**1. Running on WildFly**

A few weeks ago WildFly has introduced the “Fat JAR” packaging feature. This feature is fully supported since WildFly 21. We can apply it during a Maven build by including wildfly-jar-maven-plugin to the pom.xml file. What is important, we don’t have to re-design an application to run it inside a bootable JAR.

In order to use the “Fat JAR” packaging feature, we need to add the package execution goal. Then we should install two features inside the configuration section. The first of them, the jaxrs-server feature, is a layer that allows us to build a typical REST application. The second of them, the microprofile-platform feature, enables MicroProfile on the WildFly server.

<profile>

<id>bootable-jar</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<plugin>

<groupId>org.wildfly.plugins</groupId>

<artifactId>wildfly-jar-maven-plugin</artifactId>

<version>2.0.2.Final</version>

<executions>

<execution>

<goals>

<goal>package</goal>

</goals>

</execution>

</executions>

<configuration>

<feature-pack-location>

wildfly@maven(org.jboss.universe:community-universe)#${version.wildfly}

</feature-pack-location>

<layers>

<layer>jaxrs-server</layer>

<layer>microprofile-platform</layer>

</layers>

</configuration>

</plugin>

</plugins>

</build>

</profile>

Finally, we just need to execute the following command to build and run our “Fat JAR” application on WildFly.

$ mvn package wildfly-jar:run

If we run multiple applications on the same machine, we would have to override default HTTP and management ports. To do that we need to add the jvmArguments section inside configuration. We may insert there any number of JVM arguments. In that case, the required arguments are jboss.http.port and jboss.management.http.port.

<configuration>

...

<jvmArguments>

<jvmArgument>-Djboss.http.port=8090</jvmArgument>

<jvmArgument>-Djboss.management.http.port=9090</jvmArgument>

</jvmArguments>

</configuration>

**2. Creating JAX-RS applications**

In the first step, we will create simple REST applications with JAX-RS. WildFly provides all the required libraries, but we need to include both these artifacts for the compilation phase.

<dependency>

<groupId>org.jboss.spec.javax.ws.rs</groupId>

<artifactId>jboss-jaxrs-api\_2.1\_spec</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>jakarta.enterprise</groupId>

<artifactId>jakarta.enterprise.cdi-api</artifactId>

<scope>provided</scope>

</dependency>

Then, we should set the dependencyManagement section. We will use BOM provided by WildFly for both MicroProfile and Jakarta EE.

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.wildfly.bom</groupId>

<artifactId>wildfly-jakartaee8-with-tools</artifactId>

<version>${version.wildfly}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

<groupId>org.wildfly.bom</groupId>

<artifactId>wildfly-microprofile</artifactId>

<version>${version.wildfly}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

Here’s the JAX-RS controller inside employee-service. It uses an in-memory repository bean. It also injects a random delay to all exposed HTTP endpoints with the @Delay annotation. To clarify, I’m just setting it for future use, in order to present the metrics and fault tolerance features.

@Path("/employees")

@ApplicationScoped

@Produces(MediaType.APPLICATION\_JSON)

@Consumes(MediaType.APPLICATION\_JSON)

@Delay

public class EmployeeController {

@Inject

EmployeeRepository repository;

@POST

public Employee add(Employee employee) {

return repository.add(employee);

}

@GET

@Path("/{id}")

public Employee findById(@PathParam("id") Long id) {

return repository.findById(id);

}

@GET

public List<Employee> findAll() {

return repository.findAll();

}

@GET

@Path("/department/{departmentId}")

public List<Employee> findByDepartment(@PathParam("departmentId") Long departmentId) {

return repository.findByDepartment(departmentId);

}

@GET

@Path("/organization/{organizationId}")

public List<Employee> findByOrganization(@PathParam("organizationId") Long organizationId) {

return repository.findByOrganization(organizationId);

}

}

Here’s a definition of the delay interceptor class. It is annotated with a base @Interceptor and custom @Delay. It injects a random delay between 0 and 1000 milliseconds to each method invoke.

@Interceptor

@Delay

public class AddDelayInterceptor {

Random r = new Random();

@AroundInvoke

public Object call(InvocationContext invocationContext) throws Exception {

Thread.sleep(r.nextInt(1000));

System.out.println("Intercept");

return invocationContext.proceed();

}

}

Finally, let’s just take a look on the custom @Delay annotation.

