Micronaut Workshop (https://alvarosanchez.github.io/micronaut-workshop/)

Alvaro Sanchez-Mariscal – alvaro.sanchezmariscal@gmail.com

Table of Contents

(https://github.com/alvarosanchez/micronaut-workshop-java)

Introductory workshop about Micronaut (http://micronaut.io).

Software Requirements

In order to do this workshop, you need the following:

- Linux or MacOS with shell access, and the following installed:
  - `curl`.
  - `wget`.
  - `unzip`.
  - `git`.
- JDK 8.
- Docker. Please pull the following images before attending the workshop:
  - `consul`.
  - `mongo`.

Micronaut CLI

1. Install SDKMAN! (http://sdkman.io) if you haven't done so already.

2. Install Micronaut CLI:

```
$ sdk install micronaut
```

3. Ensure the CLI is installed properly:

```
$ mn --version
| Micronaut Version: 1.0.0
| JVM Version: 1.8.0_181
```

Clone this repository

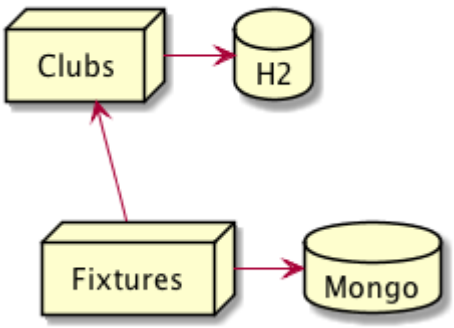
Once done, you can clone this repo:

```
git clone https://github.com/alvarosanchez/micronaut-workshop-java.git
```

You will find each exercise's template files on each `exNN` folder. Solution is always inside a `solution` folder. To highlight the actions you actually need to perform, an icon is used: 👉

Application architecture

Throughout this workshop, we will be creating a football (soccer) management system.



- `clubs` is the microservice responsible for managing clubs. It uses Hibernate as a data access layer.
- `fixtures` manages all game fixtures, storing its data in MongoDB. For the teams playing in a game, it doesn't store their full details, but rather their ID. It has a service-discovery-enabled HTTP client to fetch club details from the `clubs` microservice.

1. Getting started with Micronaut and its CLI (25 minutes)

Change to the `ex01` directory to work on this exercise

The Micronaut CLI is the recommended way to create new Micronaut projects. The CLI includes commands for generating specific categories of projects, allowing you to choose between build tools, test frameworks, and even pick the language you wish to use in your application. The CLI also provides commands for generating artifacts such as controllers, client interfaces, and serverless functions.

The `create-app` command is the starting point for creating Micronaut applications. The CLI is based on the concept of **profiles**. A profile consist of a project template (or skeleton), optional features, and profile-specific commands. Commands from a profile typically are specific to the profile application type; for example, the `service` profile

(designed for creation of microservice applications) provides the `create-controller` and `create-c` commands.

1.1. Listing profiles (3 minutes)

👉 You can list the available profiles with the `list-profiles` command:

```
$ mn list-profiles
| Available Profiles
--------------------
  cli           The cli profile
  federation    The federation profile
  function-aws  The function profile for AWS Lambda
  kafka         The Kafka messaging profile
  profile       A profile for creating new Micronaut profiles
  service       The service profile
```

Applications generated from a profile can be personalised with **features**. A feature further customises the newly created project by adding additional dependencies to the build, more files to the project skeleton, etc.

