` , etc.). Alongside synchronous and blocking REST endpoints, the specification also supports asynchronous ones:

```
@GET
@Path("async")
public void getBooksAsync(@Suspended final AsyncResponse asyncResponse) {
    // do long-running task with e.g. @Asynchronous annotation
    // form MicroProfile Fault Tolerance or from EJB
    asyncResponse.resume(this.bookStore);
}
```

If you don't specify any other lifecycle (e.g. with `@Singleton` from EJB or a CDI scope) the JAX-RS runtime instantiates a new instance for each request for this resource.

Access external resources

The JAX-RS specification also provides a convenient way to access external resources (e.g. REST endpoints of other services) as a client. We can construct such a client with the `ClientBuilder` from JAX-RS:

```
@PostConstruct
public void initClient() {
    ClientBuilder clientBuilder = ClientBuilder.newBuilder()
    .connectTimeout(5, TimeUnit.SECONDS)
    .readTimeout(5, TimeUnit.SECONDS)
    .register(UserAgentClientFilter.class)
    .register(ClientLoggingResponseFilter.class);

    this.client = clientBuilder.build();
}

@PreDestroy
public void tearDown() {
    this.client.close();
}
```

This `ClientBuilder` allows you to specify metadata like the connect and read timeouts, but also register several features (like you'll see in the next chapter). Make sure to not construct a new `Client` for every request, as they are heavy-weight objects:

> Clients are heavy-weight objects that manage the client-side communication infrastructure. Initialization as well as disposal of a {@code Client} instance may be a rather expensive operation. It is therefore advised to construct only a small number of {@code Client} instances in the application. Client instances must be {@link #close() properly closed} before being disposed to avoid leaking resources. Javadoc of the Client class

Once you have an instance of a `Client`, you can now specify the external resources and create a `WebTarget` instance for each target you want to access:

```
WebTarget quotesApiTarget = client.target("https://quotes.rest").path("qod");
```

With this `WebTarget` instance you can now perform any HTTP operation, set additional header/cookies, set the request body and specify the response type:

```
JsonObject quoteApiResult = this.quotesApiTarget
    .request()
    .header("X-Foo", "bar")
    .accept(MediaType.APPLICATION\_JSON)
    .get()
    .readEntity(JsonObject.class);
```

Furthermore, JAX-RS offer reactive support for requesting external resources with `.rx()`:

```
CompletionStage<JsonObject> rxQuoteApiResult = this.quotesApiTarget
    .request()
    .header("X-Foo", "bar")
    .accept(MediaType.APPLICATION\_JSON)
    .rx()
    .get(JsonObject.class);
```

Intercept the request and response flow

There are various entry points to intercept the flow of a JAX-RS resource and including client requests. To give you an idea of how the overall architecture looks like, have a look at the following image:

```
                         CLIENT                                                                              SERVER
+----------------------------------------------------------------+      +------------------------------------------------------------------------------------+
|                                                                |      |                                                                                    |
|  ClientRequestFilter -> WriterInterceptor ->  MessageBodyWriter ---------->  PreMatchingRequestFilter ->  ContainerRequestFilter ---> ReaderInterceptor     |
|                                                                |      |                                                                       |            |
|                                                                |      |                                                                       v            |
|                                                                |      |                                                                MessageBodyReader   |
|                                                                |      |                                                                       |            |
|                                                                |      |                                                                       v            |
|                                                                |      |                                                                 ResourceMethod     |
|                                                                |      |                                                                       |            |
|                                                                |      |                                                                       v            |
|  MessageBodyReader <- ReaderInterceptor <- ClientResponseFilter <-----------  MessageBodyWriter   <---   WriterInterceptor <--- ContainerResponseFilter    |
|                                                                |      |                                                                                    |
+----------------------------------------------------------------+      +------------------------------------------------------------------------------------+
```

```
@Provider
@PreMatching
public class HttpMethodModificationFilter implements ContainerRequestFilter {

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {

        if(requestContext.getMethod().equalsIgnoreCase("DELETE")) {
            requestContext.setMethod("GET");
        }

    }
}
```

Next, you can add e.g. common headers to the response of your resource method with a `ContainerResponseFilter`:

```
@Priority(100)
@Provider
public class XPoweredByResponseHeaderFilter implements ContainerResponseFilter {

    @Override
    public void filter(ContainerRequestContext requestContext,
                       ContainerResponseContext responseContext) throws IOException {
        responseContext.getHeaders().add("X-Powered-By", "MicroProfile");
    }
}
```

