TABLE OF CONTENTS

KUBERNETES AND THE MICRONAUT FRAMEWORK

Use Kubernetes to deploy your Micronaut applications.

Authors: Nemanja Mikic

Micronaut Version: 3.7.3

1. Getting Started

In this guide, we will create three microservices, build containerized versions and deploy them with **Kubernetes**. We will use Kubernetes Service discovery and Distributed configuration to wire up our microservices.

> Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

You will discover how the Micronaut framework eases Kubernetes integration.

2. What you will need

To complete this guide, you will need the following:

- Some time on your hands
- A decent text editor or IDE
- JDK 17 or greater installed with `JAVA_HOME` configured appropriately
- **Docker**.
- Local Kubernetes cluster. We will use **Minikube** in this guide.

3. Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the **completed example**.

- **Download** and unzip the source

4. Writing the Apps

Let’s describe the microservices you will build through the guide.

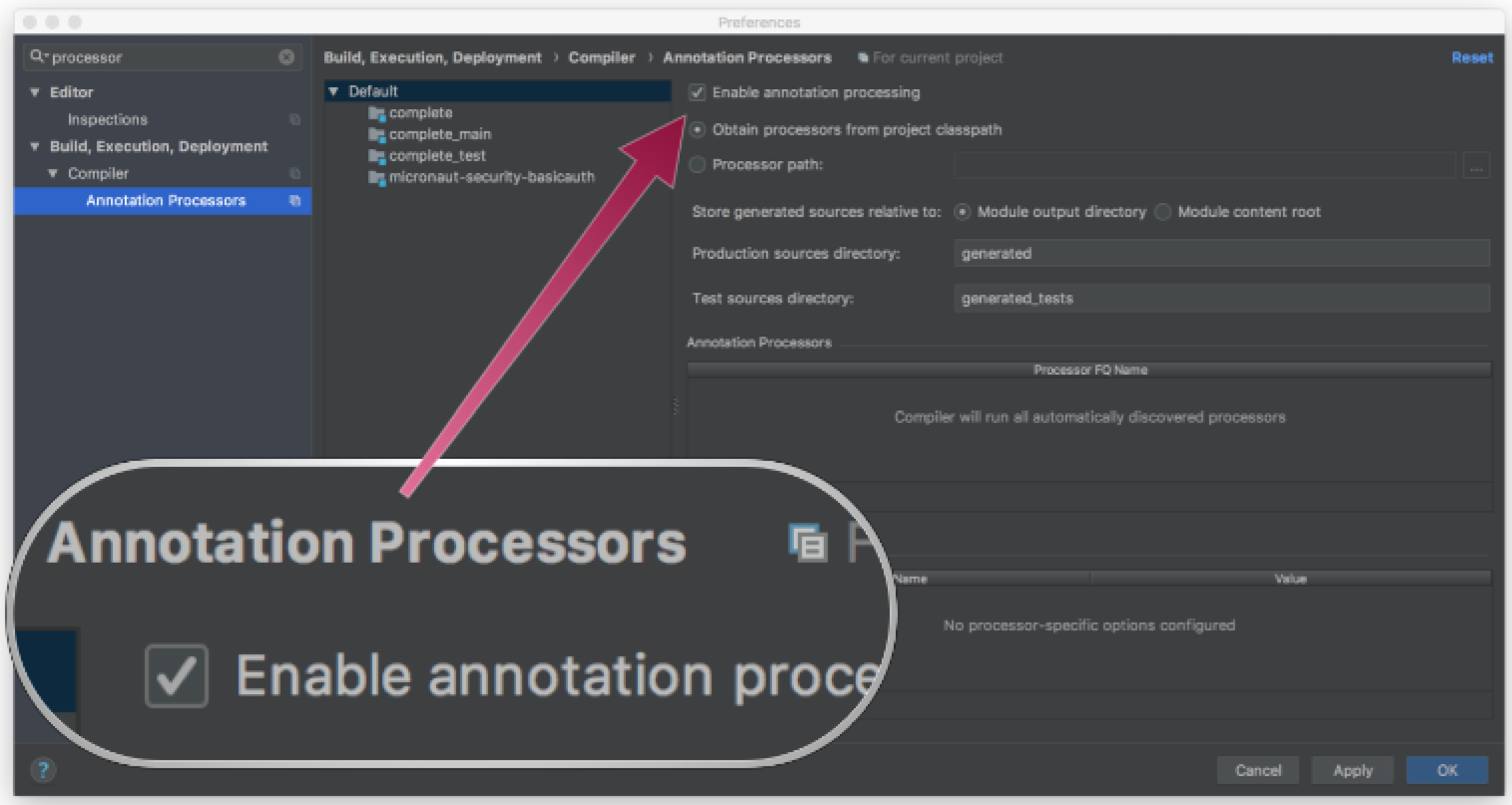
- `users` - This microservice contains all customers data that can place orders on items, also a new customer can be created. Microservice requires Basic authentication to access it.
- `orders` - This microservice contains all created orders by customers and available items that customers can use to place new item orders. Also in this microservice a new item orders can be created. Microservice requires Basic authentication to access it.
- `api` - It’s a gateway to both `orders` and `users` services. It combines results from both services and checks data when customers create a new item order.

Initially we will hard-code the addresses where the different services are in the `api` service. Additionally, we will hard-code credentials (username and password) into every microservice configuration that are required for Basic authentication.

In the second part of this guide we will use a Kubernetes discovery service and Kubernetes config maps to dynamically resolve client urls and get authentication credentials. Services register when they start up and resolve placeholders inside microservice configuration by calling Kubernetes API.

4.1. Enable annotation Processing

If you use Java or Kotlin and IntelliJ IDEA, make sure to enable annotation processing.



✓ Enable annotation processing

[● Obtain processors from project classpath]

○ Processor path:

Store generated sources relative to: [● Module output directory] ○ Module content root

Production sources directory: generated

Test sources directory: generated_tests

Annotation Processors

Processor FQ Name

Compiler will run all automatically discovered processors

Name | Value

No processor-specific options configured

4.2. Users Microservice

Create the `users` microservice using the [Micronaut Command Line Interface](#) or with [Micronaut Launch](#).

```
mn create-app --features=discovery-kubernetes,management,security,serialization-jackson,kubernetes
example.micronaut.users --build=gradle --lang=java
```

If you don't specify the `--build` argument, Gradle is used as the build tool.
If you don't specify the `--lang` argument, Java is used as the language.

If you use Micronaut Launch, select Micronaut Application as application type and add the `discovery-kubernetes`, `management`, `security`, `serialization-jackson` and `kubernetes` features.

The previous command creates a directory named `users` and a Micronaut application inside it with default package `example.micronaut`.

If you have an existing Micronaut application and want to add the functionality described here, you can view the dependency and configuration changes from the specified features and apply those changes to your application.

Create package with name `controllers` and create `UsersController` class to handle incoming HTTP requests into the `users` microservice:

users/src/main/java/example/micronaut/controllers/UsersController.java

```java
package example.micronaut.controllers;

import example.micronaut.models.User;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.exceptions.HttpStatusException;
import io.micronaut.security.annotation.Secured;
import io.micronaut.security.rules.SecurityRule;
import io.micronaut.validation.Validated;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

@Controller("/users") 1
@Secured(SecurityRule.IS_AUTHENTICATED) 2
@Validated
public class UsersController {
    List<User> persons = new ArrayList<>();

    @Post 3
    public User add(@Body @Valid User user) {
        Optional<User> existingUser = findByUsername(user.username());

        if (existingUser.isPresent()) {
            throw new HttpStatusException(HttpStatus.CONFLICT, "User with provided username already exists");
        }

        User newUser = new User(persons.size() + 1, user.firstName(), user.lastName(), user.username());
        persons.add(newUser);
        return newUser;
    }

    @Get("/{id}") 4
    public User findById(@NotNull Integer id) {
        return persons.stream()
                .filter(it -> it.id().equals(id))
                .findFirst().orElse(null);
    }

    @Get 5
    public List<User> getUsers() {
        return persons;
    }

    public Optional<User> findByUsername(@NotNull String username) {
        return persons.stream()
                .filter(it -> it.username().equals(username))
                .findFirst();
    }

}
```

1. The class is defined as a controller with the @Controller annotation mapped to the path `/users`.
2. Annotate with `io.micronaut.security.Secured` to configure secured access. The `isAuthenticated()` expression will allow access only to authenticated users.
3. The @Post annotation maps the `add` method to an HTTP POST request on `/users`.
4. The @Get annotation maps the `findById` method to an HTTP GET request on `/users/{id}`.
5. The @Get annotation maps the `getUsers` method to an HTTP GET request on `/users`.