1.2. Getting information about a profile (2 minutes)

👉 To see all the features of a profile, you can use the `profile-info` command:

```
$ mn profile-info service
| Profile: service
--------------------
The service profile

| Provided Commands:
--------------------
  create-bean           Creates a singleton bean
  create-client         Creates a client interface
  create-controller     Creates a controller and associated test
  create-job            Creates a job with scheduled method
  create-websocket-client  Creates a Websocket client
  create-websocket-server  Creates a Websocket server
  help                  Prints help information for a specific command

| Provided Features:
--------------------
  annotation-api        Adds Java annotation API
  cassandra             Adds support for Cassandra in the application
  config-consul         Adds support for Distributed Configuration with Consul (https://www.consul.io)
  discovery-consul      Adds support for Service Discovery with Consul (https://www.consul.io)
  discovery-eureka      Adds support for Service Discovery with Eureka
  graal-native-image    Allows Building a Native Image
  groovy                Creates a Groovy application
  hibernate-gorm        Adds support for GORM persistence framework
  hibernate-jpa         Adds support for Hibernate/JPA
  http-client           Adds support for creating HTTP clients
  http-server           Adds support for running a Netty server
  java                  Creates a Java application
  jdbc-dbcp             Configures SQL DataSource instances using Commons DBCP
  jdbc-hikari           Configures SQL DataSource instances using Hikari Connection Pool
  jdbc-tomcat           Configures SQL DataSource instances using Tomcat Connection Pool
  jib                   Adds support for Jib builds
  jrebel                Adds support for class reloading with JRebel (requires separate JRebel installation)
  junit                 Adds support for the JUnit testing framework
  kafka                 Adds support for Kafka
  kafka-streams         Adds support for Kafka Streams
  kotlin                Creates a Kotlin application
  management            Adds support for management endpoints
  micrometer            Adds support for Micrometer metrics
  micrometer-atlas      Adds support for Micrometer metrics (w/ Atlas reporter)
  micrometer-graphite   Adds support for Micrometer metrics (w/ Graphite reporter)
  micrometer-prometheus  Adds support for Micrometer metrics (w/ Prometheus reporter)
  micrometer-statsd     Adds support for Micrometer metrics (w/ Statsd reporter)
  mongo-gorm            Configures GORM for MongoDB for Groovy applications
  mongo-reactive        Adds support for the Mongo Reactive Streams Driver
  neo4j-bolt            Adds support for the Neo4j Bolt Driver
  neo4j-gorm            Configures GORM for Neo4j for Groovy applications
  netflix-archaius      Adds support for Netflix Archaius in the application
  netflix-hystrix       Adds support for Netflix Hystrix in the application
  netflix-ribbon        Adds support for Netflix Ribbon in the application
  picocli               Adds support for command line parsing (http://picocli.info)
  postgres-reactive     Adds support for the Reactive Postgres driver in the application
  rabbitmq              Adds support for RabbitMQ in the application
  redis-lettuce         Configures the Lettuce driver for Redis
  security-jwt          Adds support for JWT (JSON Web Token) based Authentication
  security-session      Adds support for Session based Authentication
  spek                  Adds support for the Spek testing framewokr
  spock                 Adds support for the Spock testing framework
  springloaded          Adds support for class reloading with Spring-Loaded
  swagger-groovy        Configures Swagger (OpenAPI) Integration for Groovy
  swagger-java          Configures Swagger (OpenAPI) Integration for Java
  swagger-kotlin        Configures Swagger (OpenAPI) Integration for Kotlin
  tracing-jaeger        Adds support for distributed tracing with Jaeger (https://www.jaegertracing.io)
  tracing-zipkin        Adds support for distributed tracing with Zipkin (https://zipkin.io)
```

1.3. Creating and running a *hello galaxy* (15 minutes)

As explained avobe, the `create-app` command can be used to create new projects. It accepts some flags:

Table 1. Create-App Flags

Flag	Description	Example
`build`	Build tool (one of `gradle`, `maven` - default is `gradle`)	`--build maven`
`profile`	Profile to use for the project (default is `service`)	`--profile function-aws`
`features`	Features to use for the project, comma-separated	`--features security-jwt,mongo-gorm`
`inplace`	If present, generates the project in the current directory (project name is optional if this flag is set)	`--inplace`

👉 Let's create a *hello galaxy* project:

```
$ mn create-app hello-galaxy
| Generating Java project...
| Application created at /Users/alvarosanchez/hello-galaxy
```

👉 Now, move into the generated `hello-galaxy` folder and let's create a controller:

```
$ mn create-controller hello
| Rendered template Controller.java to destination src/main/java/hello/galaxy/HelloController.java
| Rendered template ControllerTest.java to destination src/test/java/hello/galaxy/HelloControllerTest.java
```

👉 Open the generated `HelloController.java` with your favourite IDE and make it return "Hello Galaxy!":

JAVA

```
@Get("/")
String index() {
    return "Hello Galaxy!";
}
```

👉 Now, run the application:

```
$ ./gradlew run
```

You will see a line similar to the following once the application has started

```
14:40:01.187 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 957ms. Server Running:
http://localhost:8080
```

👉 Then, on another shell, make a request to your service:

```
$ curl 0:8080/hello
Hello Galaxy!
```

1.4. Write an automated test (5 minutes)

While testing manually is acceptable in some situations, going forward it is better to have automated tests ercise our applications. Fortunately, Micronaut makes testing super easy!