@InterceptorBinding

@Target({METHOD, TYPE})

@Retention(RUNTIME)

public @interface Delay {

}

**3. Enable metrics for MicroProfile microservices**

Metrics is one of the core MicroProfile modules. Data is exposed via REST over HTTP under the /metrics base path in two different data formats for GET requests. These formats are JSON and OpenMetrics. The OpenMetrics text format is supported by Prometheus. In order to enable the MicroProfile metrics, we need to include the following dependency to Maven pom.xml.

<dependency>

<groupId>org.eclipse.microprofile.metrics</groupId>

<artifactId>microprofile-metrics-api</artifactId>

<scope>provided</scope>

</dependency>

To enable the basic metrics we just need to annotate the controller class with @Timed.

@Path("/employees")

@ApplicationScoped

@Produces(MediaType.APPLICATION\_JSON)

@Consumes(MediaType.APPLICATION\_JSON)

@Delay

@Timed

public class EmployeeController {

...

}

The /metrics endpoint is available under the management port. Firstly, let’s send some test requests, for example to the GET /employees endpoint. The application employee-service is available on http://localhost:8080/. Then let’s call the endpoint http://localhost:9990/metrics. Here’s a full list of metrics generated for the findAll method. Similar metrics would be generated for all other HTTP endpoints.

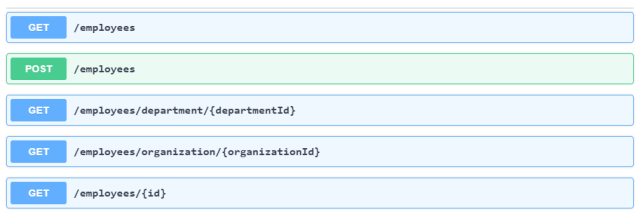
A close-up of a document

Description automatically generated with medium confidence

**4. Generate OpenAPI specification**

The REST API specification is another essential thing for all microservices. So, it is not weird that the OpenAPI module is a part of a MicroProfile core. The API specification is automatically generated after including the microprofile-openapi-api module. This module is a part microprofile-platform layer defined for wildfly-jar-maven-plugin.

After starting the application we may access OpenAPI documentation by calling http://localhost:8080/openapi endpoint. Then, we can copy the result to the Swagger editor. The graphical representation of the employee-service API is visible below.



**5. Microservices inter-communication with MicroProfile REST client**

The department-service calls endpoint GET /employees/department/{departmentId} from the employee-service. Then it returns a department with a list of all assigned employees.

@Getter

@Setter

@NoArgsConstructor

@AllArgsConstructor

public class Department {

private Long id;

private String name;

private Long organizationId;

private List<Employee> employees = new ArrayList<>();

}

Of course, we need to include the REST client module to the Maven dependencies.

<dependency>

<groupId>org.eclipse.microprofile.rest.client</groupId>

<artifactId>microprofile-rest-client-api</artifactId>

<scope>provided</scope>

</dependency>

The MicroProfile REST module allows defining a client declaratively. We should annotate the client interface with @RegisterRestClient. The rest of the implementation is rather obvious.

@Path("/employees")

@RegisterRestClient(baseUri = "http://employee-service:8080")

public interface EmployeeClient {

@GET

@Path("/department/{departmentId}")

@Produces(MediaType.APPLICATION\_JSON)

List<Employee> findByDepartment(@PathParam("departmentId") Long departmentId);

}

Finally, we just need to inject the EmployeeClient bean to the controller class.

@Path("/departments")

@ApplicationScoped

@Produces(MediaType.APPLICATION\_JSON)

@Consumes(MediaType.APPLICATION\_JSON)

@Timed

public class DepartmentController {

@Inject

DepartmentRepository repository;

@Inject

EmployeeClient employeeClient;

@POST

public Department add(Department department) {

return repository.add(department);

}

@GET

@Path("/{id}")

public Department findById(@PathParam("id") Long id) {

return repository.findById(id);

}

@GET

public List<Department> findAll() {

return repository.findAll();

}

@GET

@Path("/organization/{organizationId}")

public List<Department> findByOrganization(@PathParam("organizationId") Long organizationId) {

return repository.findByOrganization(organizationId);

}

@GET

@Path("/organization/{organizationId}/with-employees")

public List<Department> findByOrganizationWithEmployees(@PathParam("organizationId") Long organizationId) {

List<Department> departments = repository.findByOrganization(organizationId);

departments.forEach(d -> d.setEmployees(employeeClient.findByDepartment(d.getId())));

return departments;

}

}

The MicroProfile project does not implement service discovery patterns. There are some frameworks built on top of MicroProfile that provide such kind of implementation, for example, KumuluzEE. If you do not deploy our applications on OpenShift you may add the following entry in your /etc/hosts file to test it locally.

