spring-cloud-gateway-example (0.0.1)

Maksim Kostromin

Version 0.0.1, 2018-06-22 18:43:49 UTC

Table of Contents

Travis CI status: [Build Status]

Chapter 1. Introduction

Migrate monolithic app into micro-services with awesome Spring projects!

Read reference documentation for details

gradle

```
./gradlew
java -jar build/monolith/libs/*.jar
bash build/libs/monolith/*.jar

./gradlew build composeUp
./gradlew composeDown
```

links:

- additional hibernate generators
- Thymeleaf getting ready for Reactive Spring 5
- YouTube: Thymeleaf by Daniel Fernández
- Motivated by that Spencer Gibb talk on YouTube: Introducing Spring Cloud Gateway by Spencer Gibb @ Spring I/O 2018
- YouTube: Mastering Spring Boot's Actuator by Andy Wilkinson @ Spring I/O 2018

generated by generator-jvm yeoman generator (java-spring-boot)

Chapter 2. Implementation

2.1. props

This module contains all apps props, such as applications url, port, host, etc...

configurations example file: `application-props.yaml`

```
spring:
  profiles:
    active: props
props:
  monolith:
    proto: http
    host: 127.0.0.1
    port: 8001
    url: ${props.monolith.proto}://${props.monolith.host}:${props.monolith.port}
  gateway:
    proto: http
    host: 127.0.0.1
    port: 8002
    url: ${props.gateway.proto}://${props.gateway.host}:${props.gateway.port}
  ui:
    proto: http
    host: 127.0.0.1
    port: 8003
    url: ${props.ui.proto}://${props.ui.host}:${props.ui.port}
  rest:
    proto: http
    host: 127.0.0.1
    port: 8004
    url: ${props.rest.proto}://${props.rest.host}:${props.rest.port}
```

2.2. step 0: monolith

This is a zero step. We will try migrate that monolith app, which is contains: `ui` and few `rest api data` modules into micro-services apps.

Monolith server is using port: 8001

2.3. step 1: gateway

This is a first step in micro-services migration process. First of all we need create entry point of our future system — application gateway. Gateway will forward any requests to proper services of your system.

Gateway server is using port: 8002

gateway routes configuration:

```
  final PropsAutoConfiguration.Props props;

  @Bean
  RouteLocator msRouteLocator(RouteLocatorBuilder builder) {
    return builder
        .routes()

        // step 4: forward rest api calls to ms-3-rest micro-service
        .route("ms-3-rest", p -> p
            .path("/api/**")
            .uri(props.getRest().getUrl()))

        // step 3: everything else (except itself gateway actuator endpoints) forward
to ms-2-ui micro-service
        .route("self-actuator", p -> p
            .path("/actuator/**")
            .negate()
            .uri(props.getUi().getUrl()))

        /* // monolithic app at this point of time could be completely disabled -
after step 4 migration is done.
        // step 1: forward everything to monolith app
        .route("monolith", p -> p
            .path("/**")
            .uri(props.getMonolith().getUrl()))

        // step 2: oops, gateway actuator endpoints should respond by themselves, but
not with monolith's...
        .route("self-actuator", p -> p
            .path("/actuator/**")
            .negate()
            .uri(props.getMonolith().getUrl()))
        */

        .build();
  }
```

this configuration shows how we can forward every request to monolith (except itself actuator requests)

2.4. step 2: ui

In this module we moved all UI related stuff:

- react frontend app
- SPA index thymeleaf controller
- API forwarder to gateway (to avoid frontend CORS issue)

SPA Index Page thymeleaf controller:

```
@Controller
@RequiredArgsConstructor
class IndexPage {

  @GetMapping("/")
  String index() {
    return "index";
  }
}
```

WebFlux REST API proxy forwarder config:

```
@Log4j2
@Configuration
@RequiredArgsConstructor
class RestApiProxyConfig {

  final Props props;

  @Bean
  WebClient webClient() {
    return WebClient.create(props.getMonolith().getUrl());
  }

  @Bean
  RouterFunction routes(WebClient webClient) {

    final ParameterizedTypeReference<Map> maps = new ParameterizedTypeReference<Map>()
{};
    final ParameterizedTypeReference<String> strings = new ParameterizedTypeReference
<String>() {};

    return route(
        GET("/api/contents"),
        request -> ok().contentType(APPLICATION_JSON).body(webClient
            .get().uri("/api/contents")
            .accept(APPLICATION_JSON)
            .header("Content-Type", APPLICATION_JSON_VALUE)
            .retrieve().bodyToFlux(maps), maps)
    ).andRoute(
        GET("/api/**"),
        request -> ok().contentType(APPLICATION_JSON).body(webClient
            .get().uri("/api/")
            .accept(APPLICATION_JSON)
            .header("Content-Type", APPLICATION_JSON_VALUE)
            .retrieve().bodyToFlux(strings), strings)
    ).andOther(
        resources("/**", new ClassPathResource("public/"))
    );
  }
}
```

UI is using port: 8003

2.5. step 3: rest

Last part of our application is REST API. We gonna split webflux rest api from monolith into separate service.

spring-data:

```
@Data
@Entity
@NoArgsConstructor
@Accessors(chain = true)
class Content implements Serializable {

  private static final long serialVersionUID = -7618202574843387015L;

  @Id
  @GeneratedValue(generator = "uuid")
  @GenericGenerator(name = "uuid", strategy = "uuid2")
  String id;

  String body;
}

interface Contents extends JpaRepository<Content, String> {}
```

WebFlux REST API:

```
@Log4j2
@Configuration
@RequiredArgsConstructor
class WebfluxRoutesConfig {

  final Contents contents;

  @Bean
  RouterFunction routes() {
    return
        nest(
            path("/api"),
            route(
                GET("/contents"),
                contentsHandler()
            )
        ).andRoute(
            GET("/**"),
            fallbackHandler()
        )
        ;
  }

  @Bean
  HandlerFunction<ServerResponse> contentsHandler() {
    return request -> {
      final Flux<Content> publisher = Flux.fromIterable(contents.findAll());
      final ParameterizedTypeReference<Content> type = new ParameterizedTypeReference
<Content>() {};
      return ok().body(publisher, type);
    };
  }

  @Bean
  HandlerFunction<ServerResponse> fallbackHandler() {
    return request -> {
      final ParameterizedTypeReference<List<String>> type = new
ParameterizedTypeReference<List<String>>() {};
      final List<String> api = singletonList("GET contents -> /api/contents");
      final Mono<List<String>> publisher = Mono.just(api);
      return ok().body(publisher, type);
    };
  }
}
```

Rest server is using port: 8004

Chapter 3. Post implementation steps:

- remove monolith `self-actuator` gateway route
- remove `ms-0-monolith` project
- remove useless configurations from `props` module

Chapter 4. Links

- GitHub repo
- GitHub pages