# BOM

## Создание репозитория на git

<https://github.com/smilyk/atsarat-briut-microservices>

1. Клонируем и открываем этот репозиторий в интелледж

smilykq@MacBook-Pro ~ % cd Desktop

smilykq@MacBook-Pro Desktop % cd AtsaratBriut

smilykq@MacBook-Pro AtsaratBriut % ls

AtcaratBriut\_plan

AtcaratBriut\_plan.docx

children-service

email-service

gymnast-server

scheduler-service

school-service

tsofim-service

user-service

~$caratBriut\_plan.docx

~$er-service-documentation.docx

~$щая документация.docx

Общая документация

Общая документация.docx

smilykq@MacBook-Pro AtsaratBriut % mkdir atsarat

smilykq@MacBook-Pro AtsaratBriut % cd atsarat

smilykq@MacBook-Pro atsarat % git clone https://github.com/smilyk/atsarat-briut-microservices.git

Клонирование в «atsarat-briut-microservices»…

warning: Похоже, что вы клонировали пустой репозиторий.

smilykq@MacBook-Pro atsarat %

Изображение выглядит как текст

Автоматически созданное описание

<https://github.com/smilyk/atsarat-briut-microservices.git>

1. Открываем в интелледже ->New project from maven
2. pom.xml, .gitignore, readme
3. удалить src
4. добавляем модули
   1. Изображение выглядит как текст

      Автоматически созданное описание
5. git push

## Общий pom –

1. Все одинаковые зависимости убираются из pom сервисов и добавляется в общий родительский pom
2. Каждый сервис получает свой порт и свое имя

# ACTUATOR

<!-- ACTUATOR-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

#actuator  
spring.jackson.serialization.indent\_output=true – нормальный внешний вид  
management.endpoints.web.base-path=/monitor – изменили ссылку. Теперь вместо …/actuator -> …/monitor

GET http://localhost:8082/monitor  
management.endpoints.web.exposure.include=\* - включили все конечные точки  
management.endpoint.shutdown.enabled=true – подключение точки shutdown  
POST http://localhost:8082/monitor/shutdown  
info.application.name= user-service actuator  
info.application.description=project "Atsarat-Briut"  
info.application.version=0.0.1

Если все ОК, то при старте мы увидим:

Exposing 14 endpoint(s) beneath base path '/monitor'

# RABBITMQ очередь

1. Открываем раббит в докере

|  |  |
| --- | --- |
| docker run -d --hostname rabbitmq --name rabbitmq -p 5672:5672 -p 15672:15672 -p 15674:15674 -p 25672:25672 -p 61613:61613 -v rabbitmq\_data:/var/lib/rabbitmq -e RABBITMQ\_DEFAULT\_USER=user -e RABBITMQ\_DEFAULT\_PASS=password rabbitmq:3.6.14-management | Создаем и запускаем контейнер  логин:user  пароль:password |

1. В общий pom добавляем зависимость:

<!-- rabbitMQ-->  
 <dependency>  
 <groupId> org.springframework.boot </groupId>  
 <artifactId> spring-boot-starter-amqp </artifactId>  
 </dependency>  
<!--end of rabbitMQ-->

## Настраиваем сервис Sheduler.

1. Сервис будет создавать 3 очереди –

* schoolQueue
* gymnastQueue
* tsofimQueue

1. application.properties
2. #rabbitMQ  
   spring.rabbitmq.host = localhost  
   spring.rabbitmq.port = 5672  
   spring.rabbitmq.username = user  
   spring.rabbitmq.password = password  
     
   tsofim.key=ts  
   school.key=sc  
   gymnsat.key=gym  
     
   tsofim.queue=TSOFIM  
   school.queue=SCHOOL  
   gymnsat.queue=GYMNAST  
     
   rabbitmq.exchange=atBriut
3. пишем конфигурацию для producer (class RabbitProducerConfig)

@Bean этот бин создает exchange в рабите ()то есть перераспределение – как бы канал, на который передаются все сообщения  
DirectExchange exchange() {  
 String name = exchange;  
 return new DirectExchange(name, true, false);  
}