Micronaut applications can be tested with any testing framework, because `io.micronaut.context.ApplicationContext` is capable of spinning up embedded instances quite easily. The CLI adds support for using JUnit, Spock and Spek.

For example, in *plain* JUnit 4, this how a end-to-end functional test looks like:

JAVA

```
public class HelloWorldTest {
    private static EmbeddedServer server;
    private static HttpClient client;

    @BeforeClass
    public static void setupServer() {
        server = ApplicationContext.run(EmbeddedServer.class);
        client = server.getApplicationContext().createBean(HttpClient.class, server.getURL());
    }

    @AfterClass
    public static void stopServer() {
        if(server != null) { server.stop(); }
        if(client != null) { client.stop(); }
    }

    @Test
    public void testHelloWorkd() throws Exception {
        String body = client.toBlocking().retrieve("/hello");
        assertEquals(body, "Hello Galaxy!");
    }
}
```

In addition to that, if you are using JUnit 5 or Spock, there is special support that allows to remove most of the boilerplate about starting/stopping server and injecting beans. Check the Micronaut Test (https://micronaut-projects.github.io/micronaut-test/latest/guide/index.html) project for more information.

👉 Micronaut Test is not (yet) included in the projects generated by the CLI, so let's add it. Modify `build.gradle` to remove `"junit:junit:4.12"` and `"org.hamcrest:hamcrest-all:1.3"`, and include:

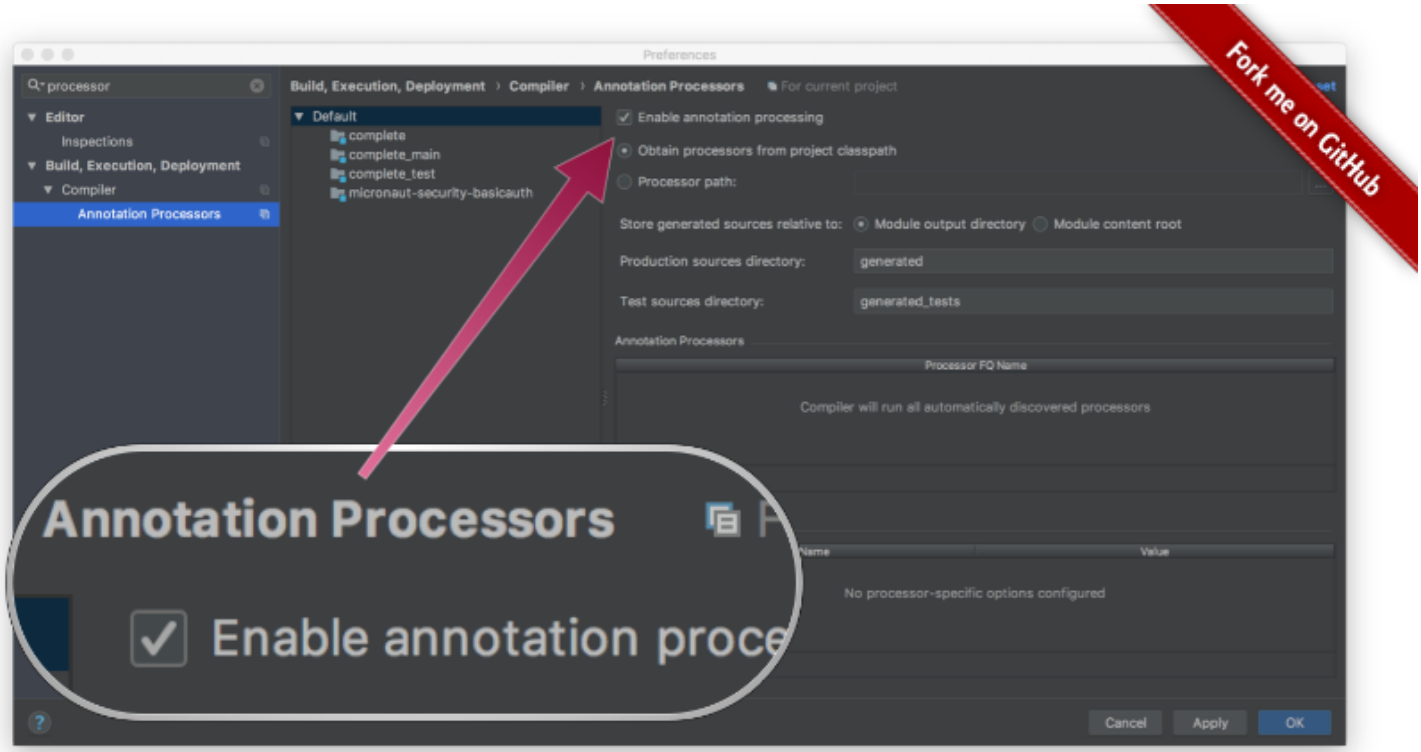
- `"org.junit.jupiter:junit-jupiter-engine:5.3.1"`
- `"io.micronaut.test:micronaut-test-junit5:1.0.0.RC2"`