Create package with name `models` where we will put our data beans.

The previous `UsersController` controller uses `User` object to represent customer. Create the `User` record

users/src/main/java/example/micronaut/models/User.java

Copy

```java
package example.micronaut.models;

import com.fasterxml.jackson.annotation.JsonProperty;
import io.micronaut.core.annotation.Nullable;
import io.micronaut.serde.annotation.Serdeable;

import javax.validation.constraints.Max;
import javax.validation.constraints.NotBlank;

@Serdeable (1)
public record User(
        @Nullable @Max(10000) Integer id, (2)
        @NotBlank @JsonProperty("first_name") String firstName,
        @NotBlank @JsonProperty("last_name") String lastName,
        String username
) {
}
```

1. Declare the `@Serdeable` annotation at the type level in your source code to allow the type to be serialized or deserialized.
2. ID will be generated by application.

Create package with name `auth` where you will check basic authentication credentials.

The `Credentials` class will load and store credentials (username and password) from configuration.

users/src/main/java/example/micronaut/auth/Credentials.java

Copy

```java
package example.micronaut.auth;

import io.micronaut.context.annotation.ConfigurationProperties;

@ConfigurationProperties("authentication-credentials") (1)
public record Credentials (
        String username,
        String password
) {
}
```

1. The `@ConfigurationProperties` annotation takes the configuration prefix.

The `CredentialsChecker` class, as name says, it will check if provided credentials inside request’s Authorization header are the same with ones that are stored inside `Credentials` class that we created above.

users/src/main/java/example/micronaut/auth/CredentialsChecker.java

Copy

```java
package example.micronaut.auth;

import io.micronaut.core.annotation.Nullable;
import io.micronaut.http.HttpRequest;
import io.micronaut.security.authentication.AuthenticationProvider;
import io.micronaut.security.authentication.AuthenticationRequest;
import io.micronaut.security.authentication.AuthenticationResponse;
import jakarta.inject.Singleton;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;

@Singleton (1)
public class CredentialsChecker implements AuthenticationProvider {

    private final Credentials credentials;

    public CredentialsChecker(Credentials credentials) {
        this.credentials = credentials;
    }

    @Override
    public Publisher<AuthenticationResponse> authenticate(@Nullable HttpRequest<?> httpRequest,
                                                          AuthenticationRequest<?, ?> authenticationRequest) {
        return Mono.<AuthenticationResponse>create(emitter -> {
            if ( authenticationRequest.getIdentity().equals(credentials.username()) &&
                    authenticationRequest.getSecret().equals(credentials.password()) ) {
                emitter.success(AuthenticationResponse.success((String) authenticationRequest.getIdentity()));
            } else {
                emitter.error(AuthenticationResponse.exception());
            }
        });
    }
}
```

1. Use `jakarta.inject.Singleton` to designate a class as a singleton.

4.2.1. WRITE TESTS TO VERIFY APPLICATION LOGIC

Create the `UsersClient` Micronaut HTTP inline client for testing:

users/src/test/java/example/micronaut/UsersClient.java

Copy

```java
package example.micronaut;

import example.micronaut.models.User;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Header;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.client.annotation.Client;

import java.util.List;

@Client("/") (1)
public interface UsersClient {

    @Get("/users/{id}")
    User getById(@Header String authorization, Integer id);

    @Post("/users")
    User createUser(@Header String authorization, @Body User user);

    @Get("/users")
    List<User> getUsers(@Header String authorization);
}
```

1. Use `@Client` to use declarative HTTP Clients. You can annotate interfaces or abstract classes. You can use the `id` member to provide a service identifier or specify the URL directly as the annotation's value.

`HealthTest` checks if there is `/health` endpoint exposed that is needed for service discovery.

users/src/test/java/example/micronaut/HealthTest.java

Copy

```java
package example.micronaut;

import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.client.annotation.Client;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;

import jakarta.inject.Inject;

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

@MicronautTest (1)
public class HealthTest {

    @Inject
    @Client("/")
    HttpClient client; (2)

    @Test
    public void healthEndpointExposed() {
        HttpStatus status = client.toBlocking().retrieve(HttpRequest.GET("/health"), HttpStatus.class);
        assertEquals(HttpStatus.OK, status);
    }
}
```

1. Annotate the class with `@MicronautTest` so the Micronaut framework will initialize the application context and the embedded server. More info.
2. Inject the `HttpClient` bean and point it to the embedded server.

`UsersControllerTest` tests endpoints inside the `UserController`.

users/src/test/java/example/micronaut/UsersControllerTest.java

Copy

```java
package example.micronaut;

import example.micronaut.auth.Credentials;
import example.micronaut.models.User;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.client.exceptions.HttpClientException;
import io.micronaut.http.client.exceptions.HttpClientResponseException;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import jakarta.inject.Inject;
import org.junit.jupiter.api.Test;

import java.util.Base64;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

@MicronautTest (1)
public class UsersControllerTest {

    @Inject
    UsersClient usersClient;

    @Inject
    Credentials credentials;

    @Test
    void testUnauthorized() {
        HttpClientException exception = assertThrows(HttpClientException.class, () -> usersClient.getUsers(""));
        assertTrue(exception.getMessage().contains("Unauthorized"));
    }

    @Test
    void getUserThatDoesntExists() {
        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());
        User retriedUser = usersClient.getById(authHeader, 100);
        assertNull(retriedUser);
    }

    @Test
    void multipleUserInteraction() {
        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());

        String firstName = "firstName";
        String lastName = "lastName";
        String username = "username";

        User user = new User(0 ,firstName, lastName, username);

        User createdUser = usersClient.createUser(authHeader, user);

        assertEquals(firstName, createdUser.firstName());
        assertEquals(lastName, createdUser.lastName());
        assertEquals(username, createdUser.username());
        assertNotNull(createdUser.id());

        User retriedUser = usersClient.getById(authHeader, createdUser.id());

        assertEquals(firstName, retriedUser.firstName());
        assertEquals(lastName, retriedUser.lastName());
        assertEquals(username, retriedUser.username());

        List<User> users = usersClient.getUsers(authHeader);
        assertNotNull(users);
        assertTrue(users.stream()
                .map(User::username)
                .anyMatch(name -> name.equals(username)));

    }
    @Test
    void createSameUserTwice() {
        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());

        String firstName = "SameUserFirstName";
        String lastName = "SameUserLastName";
        String username = "SameUserUsername";

        User user = new User(0 ,firstName, lastName, username);

        User createdUser = usersClient.createUser(authHeader, user);

        assertEquals(firstName, createdUser.firstName());
        assertEquals(lastName, createdUser.lastName());
        assertEquals(username, createdUser.username());
        assertNotNull(createdUser.id());
        assertNotEquals(createdUser.id(), 0);

        HttpClientResponseException exception = assertThrows(HttpClientResponseException.class, () ->
usersClient.createUser(authHeader, user));
        assertEquals(exception.getStatus(), HttpStatus.CONFLICT);
        assertTrue(exception.getResponse().getBody(String.class).orElse("").contains("User with provided username
already exists"));

    }
}
```

1. Annotate the class with `@MicronautTest` so the Micronaut framework will initialize the application context and the embedded server. More info.

Edit `application.yml`

users/src/main/resources/application.yml

Copy

```
micronaut:
  application:
    name: users
authentication-credentials:
  username: ${username} 1
  password: ${password} 2
```

1. Placeholder for username that will be populated by Kubernetes.
2. Placeholder for password that will be populated by Kubernetes.

Create a `bootstrap.yml` file in the `resources` directory to **enable distributed configuration**. Add the following:

users/src/main/resources/bootstrap.yml

Copy

```
micronaut:
  application:
    name: users
  config-client:
    enabled: true 1
kubernetes:
  client:
    secrets:
      enabled: true 2
      use-api: true 3
```

1. Set `microanut.config-client.enabled: true` which is used to read and resolve configuration from distributed sources.
2. Set `kubernetes.client.secrets.enabled: true` enables Kubernetes secrets as distributed source.
3. Set `kubernetes.client.secrets.use-api: true` use Kubernetes API to fetch configuration.