127.0.0.1 employee-service

Finally, let’s call endpoint GET /departments/organization/{organizationId}/with-employees. The result is visible in the picture below.

Text

Description automatically generated

**6. Java microservices fault tolerance with MicroProfile**

To be honest, fault tolerance handling is my favorite feature of MicroProfile. We may configure them on the controller methods using annotations. We can choose between @Timeout, @Retry, @Fallback and @CircuitBreaker. Alternatively, it is possible to use a mix of those annotations on a single method. As you probably remember, we injected a random delay between 0 and 1000 milliseconds into all the endpoints exposed by employee-service. Now, let’s consider the method inside department-service that calls endpoint GET /employees/department/{departmentId} from employee-service. Firstly, we will annotate that method with @Timeout as shown below. The current timeout is 500 ms.

public class DepartmentController {

@Inject

DepartmentRepository repository;

@Inject

EmployeeClient employeeClient;

...

@GET

@Path("/organization/{organizationId}/with-employees")

@Timeout(500)

public List<Department> findByOrganizationWithEmployees(@PathParam("organizationId") Long organizationId) {

List<Department> departments = repository.findByOrganization(organizationId);

departments.forEach(d -> d.setEmployees(employeeClient.findByDepartment(d.getId())));

return departments;

}

}

Before calling the method, let’s create an exception mapper. If TimeoutException occurs, the department-service endpoint will return status HTTP 504 - Gateway Timeout.

@Provider

public class TimeoutExceptionMapper implements

ExceptionMapper<TimeoutException> {

public Response toResponse(TimeoutException e) {

return Response.status(Response.Status.GATEWAY\_TIMEOUT).build();

}

}

Then, we may proceed to call our test endpoint. Probably 50% of requests will finish with the result visible below.

Text

Description automatically generated

On the other hand, we may enable a retry mechanism for such an endpoint. After that, the change for receive status HTTP 200 OK becomes much bigger than before.

@GET

@Path("/organization/{organizationId}/with-employees")

@Timeout(500)

@Retry(retryOn = TimeoutException.class)

public List<Department> findByOrganizationWithEmployees(@PathParam("organizationId") Long organizationId) {

List<Department> departments = repository.findByOrganization(organizationId);

departments.forEach(d -> d.setEmployees(employeeClient.findByDepartment(d.getId())));

return departments;

}

**7. Deploy MicroProfile microservices on OpenShift**

We can easily deploy MicroProfile Java microservices on OpenShift using the JKube plugin. It is a successor of the deprecated Fabric8 Maven Plugin. Eclipse JKube is a collection of plugins and libraries that are used for building container images using Docker, JIB or S2I build strategies. It generates and deploys Kubernetes and OpenShift manifests at compile time too. So, let’s add openshift-maven-plugin to the pom.xml file.

The configuration visible below sets 2 replicas for the deployment and enforces using health checks. In addition to this, openshift-maven-plugin generates the rest of a deployment config based on Maven pom.xml structure. For example, it generates employee-service-deploymentconfig.yml, employee-service-route.yml, and employee-service-service.yml for the employee-service application.

<plugin>

<groupId>org.eclipse.jkube</groupId>

<artifactId>openshift-maven-plugin</artifactId>

<version>1.0.2</version>

<executions>

<execution>

<id>jkube</id>

<goals>

<goal>resource</goal>

<goal>build</goal>

</goals>

</execution>

</executions>

<configuration>

<resources>

<replicas>2</replicas>

</resources>

<enricher>

<config>

<jkube-healthcheck-wildfly-jar>

<enforceProbes>true</enforceProbes>

</jkube-healthcheck-wildfly-jar>

</config>

</enricher>

</configuration>

</plugin>

In order to deploy the application on OpenShift we need to run the following command.

$ mvn oc:deploy -P bootable-jar-openshift

Since the property enforceProbes has been enabled openshift-maven-plugin adds liveness and readiness probes to the DeploymentConfig. Therefore, we need to implement both these endpoints in our MicroProfile applications. MicroProfile provides a smart mechanism for creating liveness and readiness health checks. We just need to annotate the class with @Liveness or @Readiness, and implement the HealthCheck interface. Here’s the example implementation of the liveness endpoint.