После того, как создан exchange необходимо создать бин, который будет отправлять в конкретную очередь с конкретным ключом конкретное сообщение

@Bean  
public AmqpTemplate rabbitTemplate(ConnectionFactory connectionFactory) {  
 final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);  
 rabbitTemplate.setMessageConverter(jsonMessageConverter());  
 return rabbitTemplate;  
}

Для того, что бы сообщение передавалось в формате Json а не требовало String необходим конвертер

@Bean  
public MessageConverter jsonMessageConverter() {  
 return new Jackson2JsonMessageConverter();  
}

Все эти три функции хранятся в классе RabbitProducerConfig

1. пишем сервис

Класс RabbitServiceImpl

Для того, что бы передавать сообщения в конкретную очередь с конкретным ключом – необходимо сначала создать эту очередь и привязать к ранее созданному exchange конкретную очередь с ее ключом

Метод createQuene();

Создаем новую ConnectionFactory для того, что бы переопределить ее. Нам не нужна фабрика по умолчанию, нам нужна та, которая сделает то что мы ей скажем.

Устанавливаем пароль и имя пользователя (так как у нас не стандартный набор guest/guest)

ConnectionFactory factory = new ConnectionFactory();  
factory.setPassword(rabbitPassword);  
factory.setUsername(rabbitUserName);

Создаем соединение и канал

try (Connection connection = factory.newConnection();  
 Channel channel = connection.createChannel()) {

Создаем три очереди

channel.queueDeclare(tsofimQueue, false, false, false, null);  
channel.queueDeclare(schoolQueue, false, false, false, null);  
channel.queueDeclare(gymnsatQueue, false, false, false, null);

Теперь нужно привязать очереди к exchange для этого – нужно определить exchange к которому мы будем «байндить»

channel.exchangeDeclare(exchange, *type*, true);

type в данном случае – это direct (то есть сообщение будут передаваться по ключу в свою очередь. Тут может быть любой, необходимый тип из четырех возможных)

И наконец создаем привязки. К каждой очереди по ее ключу

channel.queueBind(tsofimQueue, exchange, tsofimRoutingkey);  
channel.queueBind(schoolQueue, exchange, schoolRoutingkey);  
channel.queueBind(gymnsatQueue, exchange, gymnastRoutingkey);

## Настраиваем сервис GYMNAST.

У сервиса будет 2 задачи.

* Первая – он будет слушать и принимать сообщения от сервиса Scheduler, которые будут ему отправлены (только свои сообщения)
* Вторая – он будет создавать сообщения и отправлять их сервису email.

### Consumer на прием сообщения от Scheduler

1. Application.properties
2. #rabbitMQ  
   spring.rabbitmq.host = localhost  
   spring.rabbitmq.port = 5672  
   spring.rabbitmq.username = user  
   spring.rabbitmq.password = password  
     
     
   gymnsat.key=gym  
   gymnsat.queue=GYMNAST  
   rabbitmq.exchange=atBriut  
     
   email.key=email  
   email.queue=EMAIL  
   email.exchange=Email

!!! Ключ, очередь и exchange должны быть такими же как у scheduler service, который отправляет сообщения.

1. пишем конфигурацию для consumer (class RabbitConsumerConfig)

создаем очередь для получения сообщений (что бы не падать, если вдруг очередь почему то удалена из Rabbit)

@Bean  
 public void createGymnastQueue() {  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setPassword(rabbitPassword);  
 factory.setUsername(rabbitUserName);  
 try (Connection connection = factory.newConnection();  
 Channel channel = connection.createChannel()) {  
 channel.queueDeclare(gymnsatQueue, false, false, false, null);  
 channel.exchangeDeclare(exchange, *type*, true);  
 channel.queueBind(gymnsatQueue, exchange, gymnastRoutingkey);  
 } catch (TimeoutException | IOException e) {  
 e.printStackTrace();  
 }  
}