Also, tell Gradle to use JUnit 5 Platform and display output:

JAVA

```
test {
    useJUnitPlatform()
    testLogging {
        showStandardStreams = true
    }
}
```

We will use Gradle to run the tests, however, if you want to run them from your IDE, make sure you enable annotation processors. For example, in Intellij IDEA:



👉 Now, change the generated `src/test/java/hello/galaxy/HelloControllerTest.java` to look like this:

JAVA

```
package hello.galaxy;

import io.micronaut.http.client.RxHttpClient;
import io.micronaut.runtime.server.EmbeddedServer;
import io.micronaut.test.annotation.MicronautTest;
import org.junit.jupiter.api.Test;

import javax.inject.Inject;

import static org.junit.jupiter.api.Assertions.assertEquals;

@MicronautTest
public class HelloControllerTest {

    @Inject
    private EmbeddedServer embeddedServer;

    @Test
    void testHelloGalaxy() {
        try(RxHttpClient client = embeddedServer.getApplicationContext().createBean(RxHttpClient.class,
embeddedServer.getURL())) {
            assertEquals("Hello Galaxy!", client.toBlocking().exchange("/hello", String.class).body());
        }
    }

}
```

As you can see, is much shorter than the previous version

👉 Then, run the tests:

```
./gradlew test
```

Once finished, you should see an output similar to:

```
BUILD SUCCESSFUL in 5s
```


2. Creating the Clubs microservice (70 minutes)

Change to the `ex02` directory to work on this exercise.

👉 In this exercise we are creating the `clubs` microservice. Start with:

```
mn create-app --features hibernate-jpa clubs
```

And open it in your IDE.

The `hibernate-jpa` will bring to the newly created project:

- The required build dependencies to have Hibernate, a Tomcat-based JDBC connection pool and an H2 in-memory database (`build.gradle`).
- The data source configuration to use such H2 database (`src/main/resources/application.yml`).

👉 Check yourself the above files to see how it is configured.

👉 Also, before going any further, repeat the steps in Exercise 1 to include Micronaut Test in this project

2.1. JPA layer (15 minutes)

Our model will reside in the `clubs.domain` package. We need to configure JPA to search for entities in this package.

👉 Change `jpa` section of `src/main/resources/application.yml` so that it looks like:

YAML

```
jpa:
  default:
    packages-to-scan:
    - 'clubs.domain'
    properties:
      hibernate:
        hbm2ddl:
          auto: update
        show_sql: true
```

👉 Let's define first a `Club` entity under `src/main/java/clubs/domain/Club.java` with 2 string attributes: `name` (mandatory) and `stadium` (optional).

👉 Next, define repository named `ClubRepository` as an interface with the following operations:

JAVA

```
Long count();
Club save(@NotBlank String name, String stadium);
List<Club> findAll();
Optional<Club> find(@NotNull Long id);
```

Now, let's write the implementation using JPA:

JAVA

```java
@Singleton
public class ClubRepositoryImpl implements ClubRepository {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    @Transactional(readOnly = true)
    public Long count() {
        return entityManager.createQuery("select count(c) from Club c", Long.class).getSingleResult();
    }

    @Override
    @Transactional
    public Club save(@NotBlank String name, String stadium) {
        Club club = new Club(name);
        club.setStadium(stadium);
        entityManager.persist(club);
        return club;
    }

    @Override
    @Transactional(readOnly = true)
    public List<Club> findAll() {
        return entityManager.createQuery("select c from Club c", Club.class).getResultList();
    }

    @Override
    @Transactional(readOnly = true)
    public Optional<Club> find(@NotNull Long id) {
        return Optional.ofNullable(entityManager.find(Club.class, id));
    }

}
```