Modify the `Application` class to use `dev` as a default environment:

> The Micronaut framework supports the concept of one or many default environments. A default environment is one that is only applied if no other environments are explicitly specified or deduced.

users/src/main/java/example/micronaut/Application.java

Copy

```
package example.micronaut;

import io.micronaut.runtime.Micronaut;

public class Application {
    public static void main(String[] args) {
        Micronaut.run(Application.class, args);
    }
}
```

Create `src/main/resources/application-dev.yml`. The Micronaut framework applies this configuration file only for the `dev` environment.

users/src/main/resources/application-dev.yml

```
micronaut:
  server:
    port: 8081 (1)
authentication-credentials:
  username: "test_username" (2)
  password: "test_password" (3)
```

1. Configure the application to listen on port 8081.
2. Hardcoded username for development environment.
3. Hardcoded password for development environment.

Create a file named `bootstrap-dev.yml` to disable distributed configuration in the dev environment:

users/src/main/resources/bootstrap-dev.yml

```
kubernetes:
  client:
    secrets:
      enabled: false (1)
```

1. Disable Kubernetes secrets client.

Create a file named `application-test.yml` which is used in the test environment:

users/src/test/resources/application-test.yml

```
authentication-credentials:
  username: "test_username" (1)
  password: "test_password" (2)
```

1. Hardcoded username for test environment.
2. Hardcoded password for test environment.

Run the unit test:

users

```
./gradlew test
```

4.2.2. RUNNING THE APPLICATION

Run `users` microservice:

users

```
./gradlew run
```

```
14:28:34.034 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 499ms. Server Running:
http://localhost:8081
```

4.3. Orders Microservice

Create the `orders` microservice using the **Micronaut Command Line Interface** or with **Micronaut Launch**.

```
mn create-app --features=discovery-kubernetes,management,security,serialization-jackson,kubernetes
example.micronaut.orders --build=gradle --lang=java
```

Copy

> If you don't specify the `--build` argument, Gradle is used as the build tool.
> If you don't specify the `--lang` argument, Java is used as the language.

If you use Micronaut Launch, select Micronaut Application as application type and add the `discovery-kubernetes`, `management`, `security`, `serialization-jackson` and `kubernetes` features.

The previous command creates a directory named `orders` and a Micronaut application inside it with default package `example.micronaut`.

> If you have an existing Micronaut application and want to add the functionality described here, you can view the dependency and configuration changes from the specified features and apply those changes to your application.

Create package with name `controllers` and create `OrdersController` and `ItemsController` classes to handle incoming HTTP requests into the `orders` microservice:

orders/src/main/java/example/micronaut/controllers/OrdersController.java

Copy

```
package example.micronaut.controllers;

import example.micronaut.models.Item;
import example.micronaut.models.Order;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.exceptions.HttpStatusException;
import io.micronaut.security.annotation.Secured;
import io.micronaut.security.rules.SecurityRule;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

@Controller("/orders")  ❶
@Secured(SecurityRule.IS_AUTHENTICATED)  ❷
public class OrdersController {

    private final List<Order> orders = new ArrayList<>();

    @Get("/{id}")  ❸
    public Order findById(@NotNull Integer id) {
        return orders.stream()
                .filter(it -> it.id().equals(id))
                .findFirst().orElse(null);
    }

    @Get  ❹
    public List<Order> getOrders() {
        return orders;
    }

    @Post  ❺
    public Order createOrder(@Body @Valid Order order) {
        if (order.itemIds() == null || order.itemIds().isEmpty()) {
            throw new HttpStatusException(HttpStatus.BAD_REQUEST, "Items must be supplied");
        }

        List<Item> items = order.itemIds().stream().map(
                x -> Item.items.stream().filter(
                        y -> y.id().equals(x)
                ).findFirst().orElseThrow(
                        () -> new HttpStatusException(HttpStatus.BAD_REQUEST, String.format("Item with id %s doesn't
exists", x))
                )
        ).collect(Collectors.toList());

        BigDecimal total = items.stream().map(Item::price).reduce(BigDecimal::add).orElse(new BigDecimal("0"));
        Order newOrder = new Order(orders.size() + 1, order.userId(), items, null, total);

        orders.add(newOrder);
        return newOrder;
    }

}
```

❶ The class is defined as a controller with the @Controller annotation mapped to the path `/orders`.

❷ Annotate with `io.micronaut.security.Secured` to configure secured access. The `isAuthenticated()` expression will allow access only to authenticated users.

❸ The @Get annotation maps the `findById` method to an HTTP GET request on `/orders/{id}`.

❹ The @Get annotation maps the `getOrders` method to an HTTP GET request on `/orders`.

❺ The @Post annotation maps the `createOrder` method to an HTTP POST request on `/orders`.

orders/src/main/java/example/micronaut/controllers/ItemsController.java

Copy

```
package example.micronaut.controllers;

import example.micronaut.models.Item;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.security.annotation.Secured;
import io.micronaut.security.rules.SecurityRule;

import javax.validation.constraints.NotNull;
import java.util.List;

@Controller("/items")  1
@Secured(SecurityRule.IS_AUTHENTICATED)  2
public class ItemsController {

    @Get("/{id}")  3
    public Item findById(@NotNull Integer id) {
        return Item.items.stream()
                .filter(it -> it.id().equals(id))
                .findFirst().orElse(null);
    }

    @Get  4
    public List<Item> getItems() {
        return Item.items;
    }

}
```

1. The class is defined as a controller with the @Controller annotation mapped to the path `/items`.
2. Annotate with `io.micronaut.security.Secured` to configure secured access. The `isAuthenticated()` expression will allow access only to authenticated users.
3. The @Get annotation maps the `findById` method to an HTTP GET request on `/items/{id}`.
4. The @Get annotation maps the `getItems` method to an HTTP GET request on `/items`.

Create package with name `models` where you will put your data beans.

The previous `OrdersController` and `ItemsController` controller uses `Order` and `Item` objects to represent customer orders. Create the `Order` record:

orders/src/main/java/example/micronaut/models/Order.java

```
package example.micronaut.models;

import com.fasterxml.jackson.annotation.JsonProperty;
import io.micronaut.core.annotation.Nullable;
import io.micronaut.serde.annotation.Serdeable;

import javax.validation.constraints.Max;
import javax.validation.constraints.NotBlank;
import java.math.BigDecimal;
import java.util.List;

@Serdeable 1
public record Order(
        @Max(10000) @Nullable Integer id, 2
        @NotBlank @JsonProperty("user_id") Integer userId,
        @Nullable List<Item> items, 3
        @NotBlank @JsonProperty("item_ids") @Nullable List<Integer> itemIds, 4
        @Nullable BigDecimal total
) {
}
```

1. Declare the `@Serdeable` annotation at the type level in your source code to allow the type to be serialized or deserialized.
2. ID will be generated by application.
3. List of Item class will be populated by server and will be only visible in sever responses.
4. List of item_ids will be provided by client requests.

Create the `Item` record:

orders/src/main/java/example/micronaut/models/Item.java

Copy

```
package example.micronaut.models;

import io.micronaut.serde.annotation.Serdeable;

import javax.validation.constraints.Max;
import java.math.BigDecimal;
import java.util.Arrays;
import java.util.List;

@Serdeable (1)
public record Item(
        Integer id,
        String name,
        BigDecimal price
) {
    public static List<Item> items = Arrays.asList(
            new Item(1, "Banana", new BigDecimal("1.5")),
            new Item(2, "Kiwi", new BigDecimal("2.5")),
            new Item(3, "Grape", new BigDecimal("1.25"))
        );
}
```

(1) Declare the `@Serdeable` annotation at the type level in your source code to allow the type to be serialized or deserialized.

Create package with name `auth` where you will check basic authentication credentials.

The `Credentials` class will load and store credentials (username and password) from configuration.

orders/src/main/java/example/micronaut/auth/Credentials.java

Copy

```
package example.micronaut.auth;

import io.micronaut.context.annotation.ConfigurationProperties;

@ConfigurationProperties("authentication-credentials") (1)
public record Credentials (
        String username,
        String password
) {
}
```

(1) The `@ConfigurationProperties` annotation takes the configuration prefix.