With `@Priority` you can set the order of your filter once you use multiple and rely on execution in order. For the client-side, we can add a filter to first log all HTTP headers of the incoming response with `@ClientResponseFilter`:

```
@Provider
public class ClientLoggingResponseFilter implements ClientResponseFilter {

    @Override
    public void filter(ClientRequestContext requestContext,
                       ClientResponseContext responseContext) throws IOException {
        System.out.println("Response filter for JAX-RS Client");
        responseContext.getHeaders().forEach((k, v) -> System.out.println(k + ":" + v));
    }
}
```

» For more hands-one experience with JAX-RS, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

JSON Binding (JSON-B)

JSON is the current de-facto data format standard for exposing data via APIs. The Java ecosystem offers a bunch of libraries to create JSON from Java objects and vice-versa (GSON, Jackson, etc.).

With the release of Java EE 8 and the JSR-367, we now have a standardized approach for this: JSON-B. With the transition of Java EE to the Eclipse Foundation, this specification is now renamed to Jakarta JSON Binding (JSON-B). In addition, this spec is also part of the Eclipse MicroProfile project.

Specification profile: JSON Binding (JSON-B)

- Current version: **1.0** in Java/Jakarta EE 8 and MicroProfile **3.3**
- GitHub repository
- Specification homepage
- Basic use case: Convert Java objects from and to JSON

Map objects from and to JSON

The central use case for JSON-B is mapping Java objects to and from JSON strings. To provide you an example, I'm using the following POJO:

```java
public class Book {

    private String title;
    private LocalDate creationDate;
    private long pages;
    private boolean isPublished;
    private String author;
    private BigDecimal price;

    // constructors, getters & setters
}
```

Mapping Java objects and JSON messages requires an instance of the `Jsonb`. The specification defines a builder to create such an object. This instance can then be used for both mapping Java objects from and to JSON:

```java
Book book = new Book("Java 11", LocalDate.now(), 1, false, "Duke", new BigDecimal(44.444)
);

Jsonb jsonb = JsonbBuilder.create();

String resultJson = jsonb.toJson(book);

Book serializedBook = jsonb.fromJson(resultJson, Book.class);
```

With no further configuration or adjustments, the JSON result contains all Java member variables (ignoring null values) as attributes in camel case. Furthermore, you can also map a collection of Java objects to and from JSON arrays in a type-safe manner:

```
List<Book> bookList = new ArrayList<>();
bookList.add(new Book("Java 11", LocalDate.now(), 100, true, "Duke", new BigDecimal(39.95
)));
bookList.add(new Book("Java 15", LocalDate.now().plus(365, ChronoUnit.DAYS), 110, false,
"Duke", new BigDecimal(50.50)));

Jsonb jsonb = JsonbBuilder.create();

String result = jsonb.toJson(bookList);

List<Book> serializedBookList = jsonb
    .fromJson(result, new ArrayList<Book>(){}.getClass().getGenericSuperclass());
```

Configure the mapping of attributes

Sometimes the default mapping strategy of JSON-B might not fit your requirements and you want to e.g. customize the JSON attribute name or the date/number format. The specification offers a set of annotations to override the default mapping behavior, which can be applied to your Java POJO class.

With `@JsonbProperty` you can adjust the name of the JSON attribute name. If you use this annotation on a field level, it will affect both serialization and deserialization. On getter methods it affects only serialization and on setters only deserialization back to Java objects:

```
@JsonbProperty("book-title")
private String title;
```

Next, you can use `@JsonbTransient` to avoid the serialization of a specific attribute to JSON at all:

```
@JsonbTransient
private boolean isPublished;
```

If you plan to override the default behavior to not include null values to the JSON message, `@JsonbNillable` offers a way to do this. This annotation can only be used on class level and will affect all attributes:

```
@JsonbNillable
public class Book {
}
```

For those use cases where you just want one attribute to be serialized to null, you can use `@JsonbProperty(nillable=true)` on fields/getters/setters. In addition, you are able to adjust the format of dates and numbers with `@JsonbDateFormat` and `@JsonbNumberFormat` and specify your custom format:

```
@JsonbDateFormat("dd.MM.yyyy")
private LocalDate creationDate;

@JsonbNumberFormat("#0.00")
private BigDecimal price;
```

Above all, if you don't want JSON-B to use the default no-arg constructor to deserialize JSON to Java objects, you can specify a custom constructor and use the annotation `@JsonbCreator`:

```
public class Book {

    // ...

    @JsonbCreator
    public Book(@JsonbProperty("book-title") String title) {
        this.title = title;
    }

}
```

Make sure you use this annotation only once per class.