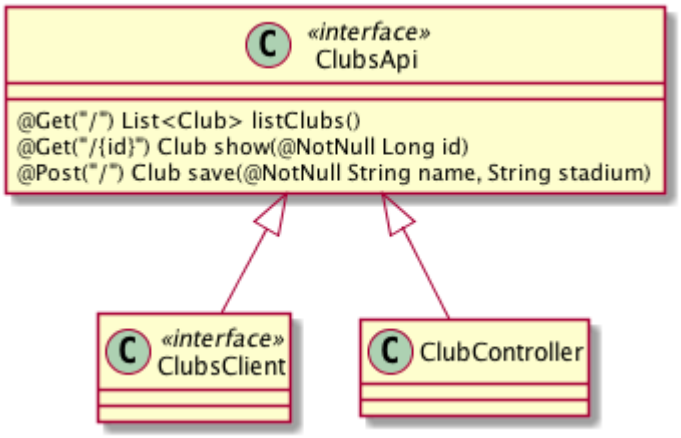
👉 Now, let's write a test for our implementation:

JAVA

```java
@MicronautTest
public class ClubRepositoryImplTest {

    @Inject
    ClubRepository repository;

    @Test
    void testCrudOperations() {
        assertEquals(0L, repository.count().longValue());

        repository.save("Real Madrid", "Santiago Bernabeu");
        repository.save("FC Barcelona", "Camp Nou");
        assertEquals(2L, repository.count().longValue());

        List<Club> allClubs = repository.findAll();
        assertEquals(2, allClubs.size());

        Club realMadrid = repository.find(1L).get();
        assertSame("Santiago Bernabeu", realMadrid.getStadium());

        assertEquals(Optional.empty(), repository.find(27L));
    }

}
```

2.2. REST API (30 minutes)

Micronaut helps you writing both the client and server sides of a REST API. In this service, we are going to create the following:



👉 Create the `ClubsApi` interface, annotating its methods with `io.micronaut.http.annotation.Get` or `io.micronaut.http.annotation.Post` as described in the diagram.

👉 Then, create `ClubsClient` by simply extending from `ClubsApi`. Annotate the interface with `io.micronaut.http.client.Client("/")`.

👉 Finally, implement the controller `ClubController`. Annotate the class with `io.micronaut.http.annotation.Controller("/")`, matching the path specified on `ClubsClient`. Use `ClubRepository` to implement the actions by declaring a constructor dependency on it.

The controller actions need to be annotated with `@Get` / `@Post` again.

👉 Finally, configure `logback.xml` to see some relevant output

XML

```
<configuration>

    <!-- Default settings. Omitted for brevity ... -->

    <logger name="clubs" level="DEBUG"/>  1
    <logger name="io.micronaut.http.client" level="TRACE"/>  2

</configuration>
```

1 Debug level for our code

2 This allows to see the HTTP request and responses from the HTTP clients.

👉 Once you have it, write an end-to-end test:

JAVA

```java
@MicronautTest
public class ClubControllerTest {

    @Inject
    ClubsClient client;

    @Test
    void testGetOneClub() {
        Club realMadrid = client.save("Real Madrid", "Santiago Bernabeu");

        Club response = client.show(realMadrid.getId());
        assertEquals("Santiago Bernabeu", response.getStadium());
    }

    @Test
    void testFindAllClubs() {
        client.save("Real Madrid", "Santiago Bernabeu");
        client.save("FC Barcelona", "Camp Nou");

        assertEquals(2, client.listClubs().size());
    }
}
```

2.3. Load some data for production (15 minutes)

During our tests, we have been seeding test data on demand, as it is a good practise to isolate test data from test to test. However, for production, we want some data loaded

👉 Let's create a bean to load some data. Run:

```
mn create-bean dataLoader
```

👉 Change it to look like:

JAVA

```java
@Singleton
@Requires(notEnv = Environment.TEST)
public class DataLoader implements ApplicationEventListener<ServerStartupEvent> {

    private ClubRepository repository;

    public DataLoader(ClubRepository repository) {
        this.repository = repository;
    }

    @Override
    public void onApplicationEvent(ServerStartupEvent event) {
        if (repository.count() == 0) {
            repository.save("Real Madrid CF", "Santiago Bernabeu");
            repository.save("FC Barcelona", "Camp Nou");
            repository.save("CD Leganes", "Butarque");
            repository.save("Getafe CF", "Coliseum");
        }
    }
}
```

👉 Now, run the application:

```
./gradlew run
```

👉 And make a request to `0:8080/` to see the results:

2.4. Register the service in Consul (10 minutes)

We want the `clubs` microservice to be discoverable by the `fixtures` service. So we will enable Micronaut's support for Consul support for service discovery.