Создаем очередь для отправки письма

@Bean  
public void createEmailQueue() {  
 ConnectionFactory factory = new ConnectionFactory();  
 factory.setPassword(rabbitPassword);  
 factory.setUsername(rabbitUserName);  
 try (Connection connection = factory.newConnection();  
 Channel channel = connection.createChannel()) {  
 channel.queueDeclare(emailQueue, false, false, false, null);  
 channel.exchangeDeclare(emailExchange, *type*, true);  
 channel.queueBind(emailQueue, emailExchange, emailRoutingkey);  
 } catch (TimeoutException | IOException e) {  
 e.printStackTrace();  
 }  
}

Создаем слушателя

@Override  
@RabbitListener(queues = "${gymnsat.queue}")  
public void receivedMessage(RabbitDto incomingMessage) {  
 String uuidChild = incomingMessage.getUuidChild();  
 gymnastCrawlerService.sendFormToGymnast(uuidChild);  
 *LOGGER*.info(LoggerMessages.*GET\_ATSARAT\_BRIUT* + LoggerMessages.*CHILD* + LoggerMessages.*WITH\_UUID* + uuidChild);  
 System.*out*.println("Recieved Message From RabbitMQ: " + incomingMessage.getUuidChild());  
}

### PRODUCER на отправку сообщения в EMAIL

@Override  
public void sendToEmailService(EmailDto emailDto) {  
 rabbitTemplate.convertAndSend(emailExchange, emailRoutingkey, emailDto);  
}

По такому же принципу пишем остальные сервисы. Tsofim и Schools – на прием и отправку, Email – на прием сообщений

# Eureka service

1. в общий pom добавляем зависимость для спринг бута

<!--for spring cloud-->  
<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>${spring-cloud.version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>  
<!--end for spring cloud-->

и зависимость для клиента эврики (так как все сервисы будут клиентами эврики)

<!-- eureka-client-->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>  
<!-- end of eureka-client-->

1. создаем модуль eureka-service
2. в нем в pom добавляем зависимость для сервера

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>

1. в  application.properties

server.port=8761  
  
#  
##eureka  
eureka.client.fetch-registry=false   
eureka.client.register-with-eureka=false – запрещаем эврике регистрировать себя как клиента (нам это не нужно)  
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka – по этой ссылке все остальные будут конекится к эврике

1. main

@EnableEurekaServer

# Eureka client

Все сервисы – в application.properties добавляем

#eureka  
eureka.client.serviceUrl.defaultZone=${EUREKA\_URL:http://localhost:8761/eureka}  
eureka.instance.instance-id=${spring.application.name}:${spring.application.instance\_id:${random.value}}

Так как раньше мы уже добавили в pom зависимость для клиента эврики, нужно просто в main ->

@EnableEurekaClient

# Gateway - service

## Построение

1. Новый модуль
2. Pom

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  
</dependency>

1. Application-properties

server.port=8011  
spring.application.name=zuul-service  
  
##eureka  
eureka.client.serviceUrl.defaultZone=${EUREKA\_URL:http://localhost:8761/eureka}  
eureka.instance.instance-id=${spring.application.name}:${spring.application.instance\_id:${random.value}}

1. Main

@EnableEurekaClient  
@EnableZuulProxy

## Проверка

1. Start eureka-service
2. Start zuul-service
3. Start user-service
4. Postman

localhost:8011/user-service/monitor

1. Результат – вывод актуатора user-service

Изображение выглядит как текст

Автоматически созданное описание

# Config-server - local

## Server

1. Создаем модуль, в котором backend файловая система

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-server</artifactId>  
</dependency>

1. Application.properties

spring.application.name=config-server  
server.port=8012  
spring.profiles.active=native //нужен для того, что бы переопределить и заставить сервисы брать данные не из git, а с рабочего стола  
spring.cloud.config.server.native.search-locations=file://${user.home}/Desktop/AtsaratBriut/atsarat-briut-microservices/config-server-repo

1. Main

@SpringBootApplication  
@EnableConfigServer

1. Start Eureka-service
2. Start Config-service
3. Должны увидеть в Eureka что конфиг сервер подключился
4. Создаем в папке config-server-repo файл application.properties в этот файл нужно перенести все свойства, которые касаются ВСЕХ сервисов

Например – эврика и актуатор (они для всех сервисов одинаковые)

1. Создаем в папке config-server-repo файл user-service.properties в этот файл нужно перенести все свойства из application.properties из user-service

!!! Название файла в config-server-repo должно совпадать с spring.application.name в user-service!!!