The `CredentialsChecker` class, as name says, it will check if provided credentials inside request's Authorization header are the same with ones that are stored inside `Credentials` class that we created above.

orders/src/main/java/example/micronaut/auth/CredentialsChecker.java

Copy

```
package example.micronaut.auth;

import io.micronaut.core.annotation.Nullable;
import io.micronaut.http.HttpRequest;
import io.micronaut.security.authentication.AuthenticationProvider;
import io.micronaut.security.authentication.AuthenticationRequest;
import io.micronaut.security.authentication.AuthenticationResponse;
import jakarta.inject.Singleton;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Mono;

@Singleton (1)
public class CredentialsChecker implements AuthenticationProvider {

    private final Credentials credentials;

    public CredentialsChecker(Credentials credentials) {
        this.credentials = credentials;
    }

    @Override
    public Publisher<AuthenticationResponse> authenticate(@Nullable HttpRequest<?> httpRequest,
                                                          AuthenticationRequest<?, ?> authenticationRequest) {
        return Mono.<AuthenticationResponse>create(emitter -> {
            if ( authenticationRequest.getIdentity().equals(credentials.username()) &&
                    authenticationRequest.getSecret().equals(credentials.password()) ) {
                emitter.success(AuthenticationResponse.success((String) authenticationRequest.getIdentity()));
            } else {
                emitter.error(AuthenticationResponse.exception());
            }
        });
    }
}
```

(1) Use `jakarta.inject.Singleton` to designate a class as a singleton.

4.3.1. WRITE TESTS TO VERIFY APPLICATION LOGIC

Create the `OrderItemClient` Micronaut HTTP inline client for testing:

orders/src/test/java/example/micronaut/OrderItemClient.java

```
package example.micronaut;

import example.micronaut.models.Item;
import example.micronaut.models.Order;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Header;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.client.annotation.Client;

import java.util.List;

@Client("/") (1)
public interface OrderItemClient {

    @Get("/orders/{id}")
    Order getOrderById(@Header String authorization, Integer id);

    @Post("/orders")
    Order createOrder(@Header String authorization, @Body Order order);

    @Get("/orders")
    List<Order> getOrders(@Header String authorization);

    @Get("/items")
    List<Item> getItems(@Header String authorization);

    @Get("/items/{id}")
    Item getItemsById(@Header String authorization, Integer id);
}
```

1. Use `@Client` to use declarative HTTP Clients. You can annotate interfaces or abstract classes. You can use the `id` member to provide a service identifier or specify the URL directly as the annotation's value.

`HealthTest` checks if there is `/health` endpoint exposed that is needed for service discovery.

orders/src/test/java/example/micronaut/HealthTest.java

Copy

```java
package example.micronaut;

import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.client.annotation.Client;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;

import jakarta.inject.Inject;

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

@MicronautTest (1)
public class HealthTest {

    @Inject
    @Client("/")
    HttpClient client; (2)

    @Test
    public void healthEndpointExposed() {
        HttpStatus status = client.toBlocking().retrieve(HttpRequest.GET("/health"), HttpStatus.class);
        assertEquals(HttpStatus.OK, status);
    }
}
```

1. Annotate the class with `@MicronautTest` so the Micronaut framework will initialize the application context and the embedded server. More info.
2. Inject the `HttpClient` bean and point it to the embedded server.

`ItemsControllerTest` tests endpoints inside the `ItemController`.

orders/src/test/java/example/micronaut/ItemsControllerTest.java

Copy

```java
package example.micronaut;

import example.micronaut.auth.Credentials;
import example.micronaut.models.Item;
import io.micronaut.http.client.exceptions.HttpClientException;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import jakarta.inject.Inject;
import org.junit.jupiter.api.Test;

import java.math.BigDecimal;
import java.util.Arrays;
import java.util.Base64;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

@MicronautTest (1)
public class ItemsControllerTest {

    @Inject
    OrderItemClient orderItemClient;

    @Inject
    Credentials credentials;

    @Test
    void testUnauthorized() {
        HttpClientException exception = assertThrows(HttpClientException.class, () -> orderItemClient.getItems(""));
        assertTrue(exception.getMessage().contains("Unauthorized"));
    }

    @Test
    void getItem() {

        Integer itemId = 1;

        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());

        Item item = orderItemClient.getItemsById(authHeader, itemId);

        assertEquals(itemId, item.id());
        assertEquals("Banana", item.name());
        assertEquals(new BigDecimal("1.5"), item.price());

    }

    @Test
    void getItems() {
        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());

        List<Item> items = orderItemClient.getItems(authHeader);

        assertNotNull(items);
        List<String> existingItemNames = Arrays.asList("Kiwi", "Banana", "Grape");
        assertEquals(3, items.size());
        assertTrue(items.stream()
                .map(Item::name)
                .allMatch(name -> existingItemNames.stream().anyMatch(x -> x.equals(name))));
    }

}
```

1 Annotate the class with `@MicronautTest` so the Micronaut framework will initialize the application context and the embedded server. More info.

`OrdersControllerTest` tests endpoints inside the `OrdersController`.

orders/src/test/java/example/micronaut/OrdersControllerTest.java

Copy

```
package example.micronaut;

import example.micronaut.auth.Credentials;
import example.micronaut.models.Order;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.client.exceptions.HttpClientException;
import io.micronaut.http.client.exceptions.HttpClientResponseException;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import jakarta.inject.Inject;
import org.junit.jupiter.api.Test;

import java.math.BigDecimal;
import java.util.Arrays;
import java.util.Base64;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;

@MicronautTest (1)
public class OrdersControllerTest {

    @Inject
    OrderItemClient orderItemClient;

    @Inject
    Credentials credentials;

    @Test
    void testUnauthorized() {
        HttpClientException exception = assertThrows(HttpClientException.class, () -> orderItemClient.getOrders(""));
        assertTrue(exception.getMessage().contains("Unauthorized"));
    }

    @Test
    void multipleOrderInteraction() {
        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());

        Integer userId = 1;
        List<Integer> itemIds = Arrays.asList(1, 1, 2, 3);

        Order order = new Order(0, userId, null, itemIds, null);

        Order createdOrder = orderItemClient.createOrder(authHeader, order);

        assertNotNull(createdOrder.items());

        assertEquals(4, createdOrder.items().size());
        assertEquals(new BigDecimal("6.75"), createdOrder.total());
        assertEquals(userId, createdOrder.userId());

        Order retrievedOrder = orderItemClient.getOrderById(authHeader, createdOrder.id());

        assertNotNull(retrievedOrder.items());

        assertEquals(4, retrievedOrder.items().size());
        assertEquals(new BigDecimal("6.75"), retrievedOrder.total());
        assertEquals(userId, retrievedOrder.userId());

        List<Order> orders = orderItemClient.getOrders(authHeader);

        assertNotNull(orders);
        assertTrue(orders.stream()
                .map(Order::userId)
                .anyMatch(id -> id.equals(userId)));

    }

    @Test
    void itemDoesntExists() {
        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());
```

```
        Integer userId = 1;
        List<Integer> itemIds = List.of(5);

        Order order = new Order(0, userId, null, itemIds, null);

        HttpClientResponseException exception = assertThrows(HttpClientResponseException.class, () ->
orderItemClient.createOrder(authHeader, order));

        assertEquals(exception.getStatus(), HttpStatus.BAD_REQUEST);
        assertTrue(exception.getResponse().getBody(String.class).orElse("").contains("Item with id 5 doesn't exists"));
    }

    @Test
    void orderEmptyItems() {
        String authHeader = "Basic " + Base64.getEncoder().encodeToString((credentials.username() + ":" +
credentials.password()).getBytes());

        Integer userId = 1;
        Order order = new Order(0, userId, null, null, null);
        HttpClientResponseException exception = assertThrows(HttpClientResponseException.class, () ->
orderItemClient.createOrder(authHeader, order));

        assertEquals(exception.getStatus(), HttpStatus.BAD_REQUEST);
        assertTrue(exception.getResponse().getBody(String.class).orElse("").contains("Items must be supplied"));
    }

}
```

1. Annotate the class with `@MicronautTest` so the Micronaut framework will initialize the application context and the embedded server. More info.

Edit `application.yml`

orders/src/main/resources/application.yml

```
authentication-credentials:
  username: ${username} 1
  password: ${password} 2
```

1. Placeholder for username that will be populated by Kubernetes.
2. Placeholder for password that will be populated by Kubernetes.

Create a `bootstrap.yml` file in the `resources` directory to **enable distributed configuration**. Add the following:

orders/src/main/resources/bootstrap.yml

```
micronaut:
  application:
    name: orders
  config-client:
    enabled: true 1
kubernetes:
  client:
    secrets:
      enabled: true 2
      use-api: true 3
```

1. Set `microanut.config-client.enabled: true` which is used to read and resolve configuration from distributed sources.
2. Set `kubernetes.client.secrets.enabled: true` enables Kubernetes secrets as distributed source.
3. Set `kubernetes.client.secrets.use-api: true` use Kubernetes API to fetch configuration.