👉 First, add the neccessary dependency in `build.gradle`:

```
compile "io.micronaut:micronaut-discovery-client"
```
JAVA

👉 Then, change `src/main/resources/application.yml` to define the Consul configuration:

```
---
consul:
  client:
    registration:
      enabled: true
  defaultZone: "${CONSUL_HOST:localhost}:${CONSUL_PORT:8500}"
```
YAML

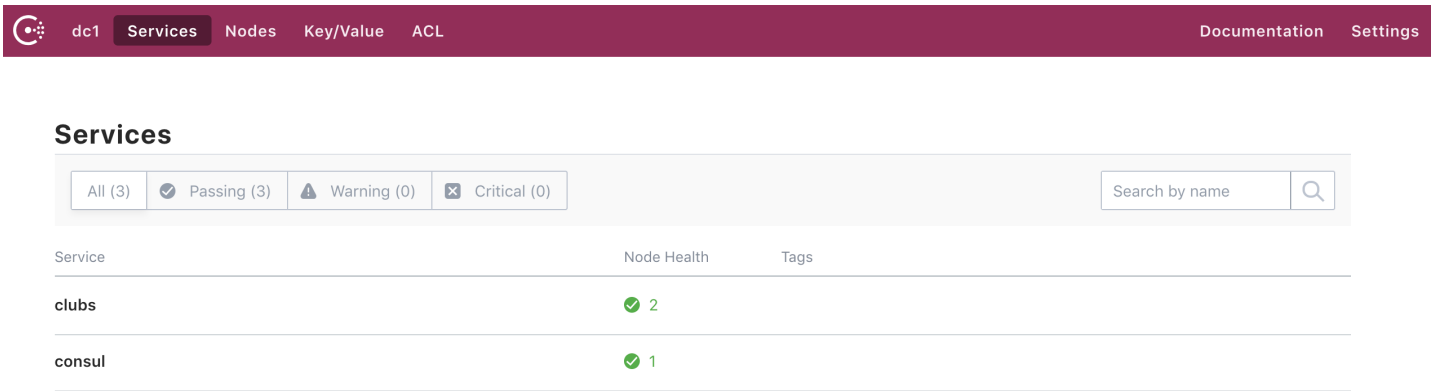
👉 Finally, run a Consul instance with Docker:

```
$ docker run -d --name=dev-consul -e CONSUL_BIND_INTERFACE=eth0 -e CONSUL_UI_BETA=true -p 8500:8500 consul
```

👉 Now, if you run the application, you will see it registers with Consul at startup:

```
$ ./gradlew run
...
04:20:09.501 [nioEventLoopGroup-1-3] INFO  i.m.d.registration.AutoRegistration - Registered service [clubs]
with Consul
...
```

👉 If you go the Consul UI (http://localhost:8500/), you can see it shows as registered:



👉 You can run yet another instance of `clubs` on a different shell, and see it registered. We will use them both with Micronaut's load-balanced HTTP client in the next exercise.

3. Creating the Fixtures microservice (70 minutes)

Change to the `ex03` directory to work on this exercise.

👉 In this exercise we are creating the `fixtures` microservice:

```
mn create-app --features=mongo-reactive,discovert-consul fixtures
```


Once again, follow the steps of exercise 1 to add Micronaut Test to this project. Also, remove the `de.flapdoodle.embed.mongo` dependency, as we are using a Dockerized MongoDB instance.

3.1. Data layer (35 minutes)

👉 First of all, run MongoDB with Docker:

```
$ docker run -d --name=dev-mongo -p 27017:27017 mongo
```

👉 Then, create the `Fixture` domain class with the following properties:

JAVA

```
@BsonId
private ObjectId id;

private Long homeClubId;
private Long awayClubId;

private Short homeScore;
private Short awayScore;

private Date date;
```

As you can see, we are only storing club's ids. When rendering fixture details, we will use Micronaut's HTTP client to fetch details from the `clubs` microservice.