## Client

1. Pom –

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>

1. В сервисе User-service создаем файл bootstrap.properties

spring.cloud.config.uri=http://localhost:8012  
spring.cloud.config.name=user-service

## Проверка

1. Запуск эврика
2. Запуск конфиг

Должны увидеть

main] com.config.ConfigServiceApplication : The following profiles are active: native

1. Запуск user-service

Должны увидеть

Fetching config from server at : <http://localhost:8012>

и

2020-10-18 16:37:35.131 INFO 8707 --- [ main] b.c.PropertySourceBootstrapConfiguration : Located property source: [BootstrapPropertySource {name='bootstrapProperties-file:///Users/smilykq/Desktop/AtsaratBriut/atsarat-briut-microservices/config-server-repo/user-service.properties'}]

То же самое нужно сделать со всеми сервисами.

После этого, одинаковые свойства выносятся в application.properties

# Config-server – git

Конфигурация для гита – в конфиг сервисе:

#git  
#spring.cloud.config.server.git.uri=https://github.com/smilyk/home-accounting-2-repo  
#spring.cloud.config.server.git.username=smilyk  
#spring.cloud.config.server.git.password=Liza200811!  
#spring.cloud.config.server.git.clone-on-start=true

# !!! так как git изменил настройки и теперь в новых проектах нет ветки мастер – необходимо всем сервисам сказать, что они теперь ищут данные с label =main

spring.cloud.config.label=main

# Monitoring config

1. Config-server –

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
</dependency> - нужен для того, что бы отслеживать изменения и перегружать только то, что необходимо  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-monitor</artifactId>  
</dependency> - нужен для того, что бы отслеживать все события и перегружать все сервисы, используя refresh config-server только. Остальные сервисы перегрузят данные сами

1. @Bean  
   GithubPropertyPathNotificationExtractor GithubPropertyPathNotificationExtractor(){  
    return new GithubPropertyPathNotificationExtractor();  
   } – нужен для того, что бы конфиг сервер отслеживал изенения, которые происходят на гите в репозитории свойств
2. Client-server

<!-- spring cloud bus-->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
 </dependency>  
<!-- end of spring cloud bus-->

1. Над тем классом, который нужно будет обновлять – ставим аннотацию

@RefreshScope

### Проверка monitor (т е перезагрузка только конфиг сервера)

* 1. Вызываем

http://localhost:8090/users/v1/ping

* 1. Получаем

Users-Service working 2020-10-21 secret word is: "MyTokenSecret"

* 1. На гите меняем значение tokenSecret на «111»
  2. Вызываем

<http://localhost:8012/monitor>

!!!!!! это актуатор КОНФИГ\_СЕРВЕРА

это POST

в хедерс прописываем

X-Github-Event:push

и в теле метода передаем

|  |
| --- |
| {  "commits":[  {  "modified":["user-service.properties"]  }  ]  }  Где user-service.properties – название того списка свойств, которое мы отслеживаем(то есть которое изменили) |

* 1. В консоли user service видим

Keys refreshed [config.client.version, tokenSecret] – что говорит о том, что tokenSecretдолжен быть перегружен.

* 1. Вызываем

http://localhost:8090/users/v1/ping

* 1. Получаем

Users-Service working 2020-10-21 secret word is: "111" – то есть свойство изменилось!!!

### Проверка monitor (т е перезагрузка только конфиг сервера)

1. Вызываем

http://localhost:8090/users/v1/ping

1. Получаем

Users-Service working 2020-10-21 secret word is: "111"

1. На гите меняем значение tokenSecret на «222»
2. Вызываем

<http://localhost:8090/actuator/bus-refresh>

POST

это актуатор USER\_SERVICE

1. Получаем 204 – No Content
2. В консоли user service видим

Keys refreshed [config.client.version, tokenSecret] – что говорит о том, что tokenSecretдолжен быть перегружен.