Modify the `Application` class to use `dev` as a default environment:

> The Micronaut framework supports the concept of one or many default environments. A default environment is one that is only applied if no other environments are explicitly specified or deduced.

orders/src/main/java/example/micronaut/Application.java

```java
package example.micronaut;

import io.micronaut.runtime.Micronaut;

public class Application {
    public static void main(String[] args) {
        Micronaut.run(Application.class, args);
    }
}
```

Create `src/main/resources/application-dev.yml`. The Micronaut framework applies this configuration file only for the `dev` environment.

orders/src/main/resources/application-dev.yml

```yaml
micronaut:
  server:
    port: 8082 (1)
authentication-credentials:
  username: "test_username" (2)
  password: "test_password" (3)
```

1. Configure the application to listen on port 8082.
2. Hardcoded username for development environment.
3. Hardcoded password for development environment.

Create a file named `bootstrap-dev.yml` to disable distributed configuration in the dev environment:

orders/src/main/resources/bootstrap-dev.yml

```yaml
kubernetes:
  client:
    secrets:
      enabled: false (1)
```

1. Disable Kubernetes secrets client.

Create a file named `application-test.yml` which is used in the test environment:

orders/src/test/resources/application-test.yml

```yaml
micronaut:
  application:
    name: orders
authentication-credentials:
  username: "test_username" (1)
  password: "test_password" (2)
```

1. Hardcoded username for development environment.
2. Hardcoded password for development environment.

Run the unit test:

orders

```
./gradlew test
```

4.3.2. RUNNING THE APPLICATION

Run `orders` microservice:

orders

```
./gradlew run
```

```
14:28:34.034 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 499ms. Server Running:
http://localhost:8082
```

4.4. API (Gateway) Microservice

Create the `api` microservice using the **Micronaut Command Line Interface** or with **Micronaut Launch**.

Copy

```
mn create-app --features=discovery-kubernetes,management,kubernetes,serialization-jackson,mockito example.microna
--build=gradle --lang=java
```

If you don't specify the `--build` argument, Gradle is used as the build tool.
If you don't specify the `--lang` argument, Java is used as the language.

If you use Micronaut Launch, select Micronaut Application as application type and add the `discovery-kubernetes`, `management`, `kubernetes`, `serialization-jackson` and `mockito` features.

The previous command creates a directory named `api` and a Micronaut application inside it with default package `example.micronaut`.

If you have an existing Micronaut application and want to add the functionality described here, you can view the dependency and configuration changes from the specified features and apply those changes to your application.

Create package with name `controllers` and create `GatewayController` class to handle incoming HTTP requests into the `api` microservice:

api/src/main/java/example/micronaut/controllers/GatewayController.java

Copy

```
package example.micronaut.controllers;

import example.micronaut.clients.OrdersClient;
import example.micronaut.clients.UsersClient;
import example.micronaut.models.Item;
import example.micronaut.models.Order;
import example.micronaut.models.User;
import io.micronaut.core.annotation.NonNull;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.exceptions.HttpStatusException;
import io.micronaut.scheduling.TaskExecutors;
import io.micronaut.scheduling.annotation.ExecuteOn;
import io.micronaut.validation.Validated;

import javax.validation.Valid;
import java.util.ArrayList;
import java.util.List;

@Controller("/api") ❶
@Validated
@ExecuteOn(TaskExecutors.IO) ❷
public class GatewayController {

    private final OrdersClient orderClient;
    private final UsersClient userClient;

    public GatewayController(OrdersClient orderClient, UsersClient userClient) {
        this.orderClient = orderClient;
        this.userClient = userClient;
    }

    @Get("/users/{id}") ❸
    public User getUserById(@NonNull Integer id) {
        return userClient.getById(id);
    }

    @Get("/orders/{id}") ❹
    public Order getOrdersById(@NonNull Integer id) {
        Order order = orderClient.getOrderById(id);
        return new Order(order.id(), null, getUserById(order.userId()), order.items(), order.itemIds(), order.total());
    }

    @Get("/items/{id}") ❺
    public Item getItemsById(@NonNull Integer id) {
        return orderClient.getItemsById(id);
    }

    @Get("/users") ❻
    public List<User> getUsers() {
        return userClient.getUsers();
    }

    @Get("/items") ❼
    public List<Item> getItems() {
        return orderClient.getItems();
    }

    @Get("/orders") ❽
    public List<Order> getOrders() {
        List<Order> orders = new ArrayList<>();
        orderClient.getOrders().forEach(x-> orders.add(new Order(x.id(), null, getUserById(x.userId()), x.items(),
x.itemIds(), x.total())));
        return orders;
    }

    @Post("/orders") ❾
    public Order createOrder(@Body @Valid Order order) {
        User user = getUserById(order.userId());
        if (user == null) {
            throw new HttpStatusException(HttpStatus.BAD_REQUEST, String.format("User with id %s doesn't exist",
order.userId()));
        }
        Order createdOrder = orderClient.createOrder(order);
        return new Order(createdOrder.id(), null, user, createdOrder.items(), createdOrder.itemIds(),
createdOrder.total());
    }

    @Post("/users") 10
    public User createUser(@Body @NonNull User user) {
        return userClient.createUser(user);
    }

}
```

1. The class is defined as a controller with the @Controller annotation mapped to the path `/api`.
2. It is critical that any blocking I/O operations (such as fetching the data from the database) are offloaded to a separate thread pool that does not block the Event loop.
3. The @Get annotation maps the `getUserById` method to an HTTP GET request on `/users/{id}`.
4. The @Get annotation maps the `getOrdersById` method to an HTTP GET request on `/orders/{id}`.
5. The @Get annotation maps the `getItemsById` method to an HTTP GET request on `/items/{id}`.
6. The @Get annotation maps the `getUsers` method to an HTTP GET request on `/users`.
7. The @Get annotation maps the `getItems` method to an HTTP GET request on `/items`.
8. The @Get annotation maps the `getOrders` method to an HTTP GET request on `/orders`.
9. The @Post annotation maps the `createUser` method to an HTTP POST request on `/users`.
10. The @Post annotation maps the `createOrder` method to an HTTP POST request on `/orders`.

Create package with name `models` where you will put your data beans.

The previous `GatewayController` and `ItemsController` controller uses `User`, `Order` and `Item` to represent customer orders. Create the `User` record:

api/src/main/java/example/micronaut/models/User.java

Copy

```
package example.micronaut.models;

import com.fasterxml.jackson.annotation.JsonProperty;
import io.micronaut.core.annotation.Nullable;
import io.micronaut.serde.annotation.Serdeable;

import javax.validation.constraints.Max;
import javax.validation.constraints.NotBlank;

@Serdeable 1
public record User(
        @Nullable @Max(10000) Integer id,
        @NotBlank @JsonProperty("first_name") String firstName,
        @NotBlank @JsonProperty("last_name") String lastName,
        String username
) {
}
```

1. Declare the `@Serdeable` annotation at the type level in your source code to allow the type to be serialized or deserialized.

Create the `Order` record:

api/src/main/java/example/micronaut/models/Order.java

Copy

```java
package example.micronaut.models;

import com.fasterxml.jackson.annotation.JsonProperty;
import io.micronaut.core.annotation.Nullable;
import io.micronaut.serde.annotation.Serdeable;

import javax.validation.constraints.Max;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotEmpty;
import javax.validation.constraints.NotNull;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

@Serdeable (1)
public record Order(
        @Nullable @Max(10000) Integer id,
        @NotBlank @Nullable @JsonProperty("user_id") Integer userId,
        @Nullable User user,
        @Nullable List<Item> items,
        @NotBlank @Nullable @JsonProperty("item_ids") List<Integer> itemIds,
        @Nullable BigDecimal total
) {
}
```

(1) Declare the `@Serdeable` annotation at the type level in your source code to allow the type to be serialized or deserialized.

Create the `Item` record:

api/src/main/java/example/micronaut/models/Item.java

Copy

```java
package example.micronaut.models;

import io.micronaut.serde.annotation.Serdeable;

import javax.validation.constraints.Max;
import java.math.BigDecimal;
import java.util.Arrays;
import java.util.List;

@Serdeable (1)
public record Item(
        Integer id,
        String name,
        BigDecimal price
) {
}
```

(1) Declare the `@Serdeable` annotation at the type level in your source code to allow the type to be serialized or deserialized.

Create package with name `clients` where you will put HTTP Clients that will call `users` and `orders` services.