Define metadata for mapping JSON objects

Applying e.g. the `@JsonbDateFormat` to all your POJOs so they are all compliant to your custom date format, might be cumbersome and error-prone. Furthermore, if you use the annotations above to customize the mapping, you are not able to provide multiple representations if different clients require their own.

You can solve such requirements with a `JsonbConfig` instance and define global metadata for the mapping. Together with this configuration class, you can create a configured `Jsonb` instance and apply the mapping rules to all mappings of this instance:

```
Book book = new Book("Java 11", LocalDate.now(), 1, false, null, new BigDecimal(50.50));

JsonbConfig config = new JsonbConfig()
    .withNullValues(false)
    .withFormatting(true)
    .withPropertyOrderStrategy(PropertyOrderStrategy.LEXICOGRAPHICAL)
    .withPropertyNamingStrategy(PropertyNamingStrategy.LOWER\_CASE\_WITH\_UNDERSCORES)
    .withDateFormat("dd-MM-YYYY", Locale.GERMAN);

Jsonb jsonb = JsonbBuilder.create(config);

String jsonString = jsonb.toJson(book);
```

Using this `JsonbConfig`, you are also able to configure things you can't with the annotations of the previous chapter: pretty-printing, locale information, naming strategies, ordering of attributes, encoding information, binary data strategies, etc. Have a look at the official user guide for all configuration attributes.

Provide a custom JSON-B mapping strategy

If all of the above solutions don't meet your requirements for mapping Java objects to and from JSON, you can implement your own `JsonAdpater` and get full access to serialization and deserialization:

```
public class BookAdapter implements JsonbAdapter<Book, JsonObject> {

    @Override
    public JsonObject adaptToJson(Book book) throws Exception {
        return Json.createObjectBuilder()
                .add("title", book.getTitle() + " - " + book.getAuthor())
                .add("creationDate", book.getCreationDate().toEpochDay())
                .add("pages", book.getPages())
                .add("price", book.getPrice().multiply(BigDecimal.valueOf(2l)))
                .build();
    }

    @Override
    public Book adaptFromJson(JsonObject jsonObject) throws Exception {
        Book book = new Book();
        book.setTitle(jsonObject.getString("title").split("-")\[0\].trim());
        book.setAuthor(jsonObject.getString("title").split("-")\[1\].trim());
        book.setPages(jsonObject.getInt("pages"));
        book.setPublished(false);
        book.setPrice(BigDecimal.valueOf(jsonObject.getJsonNumber("price").longValue()));
        book.setCreationDate(LocalDate.ofEpochDay(jsonObject.getInt("creationDate")));
        return book;
    }
}
```

With this adapter, you have full access to manage your JSON representation and to the deserialization logic. In this example, I'm using both the title and author for the final book title and concatenate both.

Keep in mind that with a custom adapter, your JSON-B annotations on your POJO are overruled. To make use of this `JsonAdapter`, you have to register it using a custom `JsonbConfig`:

```
JsonbConfig config = new JsonbConfig()
    .withAdapters(new BookAdapter());

Jsonb jsonb = JsonbBuilder.create(config);

String jsonString = jsonb.toJson(book);

Book serializedBook = jsonb.fromJson(jsonString, Book.class);
```

If you need more low-level access to the serialization and deserialization, have a look at the `JsonbSerializer` and `JsonbDeserializer` interface (an example can be found in the official user guide).

» For more hands-one experience with JSON-B, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

JSON Processing (JSON-P)

Besides binding and converting JSON from an to Java objects with JSON-B, the Java EE specification (now Jakarta EE) offers a spec to process JSON data: JSON Processing (JSON-P).