👉 We also need a constructor with annotations that allow `Fixture` instances to be marshalled and unmarshalled to/from JSON and as a MongoDB document:

JAVA

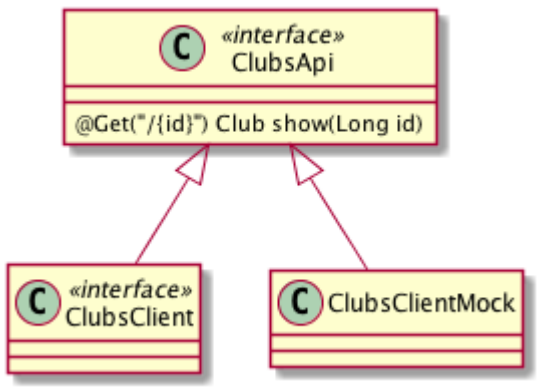
```
@BsonCreator
@JsonCreator
public Fixture(@BsonProperty("homeClubId") @JsonProperty("homeClubId") Long homeClubId,
               @BsonProperty("awayClubId") @JsonProperty("awayClubId") Long awayClubId,
               @BsonProperty("homeScore") @JsonProperty("homeScore") Short homeScore,
               @BsonProperty("awayScore") @JsonProperty("awayScore") Short awayScore,
               @BsonProperty("date") @JsonProperty("date") Date date) {
    this.homeClubId = homeClubId;
    this.awayClubId = awayClubId;
    this.homeScore = homeScore;
    this.awayScore = awayScore;
    this.date = date;
}
```

Be sure to add all the getter and setters as well.

👉 The next thing we need is an HTTP client for the `clubs` microservice. Create one with:

```
$ mn create-client clubs
```

Before actually mapping any endpoint, we are going to create the following hierarchy:



- `ClubsApi` is the interface that contains the client endpoint mappings.
- `ClubsClient` is the production client, is annotated with `@Client` and simply extends from `ClubsApi`.
- `ClubsClientMock` is a mocking client (resides within `src/test/java`), is annotated with `@Fallback`, and implements `ClubsApi` by returning hardcoded instances.

This is how `ClubsApi` looks like:

JAVA
```
public interface ClubsApi {

    @Get("/{id}")
    Maybe<Club> findTeam(Long id);

}
```

We are using a reactive type in the HTTP client response, so that is a hint for Micronaut to make it non-blocking.

Then, the production client:

JAVA
```
@Client("clubs")     1
public interface ClubsClient extends ClubsApi {}
```

1 `"clubs"` is the Consul name for the Clubs microservice (which registers itself with the `micronaut.application.name` property).

Finally, the mocking client:

JAVA
```
@Fallback
public class ClubsClientMock implements ClubsApi {
    @Override
    public Maybe<Club> findTeam(Long id) {
        if (id == 1) {
            return Maybe.just(new Club("CD Leganes", "Butarque"));
        } else {
            return Maybe.just(new Club("Getafe CF", "Coliseum"));
        }
    }
}
```

👉 We also need a `Club` POJO to capture the JSON response from `clubs`. Define it with 2 string fields: `name` and `stadium`, and its constructor, getters, etc.

👉 Now let's create a repository for `Fixture`. Following the same convention as in the previous exercise, begin with an interface:

JAVA

```java
public interface FixtureRepository {

    Single<Fixture> save(@Valid Fixture fixture);
    Flowable<Fixture> findAll();
    Single<Long> count();

}
```

👉 Then, the implementation:

JAVA

```java
@Singleton
public class FixtureRepositoryImpl implements FixtureRepository {

    public static final String DB_NAME = "fixturesDb";
    public static final String COLLECTION_NAME = "fixtures";

    private MongoClient mongoClient;

    public FixtureRepositoryImpl(MongoClient mongoClient) {
        this.mongoClient = mongoClient;
    }

    @Override
    public Single<Fixture> save(@Valid Fixture fixture) {
        return Single.fromPublisher(getCollection().insertOne(fixture)).map(success -> fixture);
    }

    @Override
    public Flowable<Fixture> findAll() {
        return Flowable.fromPublisher(getCollection().find());
    }

    @Override
    public Single<Long> count() {
        return Single.fromPublisher(getCollection().count());
    }

    private MongoCollection<Fixture> getCollection() {
        return mongoClient.getDatabase(DB_NAME).getCollection(COLLECTION_NAME, Fixture.class);
    }
}
```