Create `UserClient` for `users` service.

api/src/main/java/example/micronaut/clients/UsersClient.java

Copy

```java
package example.micronaut.clients;

import example.micronaut.models.User;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.client.annotation.Client;

import java.util.List;

@Client("users") (1)
public interface UsersClient {
    @Get("/users/{id}")
    User getById(Integer id);

    @Post("/users")
    User createUser(@Body User user);

    @Get("/users")
    List<User> getUsers();
}
```

1. Use `@Client` to use declarative HTTP Clients. You can annotate interfaces or abstract classes. You can use the `id` member to provide a service identifier or specify the URL directly as the annotation’s value.

Create `OrdersClient` for `orders` service.

api/src/main/java/example/micronaut/clients/OrdersClient.java

Copy

```java
package example.micronaut.clients;

import example.micronaut.models.Item;
import example.micronaut.models.Order;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.client.annotation.Client;

import java.util.List;


@Client("orders") (1)
public interface OrdersClient {
    @Get("/orders/{id}")
    Order getOrderById(Integer id);

    @Post("/orders")
    Order createOrder(@Body Order order);

    @Get("/orders")
    List<Order> getOrders();

    @Get("/items")
    List<Item> getItems();

    @Get("/items/{id}")
    Item getItemsById(Integer id);
}
```

1. Use `@Client` to use declarative HTTP Clients. You can annotate interfaces or abstract classes. You can use the `id` member to provide a service identifier or specify the URL directly as the annotation’s value.

Create package with name `auth` where we will check basic authentication credentials.

Create `Credentials` class that will load username and password from configuration that will be needed for comparison.

api/src/main/java/example/micronaut/auth/Credentials.java

Copy

```java
package example.micronaut.auth;

import io.micronaut.context.annotation.ConfigurationProperties;

@ConfigurationProperties("authentication-credentials") // 1
public record Credentials (
        String username,
        String password
) {
}
```

1. The `@ConfigurationProperties` annotation takes the configuration prefix.

Create `AuthClientFilter` class that is a client filter and will be applied to every client. It adds basic authentication header with credentials that are stored in `Credentials` class.

api/src/main/java/example/micronaut/auth/AuthClientFilter.java

```java
package example.micronaut.auth;

import io.micronaut.http.HttpResponse;
import io.micronaut.http.MutableHttpRequest;
import io.micronaut.http.annotation.Filter;
import io.micronaut.http.filter.ClientFilterChain;
import io.micronaut.http.filter.HttpClientFilter;
import org.reactivestreams.Publisher;

@Filter(Filter.MATCH_ALL_PATTERN)
public class AuthClientFilter implements HttpClientFilter {

    private final Credentials credentials;

    public AuthClientFilter(Credentials credentials) {
        this.credentials = credentials;
    }

    @Override
    public Publisher<? extends HttpResponse<?>> doFilter(MutableHttpRequest<?> request, ClientFilterChain chain) {
        return chain.proceed(request.basicAuth(credentials.username(), credentials.password()));
    }
}
```

Create `ErrorExceptionHandler` in `example.micronaut` package. The `ErrorExceptionHandler` will propagate errors from `orders` and `users` microservices.

api/src/main/java/example/micronaut/ErrorExceptionHandler.java

```java
package example.micronaut;

import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.annotation.Produces;
import io.micronaut.http.client.exceptions.HttpClientResponseException;
import io.micronaut.http.server.exceptions.ExceptionHandler;
import jakarta.inject.Singleton;

@Singleton // 1
public class ErrorExceptionHandler implements ExceptionHandler<HttpClientResponseException, HttpResponse<?>> {

    @Override
    public HttpResponse<?> handle(HttpRequest request, HttpClientResponseException exception) {
        return
HttpResponse.status(exception.getResponse().status()).body(exception.getResponse().getBody(String.class).orElse(null));
    }
}
```

1. Use `jakarta.inject.Singleton` to designate a class as a singleton.

4.4.1. WRITE TESTS TO VERIFY APPLICATION LOGIC

Create the `GatewayClient` Micronaut HTTP inline client for testing:

api/src/test/java/example/micronaut/GatewayClient.java

```
package example.micronaut;

import example.micronaut.models.Item;
import example.micronaut.models.Order;
import example.micronaut.models.User;
import io.micronaut.http.annotation.Body;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;
import io.micronaut.http.client.annotation.Client;

import java.util.List;

@Client("/") (1)
public interface GatewayClient {

    @Get("/api/items/{id}")
    Item getItemById(Integer id);

    @Get("/api/orders/{id}")
    Order getOrderById(Integer id);

    @Get("/api/users/{id}")
    User getUsersById(Integer id);

    @Get("/api/users")
    List<User> getUsers();

    @Get("/api/items")
    List<Item> getItems();

    @Get("/api/orders")
    List<Order> getOrders();

    @Post("/api/orders")
    Order createOrder(@Body Order order);

    @Post("/api/users")
    User createUser(@Body User user);
}
```

1. Use `@Client` to use declarative HTTP Clients. You can annotate interfaces or abstract classes. You can use the `id` member to provide a service identifier or specify the URL directly as the annotation's value.

`HealthTest` checks if there is `/health` endpoint exposed that is needed for service discovery.

api/src/test/java/example/micronaut/HealthTest.java

```
package example.micronaut;

import io.micronaut.http.HttpRequest;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.client.HttpClient;
import io.micronaut.http.client.annotation.Client;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import org.junit.jupiter.api.Test;

import jakarta.inject.Inject;

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

@MicronautTest (1)
public class HealthTest {

    @Inject
    @Client("/")
    HttpClient client; (2)

    @Test
    public void healthEndpointExposed() {
        HttpStatus status = client.toBlocking().retrieve(HttpRequest.GET("/health"), HttpStatus.class);
        assertEquals(HttpStatus.OK, status);
    }
}
```

1. Annotate the class with `@MicronautTest` so the Micronaut framework will initialize the application context and the embedded server. More info.
2. Inject the `HttpClient` bean and point it to the embedded server.

`GatewayControllerTest` tests endpoints inside the `GatewayController`.