With this spec, you can easily create, write, read, stream, transform and query JSON objects. This specification is also part of the Eclipse MicroProfile project and provides a simple API to handle and further process JSON data structures as you'll see it in the following examples. === Specification profile: JSON Processing (JSON-P)

- Current version: **1.1** in Java/Jakarta EE 8 and MicroProfile **3.3**
- GitHub repository
- Specification homepage
- Basic use case: Process JSON messages (parse, generate, transform and query)

Construct JSON objects using JSON-P

With JSON-P you can easily build JSON objects on-demand. You can create a `JsonObjectBuilder` using the `Json` class and build the JSON object while adding new attributes to the object:

```
JsonObject json = Json.createObjectBuilder()
    .add("name", "Duke")
    .add("age", 42)
    .add("skills",
        Json.createArrayBuilder()
        .add("Java SE").add("Java EE").build())
    .add("address",
        Json.createObjectBuilder()
        .add("street", "Mainstreet")
        .add("city", "Jakarta")
        .build())
    .build();
```

If you print this object, you already have a valid JSON and can return this e.g. from a JAX-RS endpoint or use it as an HTTP request body:

```
{"name":"Duke","age":42,"skills":\["Java SE","Java
EE"\],"address":{"street":"Mainstreet","city":"Jakarta"}}
```

You are not limited to create JSON objects only, you can also request for a `JsonArrayBuilder` and start constructing your JSON array:

```
JsonArray jsonArray = Json.createArrayBuilder()
    .add("foo")
    .add("bar")
    .add("duke")
    .build();
```

Write JSON objects

Given a JSON object, you can also write it to a different source using JSON-P and its `JsonWriterFactory`. As an example, I'm writing a JSON object to a `File` in pretty-print:

```
private void prettyPrintJsonToFile(JsonObject json) throws IOException {
    Map<String,Boolean> config = new HashMap<>();
    config.put(JsonGenerator.PRETTY\_PRINTING, true);

    JsonWriterFactory writerFactory = Json.createWriterFactory(config);
    try (OutputStream outputStream = new FileOutputStream(new File("/tmp/output.json"));
         JsonWriter jsonWriter = writerFactory.createWriter(outputStream)) {

        jsonWriter.write(json);
    }
}
```

The `JsonWriterFactory` accepts any `Writer` or `OutputStream` to instantiate the `JsonWriter`:

```
private void prettyPrintJsonToConsole(JsonObject json) throws IOException {
    Map<String,Boolean> config = new HashMap<>();
    config.put(JsonGenerator.PRETTY\_PRINTING, true);

    JsonWriterFactory writerFactory = Json.createWriterFactory(config);
    try (Writer stringWriter = new StringWriter();
         JsonWriter jsonWriter = writerFactory.createWriter(stringWriter)) {
        jsonWriter.write(json);
        System.out.println(stringWriter);
    }
}
```

Using the JSON object from the chapter above, the output on the console will look like the following:

```
{
    "name": "Duke",
    "age": 42,
    "skills": [
        "Java SE",
        "Java EE"
    ],
    "address": {
        "street": "Mainstreet",
        "city": "Jakarta"
    }
}
```

Read JSON with JSON-P

The specification also provides a convenient way to read and parse JSON from a given source (e.g. `File` or `String`). To create a `JsonReader` instance, you either have to provide a `InputStream` or a `Reader`. As an example, I'm reading from both a `String` and a `File` on the classpath:

```
private void readFromString() {
    JsonReader jsonReader = Json.createReader(
    new StringReader("{\"name\":\"duke\",\"age\":42,\"skills\":\[\"Java SE\", \"Java EE
\"]}"));
    JsonObject jsonObject = jsonReader.readObject();
    System.out.println(jsonObject);
}

private void readFromFile() {
    JsonReader jsonReader = Json.createReader(this.getClass().getClassLoader()
    .getResourceAsStream("books.json"));
    JsonArray jsonArray = jsonReader.readArray();
    System.out.println(jsonArray);
}
```

If the JSON is not valid, the `JsonReader` throws a `JsonParsingExcpetion` while parsing it and will give a hint about what is wrong e.g. `Invalid token=SQUARECLOSE at (line no=1, column no=54, offset=53). Expected tokens are: [COLON]`.

Stream JSON data

For use cases where you have to process big JSON objects (which might not fit into memory), you should have a look at the streaming options of JSON-P. The specification says the following about its streaming capabilities:

> Unlike the Object model this offers more generic access to JSON strings that may change more often with attributes added or similar structural changes. Streaming API is also the preferred method for very large JSON strings that could take more memory reading them altogether through the Object model API.