👉 And a test:

JAVA

```
@MicronautTest
public class FixtureRepositoryImplTest {

    @Inject
    FixtureRepository repository;

    @Inject
    MongoClient mongoClient;

    @BeforeEach
    void cleanup() {
        Flowable.fromPublisher(mongoClient.getDatabase(DB_NAME).getCollection(COLLECTION_NAME,
Fixture.class).deleteMany(new Document())).blockingFirst();
    }

    @Test
    void testCrud() {
        assertEquals(0, repository.count().blockingGet().longValue());

        repository.save(new Fixture(1L, 2L, (short)5, (short)0, new Date())).blockingGet();
        repository.save(new Fixture(3L, 4L, (short)5, (short)0, new Date())).blockingGet();
        assertEquals(2, repository.count().blockingGet().longValue());
        assertEquals(2, repository.findAll().toList().blockingGet().size());
    }
}
```

Make sure it passes.

3.2. REST API (35 minutes)

👉 Let's create a controller for displaying fixtures:

```
$ mn create-controller fixture
```

As it was said earlier, our `Fixture` class doesn't store club names, but their id's (with the intention of having this microservice call the other). Therefore, we need a DTO class to represent what our JSON response is going to look like.

👉 Create a POJO named `FixtureResponse` with the following attributes:

JAVA

```
private String homeClubName;
private String awayClubName;

private String stadium;

private Short homeScore;
private Short awayScore;

private Date date;
```

Now we need a service that transforms a `Fixture` into a `FixtureResponse`. To do so, it need to make 2 HTTP calls to the `clubs` microservice, to get the name of each clubs. It will use `ClubsClient` for that.

👉 Create a `FixtureService` like this:

JAVA

```java
@Singleton
public class FixtureService {

    private ClubsClient clubsClient;

    public FixtureService(ClubsClient clubsClient) {
        this.clubsClient = clubsClient;
    }

    public Maybe<FixtureResponse> toResponse(Fixture fixture) {
        return Maybe.zip(
                clubsClient.findTeam(fixture.getHomeClubId()),
                clubsClient.findTeam(fixture.getAwayClubId()),
                (homeClub, awayClub) -> new FixtureResponse(homeClub.getName(),
                                                            awayClub.getName(),
                                                            homeClub.getStadium(),
                                                            fixture.getHomeScore(),
                                                            fixture.getAwayScore(),
                                                            fixture.getDate())
        );
    }
}
```

👉 And write a test for it:

JAVA

```java
@MicronautTest
public class FixtureServiceTest {

    @Inject
    FixtureService fixtureService;

    @Inject
    FixtureRepository repository;

    @Inject
    MongoClient mongoClient;

    @BeforeEach
    void cleanup() {
        Flowable.fromPublisher(mongoClient.getDatabase(DB_NAME).getCollection(COLLECTION_NAME,
Fixture.class).deleteMany(new Document())).blockingFirst();
    }

    @Test
    void testToResponse(){
        Fixture fixture = repository.save(new Fixture(1L, 2L, (short)5, (short)0, new Date())).blockingGet();
        FixtureResponse response = fixtureService.toResponse(fixture).blockingGet();

        assertEquals("CD Leganes", response.getHomeClubName());
        assertEquals("Getafe CF", response.getAwayClubName());
    }

}
```

Finally, we need the REST controller that connect the dots.

👉 Create a `FixtureController` that uses `FixtureRepository` and `FixtureService` as collaborators to produce a `Flowable<FixtureResponse>` response:

JAVA

```java
@Get("/")
public Flowable<FixtureResponse> list() {
    return fixtureRepository.findAll().flatMapMaybe(fixture -> fixtureService.toResponse(fixture));
}
```

3.3. Load some data and run the application (10 minutes)

👉 Similarly to the previous exercise, seed the application with some data.

Also, we need to set the `micronaut.server.port` configuration property a value other than 8080, otherwise, we won't be able to run both services.

👉 In `application.yml`, set `micronaut.server.port` to `80801`

👉 Now, run the application:

```
./gradlew run
```

If you make a request to the default controller, and the `clubs` microservice is not running, you will see an error:

```
{"message":"Internal Server Error: No available services for ID: clubs"}
```

👉 Now, run the `clubs` service on a different terminal, and try the request again.

Last updated 2018-11-12 10:56:04 CET