api/src/test/java/example/micronaut/GatewayControllerTest.java

Copy

```java
package example.micronaut;

import example.micronaut.clients.OrdersClient;
import example.micronaut.clients.UsersClient;
import example.micronaut.models.Item;
import example.micronaut.models.Order;
import example.micronaut.models.User;
import io.micronaut.http.HttpResponse;
import io.micronaut.http.HttpStatus;
import io.micronaut.http.client.exceptions.HttpClientResponseException;
import io.micronaut.test.annotation.MockBean;
import io.micronaut.test.extensions.junit5.annotation.MicronautTest;
import jakarta.inject.Inject;
import org.junit.jupiter.api.Test;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertNull;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;


@MicronautTest 1
public class GatewayControllerTest {

    @Inject
    OrdersClient ordersClient;

    @Inject
    UsersClient usersClient;

    @Inject
    GatewayClient gatewayClient;

    @MockBean(OrdersClient.class)
    OrdersClient ordersClient() {
        return mock(OrdersClient.class);
    }

    @MockBean(UsersClient.class)
    UsersClient usersClient() {
        return mock(UsersClient.class);
    }


    @Test
    void getItemById() {
        int itemId = 1;
        Item item = new Item(itemId, "test", BigDecimal.ONE);

        when(ordersClient.getItemsById(1)).thenReturn(item);

        Item retrievedItem = gatewayClient.getItemById(item.id());

        assertEquals(item.id(), retrievedItem.id());
        assertEquals(item.name(), retrievedItem. name());
        assertEquals(item.price(), retrievedItem.price());

    }

    @Test
    void getOrderById() {

        Order order = new Order(1, 2, null, null, new ArrayList<>(), null);
        User user = new User(order.userId(), "firstName", "lastName", "test");
        when(ordersClient.getOrderById(1)).thenReturn(order);
        when(usersClient.getById(user.id())).thenReturn(user);

        Order retrievedOrder = gatewayClient.getOrderById(order.id());

        assertEquals(order.id(), retrievedOrder.id());
        assertEquals(order.userId(), retrievedOrder.user().id());
        assertNull(retrievedOrder.userId());
        assertEquals(user.username(), retrievedOrder.user().username());
    }

    @Test
    void getUserById() {
        User user = new User(1, "firstName", "lastName", "test");

        when(usersClient.getById(1)).thenReturn(user);

        User retrievedUser = gatewayClient.getUsersById(user.id());

        assertEquals(user.id(), retrievedUser.id());
        assertEquals(user.username(), retrievedUser.username());
    }

    @Test
    void getUsers() {
        User user = new User(1, "firstName", "lastName", "test");

        when(usersClient.getUsers()).thenReturn(List.of(user));

        List<User> users = gatewayClient.getUsers();

        assertNotNull(users);
        assertEquals(1, users.size());
        assertEquals(user.id(), users.get(0).id());
        assertEquals(user.username(), users.get(0).username());
    }

    @Test
    void getItems() {

        Item item = new Item(1, "test", BigDecimal.ONE);

        when(ordersClient.getItems()).thenReturn(List.of(item));

        List<Item> items = gatewayClient.getItems();

        assertNotNull(items);
        assertEquals(1, items.size());
        assertEquals(item.name(), items.get(0).name());
        assertEquals(item.price(), items.get(0).price());
    }

    @Test
    void getOrders() {
        Order order = new Order(1, 2, null, null, new ArrayList<>(), null);
        User user = new User(order.userId(), "firstName", "lastName", "test");

        when(ordersClient.getOrders()).thenReturn(List.of(order));
        when(usersClient.getById(order.userId())).thenReturn(user);

        List<Order> orders = gatewayClient.getOrders();

        assertNotNull(orders);
        assertEquals(1, orders.size());
        assertNull(orders.get(0).userId());
        assertEquals(user.id(), orders.get(0).user().id());

        assertEquals(order.id(), orders.get(0).id());
        assertEquals(user.username(), orders.get(0).user().username());
    }

    @Test
    void createUser() {
        String firstName = "firstName";
        String lastName = "lastName";
        String username = "username";
        User user = new User(0, firstName, lastName, username);

        when(usersClient.createUser(any())).thenReturn(user);

        User createdUser = gatewayClient.createUser(user);

        assertEquals(firstName, createdUser.firstName());
        assertEquals(lastName, createdUser.lastName());
        assertEquals(username, createdUser.username());
    }

    @Test
    void createOrder() {
        Order order = new Order(1, 2, null, null, new ArrayList<>(), null);
        User user = new User(order.userId(), "firstName", "lastName", "test");

        when(usersClient.getById(user.id())).thenReturn(user);

        when(ordersClient.createOrder(any())).thenReturn(order);

        Order createdOrder = gatewayClient.createOrder(order);

        assertEquals(order.id(), createdOrder.id());
        assertNull(createdOrder.userId());
        assertEquals(order.userId(), createdOrder.user().id());
        assertEquals(user.username(), createdOrder.user().username());
    }

    @Test
    void createOrderUserDoesntExists() {
        Order order = new Order(1, 2, null, null, new ArrayList<>(), new BigDecimal(0));;

        when(ordersClient.createOrder(any())).thenReturn(order);

        when(usersClient.getById(order.userId())).thenReturn(null);

        HttpClientResponseException exception = assertThrows(HttpClientResponseException.class, () ->
gatewayClient.createOrder(order));

        assertEquals(exception.getStatus(), HttpStatus.BAD_REQUEST);

        assertTrue(exception.getResponse().getBody(String.class).orElse("").contains("User with id 2 doesn't exist"));
    }

    @Test
    void exceptionHandler() {
        User user = new User(1, "firstname", "lastname", "username");

        String message = "Test error message";

        when(usersClient.createUser(any())).thenThrow(new HttpClientResponseException("Test",
HttpResponse.badRequest(message)));

        HttpClientResponseException exception = assertThrows(HttpClientResponseException.class, () ->
gatewayClient.createUser(user));

        assertEquals(exception.getStatus(), HttpStatus.BAD_REQUEST);

        assertTrue(exception.getResponse().getBody(String.class).orElse("").contains("Test error message"));
    }

}
```

1. Annotate the class with `@MicronautTest` so the Micronaut framework will initialize the application context and the embedded server. More info.

Edit `application.yml`

api/src/main/resources/application.yml

```
authentication-credentials:
  username: ${username} ❶
  password: ${password} ❷
```

1. Placeholder for username that will be populated by Kubernetes.

2. Placeholder for password that will be populated by Kubernetes.

Create a `bootstrap.yml` file in the `resources` directory to **enable distributed configuration**. Add the following:

api/src/main/resources/bootstrap.yml

```yaml
micronaut:
  application:
    name: api
  config-client:
    enabled: true (1)
kubernetes:
  client:
    secrets:
      enabled: true (2)
      use-api: true (3)
```

1. Set `microanut.config-client.enabled: true` which is used to read and resolve configuration from distributed sources.
2. Set `kubernetes.client.secrets.enabled: true` enables Kubernetes secrets as distributed source.
3. Set `kubernetes.client.secrets.use-api: true` use Kubernetes API to fetch configuration.

Modify the `Application` class to use `dev` as a default environment:

> The Micronaut framework supports the concept of one or many default environments. A default environment is one that is only applied if no other environments are explicitly specified or deduced.

api/src/main/java/example/micronaut/Application.java

```java
package example.micronaut;

import io.micronaut.runtime.Micronaut;

public class Application {
    public static void main(String[] args) {
        Micronaut.run(Application.class, args);
    }
}
```

Create `src/main/resources/application-dev.yml`. The Micronaut framework applies this configuration file only for the `dev` environment.

api/src/main/resources/application-dev.yml

```yaml
authentication-credentials:
  username: "test_username" (1)
  password: "test_password" (2)
```

1. Hardcoded username for development environment.
2. Hardcoded password for development environment.

Create a file named `bootstrap-dev.yml` to disable distributed configuration in the dev environment:

api/src/main/resources/bootstrap-dev.yml

```yaml
micronaut:
  http:
    services:
      users:
        urls:
          - http://localhost:8081 (1)
      orders:
        urls:
          - http://localhost:8082 (2)
kubernetes:
  client:
    secrets:
      enabled: false (3)
```

1. Url of the `users` microservice

❷ Url of the `orders` microservice
❸ Disable Kubernetes secrets client.

Create a file named `application-test.yml` which is used in the test environment:

api/src/test/resources/application-test.yml

```yaml
authentication-credentials:
  username: "test_username" ❶
  password: "test_password" ❷
```

❶ Hardcoded username for development environment.
❷ Hardcoded password for development environment.

Run the unit test:

users

```
./gradlew test
```

4.4.2. RUNNING THE APPLICATION

Run `api` microservice:

orders

```
./gradlew run
```

```
14:28:34.034 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 499ms. Server Running:
http://localhost:8080
```

4.5. Test integration between applications

Store the api microservice url to API_URL variable.

```
export API_URL=http://localhost:8080
```

Run a cURL command to create user through api:

```
curl -X "POST" "$API_URL/api/users" -H 'Content-Type: application/json; charset=utf-8' -d '{ "first_name": "Neman
"last_name": "Mikic", "username": "nmikic" }'
```

```
{"id":1,"username":"nmikic","first_name":"Nemanja","last_name":"Mikic"}
```

Run a cURL command to create order through api:

```
curl -X "POST" "$API_URL/api/orders" -H 'Content-Type: application/json; charset=utf-8' -d '{ "user_id": 1, "item_
[1,2] }'
```

```
{"id":1,"user":{"first_name":"Nemanja","last_name":"Mikic","id":1,"username":"nmikic"},"items":
[{"id":1,"name":"Banana","price":1.5},{"id":2,"name":"Kiwi","price":2.5}],"total":4.0}
```

Run a cURL to list created orders:

```
curl "$API_URL/api/orders" -H 'Content-Type: application/json; charset=utf-8'
```

```
[{"id":1,"user":{"first_name":"Nemanja","last_name":"Mikic","id":1,"username":"nmikic"},"items":
[{"id":1,"name":"Banana","price":1.5},{"id":2,"name":"Kiwi","price":2.5}],"total":4.0}]
```

We can try to place an order with user that doesn't exist (with id 100). Run a cURL command:

```
curl -X "POST" "$API_URL/api/orders" -H 'Content-Type: application/json; charset=utf-8' -d '{ "user_id": 100,
"item_ids": [1,2] }'
```

```
{"message":"Bad Request","_links":{"self":[{"href":"/api/orders","templated":false}]},"_embedded":{"errors":
[{"message":"User with id 100 doesn't exist"}]}}
```

5. Kubernetes and the Micronaut framework

In this chapter we will first create necessary kubernetes resources for our services that will make them work properly then we will configure, build docker image and deploy each of microservices that we created on the local Kubernetes cluster.