@Liveness

@ApplicationScoped

public class LivenessEndpoint implements HealthCheck {

@Override

public HealthCheckResponse call() {

return HealthCheckResponse.up("Server up");

}

}

On the other hand, the implementation of the readiness probe also verifies the status of the repository bean. Of course, it is just a simple example.

@Readiness

@ApplicationScoped

public class ReadinessEndpoint implements HealthCheck {

@Inject

DepartmentRepository repository;

@Override

public HealthCheckResponse call() {

HealthCheckResponseBuilder responseBuilder = HealthCheckResponse

.named("Repository up");

List<Department> departments = repository.findAll();

if (repository != null && departments.size() > 0)

responseBuilder.up();

else

responseBuilder.down();

return responseBuilder.build();

}

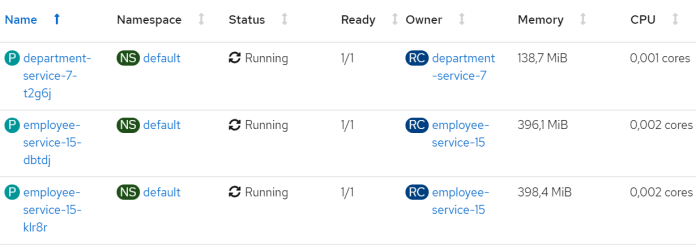
}

After deploying both employee-service and department-service application we may verify a list of DeploymentConfigs.

Graphical user interface, application

Description automatically generated

We can also navigate to the OpenShift console. Let’s take a look at a list of running pods. There are two instances of the employee-service and a single instance of department-service.



**8. MicroProfile OpenTracing with Jaeger**

Tracing is another important pattern in microservices architecture. The OpenTracing module is a part of MicroProfile specification. Besides the microprofile-opentracing-api library we also need to include the opentracing-api module.

<dependency>

<groupId>org.eclipse.microprofile.opentracing</groupId>

<artifactId>microprofile-opentracing-api</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>io.opentracing</groupId>

<artifactId>opentracing-api</artifactId>

<version>0.31.0</version>

</dependency>

By default, MicroProfile OpenTracing integrates the application with Jaeger. If you are testing our sample microservices on OpenShift, you may install Jaeger using an operator. Otherwise, we may just start it on the Docker container. The Jaeger UI is available on the address http://localhost:16686.

$ docker run -d --name jaeger \

-p 6831:6831/udp \

-p 16686:16686 \

jaegertracing/all-in-one:1.16.0

We don’t have to do anything more than adding the required dependencies to enable tracing. However, it is worth overriding the names of recorded operations. We may do it by annotating a particular method with @Traced and then by setting parameter operationName. The implementation of findByOrganizationWithEmployees method in the department-service is visible below.

public class DepartmentController {

@Inject

DepartmentRepository repository;

@Inject

EmployeeClient employeeClient;

...

@GET

@Path("/organization/{organizationId}/with-employees")

@Timeout(500)

@Retry(retryOn = TimeoutException.class)

@Traced(operationName = "findByOrganizationWithEmployees")

public List<Department> findByOrganizationWithEmployees(@PathParam("organizationId") Long organizationId) {

List<Department> departments = repository.findByOrganization(organizationId);

departments.forEach(d -> d.setEmployees(employeeClient.findByDepartment(d.getId())));

return departments;

}

}

We can also take a look at the fragment of implementation of EmployeeController.

public class EmployeeController {

@Inject

EmployeeRepository repository;

...

@GET

@Traced(operationName = "findAll")

public List<Employee> findAll() {

return repository.findAll();

}

@GET

@Path("/department/{departmentId}")

@Traced(operationName = "findByDepartment")

public List<Employee> findByDepartment(@PathParam("departmentId") Long departmentId) {

return repository.findByDepartment(departmentId);

}

}

Before running the applications we should at least set the environment variable JAEGER\_SERVICE\_NAME. It configures the name of the application visible by Jaeger. For example, before starting the employee-service application we should set the value JAEGER\_SERVICE\_NAME=employee-service. Finally, let’s send some test requests to the department-service endpoint GET departments/organization/{organizationId}/with-employees.

$ curl http://localhost:8090/departments/organization/1/with-employees

$ curl http://localhost:8090/departments/organization/2/with-employees

After sending some test requests we may go to the Jaeger UI. The picture visible below shows the history of requests processed by the method findByOrganizationWithEmployees inside department-service.