Streaming works for both parsing and generating JSON objects. To parse and process a big JSON object, the spec provides the `JsonParser`:

```
String jsonString =
"{\\"name\\":\\"duke\\",\\"isRetired\\":false,\\"age\\":42,\\"skills\\":\[\\"Java SE\\",
\\"Java EE\\"\]}";
try (JsonParser parser = Json.createParser(new StringReader(jsonString))) {
    while (parser.hasNext()) {
        final Event event = parser.next();
        switch (event) {
            case START\_ARRAY:
                System.out.println("Start of array");
                break;
            case END\_ARRAY:
                System.out.println("End of array");
                break;
            case KEY\_NAME:
                System.out.println("Key found " + parser.getString());
                break;
            case VALUE\_STRING:
                System.out.println("Value found " + parser.getString());
                break;
            case VALUE\_NUMBER:
                System.out.println("Number found " + parser.getLong());
                break;
            case VALUE\_TRUE:
                System.out.println(true);
                break;
            case VALUE\_FALSE:
                System.out.println(false);
                break;
        }
    }
}
```

This offers rather low-level access to the JSON object and you can access all `Event` objects (e.g. START_ARRAY, KEY_NAME, VALUE_STRING) while parsing. For creating a JSON object in a streaming-fashion, you can use the `JsonGenerator` class and write to any source using a `Writer` or `OutputStream`:

```
StringWriter stringWriter = new StringWriter();

try (JsonGenerator jsonGenerator = Json.createGenerator(stringWriter)) {
    jsonGenerator.writeStartArray()
        .writeStartObject()
        .write("name", "duke")
        .writeEnd()
        .writeStartObject()
        .write("name", "jakarta")
        .writeEnd()
        .writeEnd();
    jsonGenerator.flush();
}

System.out.println(stringWriter.toString());
```

Transform JSON with JsonPointer, JsonPatch and JsonMergePatch

Since JSON-P 1.1, the specification offers a great way to query and transform JSON structures using the following standardized JSON operations:

- JSON Pointer (official RFC)
- JSON Patch (official RFC)

- JSON MergePatch (official RFC)

Identify a specific value with JSON Pointer

If your JSON object contains several sub-objects and arrays and you have to find the value of a specific attribute, iterating over the whole object is cumbersome. With JSON Pointer you can specify an expression and point to a specific attribute and directly access it.

The expression is defined in the official RFC. Once you have a JSON pointer in place, you can get the value, remove it, replace it, add a new and check for existence with JSON-P and its `JsonPointer` class:

```
String jsonString = "{\"name\":\"duke\",\"age\":42,\"skills\":\[\"Java SE\", \"Java EE\"
]}";

JsonObject jsonObject = Json.createReader(new StringReader(jsonString)).readObject();

JsonPointer arrayElementPointer = Json.createPointer("/skills/1");
JsonPointer agePointer = Json.createPointer("/age");
JsonPointer namePointer = Json.createPointer("/name");
JsonPointer addressPointer = Json.createPointer("/address");
JsonPointer tagsPointer = Json.createPointer("/tags");

System.out.println("Get array element with pointer: " + arrayElementPointer.getValue
(jsonObject).toString());
System.out.println("Remove age with pointer: " + agePointer.remove(jsonObject));
System.out.println("Replace name with pointer: " + namePointer.replace(jsonObject, Json
.createValue("john")));
System.out.println("Check address with pointer: " + addressPointer.containsValue
(jsonObject));
System.out.println("Add tags with pointer: " + tagsPointer.add(jsonObject, Json
.createArrayBuilder().add("nice").build()));
```

Define a sequence of operations to apply using JSON Patch

Similar to the JSON Pointer in the example above, you can define a set of operations to apply on a given JSON with JSON Patch. The possible operations to apply to a JSON are defined in the official RFC. As an example, I'm modifying an existing JSON with `JsonPatch` like the following:

```
String jsonString = "{\"name\":\"duke\",\"age\":42,\"skills\":\[\"Java SE\", \"Java EE\"
]}";

JsonObject jsonObject = Json.createReader(new StringReader(jsonString)).readObject();

JsonPatch patch = Json.createPatchBuilder()
    .add("/isRetired", false)
    .add("/skills/2", "Jakarta EE")
    .remove("/age")
    .replace("/name", "duke two")
    .build();

JsonObject patchedJson = patch.apply(jsonObject);
System.out.println("Patched JSON: " + patchedJson);
```