Create `auth.yml` file that will service role for microservices and have secret configuration.

auth.yml

```
apiVersion: v1
kind: ServiceAccount (1)
metadata:
  name: micronaut-service
---
kind: Role (2)
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: micronaut_service_role
rules:
  - apiGroups: [""]
    resources: ["services", "endpoints", "configmaps", "secrets", "pods"]
    verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding (3)
metadata:
  name: micronaut_service_role_bind
subjects:
  - kind: ServiceAccount
    name: micronaut-service
roleRef:
  kind: Role
  name: micronaut_service_role
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: v1
kind: Secret (4)
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4= (5)
  password: bWljcm9uYXV0aXNhd2Vzb21l (6)
```

1. We crate a service account with name `micronaut-service`.
2. We create a role with name `micronaut_service_role`.
3. We bind `micronaut_service_role` role to the `micronaut-service` service account.
4. We create a secret with name `micronaut_service_role`.
5. Base64 value of username secret that will be used by ours microservices.
6. Base64 value of password secret that will be used by ours microservices.

Run next command to create described resources:

```
kubectl create -f auth.yml
```

Before we start deploying each of service, we have to make sure that Docker daemon is configured to use Kubernetes. if you are using **Minikube** execute next comand to switch docker deamon to use Minikube.

```
eval $(minikube docker-env)
```

5.1. Users Microservice

The next command will build docker image of `users` service with image name `users`.

```
./gradlew dockerBuild
```

Edit `k8s.yml` inside `users` service.

/users/k8s.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "users"
spec:
  selector:
    matchLabels:
      app: "users"
  template:
    metadata:
      labels:
        app: "users"
    spec:
      serviceAccountName: micronaut-service ❶
      containers:
        - name: "users"
          image: users ❷
          imagePullPolicy: Never ❸
          ports:
            - name: http
              containerPort: 8080
          readinessProbe:
            httpGet:
              path: /health/readiness
              port: 8080
            initialDelaySeconds: 5
            timeoutSeconds: 3
          livenessProbe:
            httpGet:
              path: /health/liveness
              port: 8080
            initialDelaySeconds: 5
            timeoutSeconds: 3
            failureThreshold: 10
---
apiVersion: v1
kind: Service
metadata:
  name: "users" ❹
spec:
  selector:
    app: "users"
  type: NodePort
  ports:
    - protocol: "TCP"
      port: 8080  ❺
```

❶ Service name that we created in auth.yaml.
❷ Name of the image that deployment use.
❸ The imagePullPolicy is set to Never. We will always use local one that we built in previous step.
❹ Name of a service, required for service discovery.
❺ Micronaut default port on which application is running.

Run next command to deploy users microservice:

```
kubectl create -f users/k8s.yml
```

5.2. Orders Microservice

The next command will build docker image of `orders` service with image name `orders`.

```
./gradlew dockerBuild
```

Edit `k8s.yml` inside `orders` service.

/orders/k8s.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "orders"
spec:
  selector:
    matchLabels:
      app: "orders"
  template:
    metadata:
      labels:
        app: "orders"
    spec:
      serviceAccountName: micronaut-service  (1)
      containers:
        - name: "orders"
          image: orders  (2)
          imagePullPolicy: Never  (3)
          ports:
            - name: http
              containerPort: 8080
          readinessProbe:
            httpGet:
              path: /health/readiness
              port: 8080
            initialDelaySeconds: 5
            timeoutSeconds: 3
          livenessProbe:
            httpGet:
              path: /health/liveness
              port: 8080
            initialDelaySeconds: 5
            timeoutSeconds: 3
            failureThreshold: 10
---
apiVersion: v1
kind: Service
metadata:
  name: "orders"  (4)
spec:
  selector:
    app: "orders"
  type: NodePort
  ports:
    - protocol: "TCP"
      port: 8080  (5)
```

1. Service name that we created in auth.yaml.
2. Name of the image that deployment use.
3. The imagePullPolicy is set to Never. We will always use local one that we built in previous step.
4. Name of a service, required for service discovery.
5. Micronaut default port on which application is running.

Run next command to deploy orders microservice:

```
kubectl create -f orders/k8s.yml
```

5.3. API (Gateway) Microservice

The next command will build docker image of `api` service with image name `api`.

```
./gradlew dockerBuild
```

Edit `k8s.yml` inside `api` service.

/api/k8s.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "api"
spec:
  selector:
    matchLabels:
      app: "api"
  template:
    metadata:
      labels:
        app: "api"
    spec:
      serviceAccountName: micronaut-service ❶
      containers:
        - name: "api"
          image: api  ❷
          imagePullPolicy: Never  ❸
          ports:
            - name: http
              containerPort: 8080
          readinessProbe:
            httpGet:
              path: /health/readiness
              port: 8080
            initialDelaySeconds: 5
            timeoutSeconds: 3
          livenessProbe:
            httpGet:
              path: /health/liveness
              port: 8080
            initialDelaySeconds: 5
            timeoutSeconds: 3
            failureThreshold: 10
---
apiVersion: v1
kind: Service
metadata:
  name: "api"  ❹
spec:
  selector:
    app: "api"
  type: LoadBalancer
  ports:
    - protocol: "TCP"
      port: 8080  ❺
```

❶ Service name that we created in auth.yaml.
❷ Name of the image that deployment use.
❸ The imagePullPolicy is set to Never. We will always use local one that we built in previous step.
❹ Name of a service, required for service discovery.
❺ Micronaut default port on which application is running.

Run next command to deploy api microservice:

```
kubectl create -f api/k8s.yml
```

5.4. Test integration between applications deployed on Kubernetes

Run next command to check status of created pods and make sure that all have status "Running":

```
kubectl get pods
```

```
NAME                        READY   STATUS    RESTARTS   AGE
api-774fd667b9-dmws4        1/1     Running   0          24s
orders-74ff4fcbc4-dnfbw     1/1     Running   0          19s
users-9f46dd7c6-vs8z7       1/1     Running   0          13s
```

Run next command to check status of created services:

```
kubectl get services
```

```
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
api          LoadBalancer   10.102.7.238    <pending>     8080:30049/TCP   24h
kubernetes   ClusterIP      10.96.0.1       <none>        443/TCP          26d
orders       NodePort       10.99.172.89    <none>        8080:31811/TCP   24h
users        NodePort       10.109.219.86   <none>        8080:32409/TCP   24h
```

By default, EXTERNAL-IP of LoadBalancer service inside Minikube will be in <pending> state. If you want to assign an external ip you have to run `minikube tunnel` command.

Run next command to get url of api service:

```
export API_URL=$(minikube service api --url)
```

Run a cURL command to create user through api:

```
curl -X "POST" "$API_URL/api/users" -H 'Content-Type: application/json; charset=utf-8' -d '{ "first_name": "Neman
"last_name": "Mikic", "username": "nmikic" }'
```

```
{"id":1,"username":"nmikic","first_name":"Nemanja","last_name":"Mikic"}
```

Run a cURL command to create order through api:

```
curl -X "POST" "$API_URL/api/orders" -H 'Content-Type: application/json; charset=utf-8' -d '{ "user_id": 1, "item_
[1,2] }'
```

```
{"id":1,"user":{"first_name":"Nemanja","last_name":"Mikic","id":1,"username":"nmikic"},"items":
[{"id":1,"name":"Banana","price":1.5},{"id":2,"name":"Kiwi","price":2.5}],"total":4.0}
```

Run a cURL to list created orders:

```
curl "$API_URL/api/orders" -H 'Content-Type: application/json; charset=utf-8'
```

```
[{"id":1,"user":{"first_name":"Nemanja","last_name":"Mikic","id":1,"username":"nmikic"},"items":
[{"id":1,"name":"Banana","price":1.5},{"id":2,"name":"Kiwi","price":2.5}],"total":4.0}]
```

We can try to place an order with user that doesn't exist (with id 100). Run a cURL command:

```
curl -X "POST" "$API_URL/api/orders" -H 'Content-Type: application/json; charset=utf-8' -d '{ "user_id": 100,
"item_ids": [1,2] }'
```

```
{"message":"Bad Request","_links":{"self":[{"href":"/api/orders","templated":false}]},"_embedded":{"errors":
[{"message":"User with id 100 doesn't exist"}]}}
```

6. Next steps

Explore more features with **Micronaut Guides**.

Read more about **Kubernetes**.

Read more about **Micronaut Kubernetes** module.

MICRONAUT FOUNDATION™

BRAND GUIDELINES

COMMUNITY GUIDELINES

PRIVACY POLICY