1. Вызываем

http://localhost:8090/users/v1/ping

1. Получаем

Users-Service working 2020-10-21 secret word is: "222"

# Hystrix

1. Hystrix будет использоваться в сервисах:
   1. User – не используется
   2. Children не используется
   3. Email не используется
   4. Gymnast – парсер должен получить имя фамилию ребенка и имя фамилию родителя(ответственного лица) +
   5. Scheduler не использует
   6. School – парсер должен получить
   7. Tsofim – парсер должен получить +   
      String uuid = "777";  
      String childFirstNAme = "Liza";  
      String childSecondName = "HREW";  
      String chikldTZ = "111111111";  
      String parentFirstName = "787878";  
      String parentSecondNAme = "90909";  
      String parentTZ = "333333333";

## Пример – Tsofim service

Задача – TsofimCrawlerServiceImpl в методе sendFormToTsofim

Должен получить

String uuid = "777" – приходит в метод;  
String childFirstNAme = "Liza" – из cjild service по cjildUuid;  
String childSecondName = "HREW" из cjild service по cjildUuid;  
String chikldTZ = "111111111" из cjild service по cjildUuid;  
String parentFirstName = "787878" - из cjild service по cjildUuid (получает uuid родителя или uuid ответственного лица – если оно есть);  
String parentSecondNAme = "90909" – из user service по гuuid родителя или ответственого лица;  
String parentTZ = "333333333" - из user service по гuuid родителя или ответственого лица;

### Пишем получение данных из child-service

ֿ„

1. общий пом

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
</dependency>

1. Main -> Tsofim service +

@EnableCircuitBreaker

@EnableFeignClients

1. Properties

feign.hystrix.enabled=true

1. Из Эврики свойств нужно убрать  
   #eureka.client.fetch-registry=false
2. Иначе, эврика не сможет передать обратно данные ребенка
3. Class ChildHystrixDto in service hystrix это ДТО ребенка, которое мы будем получать из child service
4. ChildFallbackFactory это фабрика, которая будет обрабатывать ошибки
5. ChildrenServiceClientFallback – это фабрика, которая будет обрабатывать ошибки
6. ChildServiceClient - получает ДТО ребенка из child server
7. В парсере вызвать метод хистрикс, который получает ребёнка!!!

!!! так как все сервисы отдают не ДТО, а респонс, нужно еще преобразовать полученные данные в ДТО которое нам необходимо.

### Пишем получение данных из user-service

Все то же самое. Если у ребёнка есть человек, который за него отвечает – то работаем. Ним, если нет – то с родителем

# Hystrix monitor

Позволяет смотреть отказоустойчивость сервисов. Разбор на примере Gymnast-service

## Добавление

1. В пом файл сервиса добавляем зависимость для мориторинга

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-netflix-hystrix-dashboard</artifactId>  
</dependency>

1. В общем поме меняем версию спринга на

<spring-cloud.version>Hoxton.SR3</spring-cloud.version>

!!! Hoxton.SR5 не работает с мониторингом

1. В мain gymnast service доюавляем

@EnableHystrix  
@EnableHystrixDashboard

1. В файл свойств подключаем hystrix.stream

management.endpoints.web.exposure.include=hystrix.stream, \*

## Проверка

1. Наш gymnast-service работает на порт = 8087
2. Запускаем <http://localhost:8087/actuator/hystrix.stream> в браузере - видим просто надпист ping:
3. Запускаем в постмене

localhost:8011/gymnast-service/gymnast/v1/parse/cb2ae6c5-07c1-4012-81d5-4e76825d7498

1. Видим:
2. ping:

data: {"type":"HystrixCommand","name":"ChildServiceClient …

1. загружаем в браузере <http://localhost:8087/hystrix>



1. вводим в строку отображения - <http://localhost:8087/actuator/hystrix.stream> и нажимаем Monitor Stream
2. Видим действия

Изображение выглядит как текст

Автоматически созданное описание

# Логирование

## Логирование в файл

1. Общий пом

2. отключить logback :