The patched JSON object looks like the following:

```
Patched JSON: {"name":"duke two","skills":["Java SE","Java EE","Jakarta
EE"],"isRetired":false}
```

Merge two JSON objects with JSON Merge Patch

If you want to merge a given JSON object with another JSON, you can make use of the JSON Merge Patch. With this, you first have to define how the merge JSON object looks like and can then apply it to a target JSON structure.

```
String jsonString = "{\"name\":\"duke\",\"age\":42,\"skills\":\[\"Java SE\", \"Java EE\"
]}";

JsonObject jsonObject = Json.createReader(new StringReader(jsonString)).readObject();

JsonObject merge = Json.createObjectBuilder()
    .add("name", "duke2")
    .add("isEmployee", true)
    .add("skills",
        Json.createArrayBuilder()
            .add("CSS")
            .add("HTML")
            .add("JavaScript")
        .build())
    .build();

JsonMergePatch mergePatch = Json.createMergePatch(merge);
JsonValue mergedJson = mergePatch.apply(jsonObject);
System.out.println("Merged JSON: " + mergedJson);
```

The merged JSON in this example looks like the following:

```
Merged JSON:
{"name":"duke2","age":42,"skills":["CSS","HTML","JavaScript"],"isEmployee":true}
```

For more information about the JSON Merge Patch, have a look at the official RFC.

» For more hands-one experience with JSON-P, watch the corresponding video course section of the Getting Started with Eclipse MicroProfile series.

Integration with Jakarta EE

Fortunately all major application server vendors (Open Liberty, Payara, WildFly, TomEE) support and implement both Eclipse MicroProfile and Jakarta EE. Given this fact, we can add MicroProfile to an existing Java/Jakarta EE application without further configuration.

So you can either add missing parts like persistence to your MicroProfile application with adding Jakarta EE or add the missing parts for a microservice based architecture to your existing Jakarta EE application.

A minimal `pom.xml` for a Jakarta EE and MicroProfile project might look like the following:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.your.company</groupId>
    <artifactId>mp-and-jakarta-ee</artifactId>
    <version>1.0.0</version>
    <packaging>war</packaging>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>jakarta.platform</groupId>
            <artifactId>jakarta.jakartaee-api</artifactId>
            <version>8.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.eclipse.microprofile</groupId>
            <artifactId>microprofile</artifactId>
            <version>3.3</version>
            <type>pom</type>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

If you are looking for a simple way to bootstrap a Jakarta EE + MicroProfile project, have a look at one of my Maven Archetypes here.

For a full application setup with a React frontend and PostgreSQL, I'm providing you a project template here.

Further resources on Eclipse MicroProfile

Further resources can be found here:

- Official homepage of Eclipse MicroProfile
- Official blog of Eclipse MicroProfile
- YouTube channel of the Eclipse Foundation
- Project overview of Eclipse MicroProfile in general
- Eclipse MicroProfile Google Group to ask questions or get involved
- Umbrella GitHub repository
- Eclipse MicroProfile mailing list
- Official Wiki page
- Further tutorials on my blog

Changelog notes

Updates with Eclipse MicroProfile 3.3

- Rest Client specification update to version 1.4 (TCK improvements, CDI usage in `ClientHeadersFactory`, and further small improvements)
- Config specification update to version 1.4 (fixes classloading issues, adding new build-in converters, improvements to the `Converter` interface)
- Fault Tolerance specification update to version 2.1 (enrichment to annotations with new attributes, relaxing requirements on `Future` and `CompletionStage` implementation)
- Metrics specification update to 2.3 (introduction of `@SimplyTimed`, new base metric derived from RESTful stats)
- Health specification update to 2.2 (API and TCK improvements, upgrade to CDI version 2.0)
- Further release notes can be found on the MicroProfile blog
- Get the full specification document for this release here

Updates with Eclipse MicroProfile 3.2

- Correct backward compatibility for the Metrics specification (this reverts a problematic change from Metrics 2.1 where Gauges were required to return subclasses of `java.lang.Number`)
- Get a more detailed changelog for this release here

Updates with Eclipse MicroProfile 3.1

- Metrics specification update to version 2.1 (clarification for the Gauge metric, `.reusable()` method of the `MetadataBuilder` now takes a boolean value as input, TCK improvements)
- Health Check specification update to version 2.1 (default procedures of the vendor can be disabled, Javadoc improvements, updates to the test setup)
- Get a detailed review of what changed with this release here