! — Exclude Spring Boot’s Default Logging →  
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter</artifactId>  
**<exclusions>**  
**<exclusion>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-logging</artifactId>  
</exclusion>**  
**</exclusions>**  
</dependency>

3. подключить log4j2

<! — Add Log4j2 Dependency →

<dependency>  
**<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-log4j2</artifactId>**  
</dependency>

1. В gymnast-service добавляем log4j2.xml
2. 
3. Добавляем текст
4. Запускаем

Изображение выглядит как текст

Автоматически созданное описание

## Логирование микросервисов

Sleuth + Zipkin

Слеус отслеживает связи, Зипкин показывает их графически

У одного запроса во всех микросервисах будет одинаковый Trace + Span

Разработка на примере микросеврвисов gymnast-service + user-service

### Sleuth

1. D pom-global +

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-sleuth-zipkin</artifactId>  
</dependency>

1. Properties общий

|  |
| --- |
| #sleuth |
|  | spring.zipkin.base-url=http://localhost:9411 |
|  | spring.zipkin.sender.type=web |
|  | spring.zipkin.sampler.probably=0.1 |

### Zipkin

1. Start zipkin

docker run -d -p 9411:9411 openzipkin/zipkin

1. Test Zipkin

http://localhost:9411/zipkin/

1. Проверяем. Если послать запрос на gymnast-service должны увидеть логи в которых появятся {service-name}-id, , span-id, true



1. В zipkin увидим все логи



## Логсташ

<https://www.bogotobogo.com/DevOps/Docker/Docker_ELK_7_6_Logstash.php>

# SpringBootAdmin

<https://habr.com/ru/post/479954/>

## Сервер

1. Создаем сервер. Приложение под названием admin
2. Pom

<dependencies>  
 <dependency>  
 <groupId>de.codecentric</groupId>  
 <artifactId>spring-boot-admin-starter-server</artifactId>  
 <version>2.2.2</version>  
 </dependency>  
</dependencies>

!!! Это приложение не может быть модулем микросервисов, так как ему требуется версия SpringBoot 2.2.2

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.2.2.RELEASE</version>  
  
 <relativePath /> <!-- lookup parent from repository -->  
</parent>

1. В классе main

@SpringBootApplication  
@EnableAdminServer  
public class SpringBootAdminServerApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringBootAdminServerApplication.class, args);  
 }  
}

1. Application.properties

spring.application.name=admin-server  
server.port=9999

1. Запускаем

http://localhost:9999/applications

1. Должны увидеть ui спринг бут админа

## Клиент

1. В общий пом добавляем

<!-- spring-boot-admin-->  
 <dependency>  
 <groupId>de.codecentric</groupId>  
 <artifactId>spring-boot-admin-starter-client</artifactId>  
 <version>2.2.2</version>  
 </dependency>

1. В application properties

|  |
| --- |
| #adminserver |
|  | spring.boot.admin.client.url=http://localhost:9999 |

1. Проверяем админ - http://localhost:9999/applications
2. Видим все наши приложения и их характеристики!!!
3. Изображение выглядит как знак, остановка, внешний, объект

   Автоматически созданное описание

# SECURITY

## Подключаем секьюрити

Users-service

In Users-service

1. -pom

<!-- spring-security-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 </dependency>  
<!-- end of spring security-->

1. Start service, postman ->

localhost:8080/users/v1

1. получаем ответ – 401 – Unauthorized
2. Исправляем
3. Класс WebSecurity

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.csrf().disable();  
 // разрешаем запросы по определенному пути  
 http.authorizeRequests().antMatchers("/users/\*\*").permitAll()  
 .anyRequest().authenticated();  
 http.headers().frameOptions().disable();  
}

1. Перезапускаем сервис и запрос и постмен
2. Видим что работает – не нужна авторизация
3. Пробуем запрос, который начинается не с users -> 403 Forbitten (потому что необходима аутентификация)

## Заставляем всех получать токен

@Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http.csrf().disable();  
 // разрешаем запросы по определенному пути  
// http.authorizeRequests().antMatchers("/users/\*\*").permitAll()  
// .anyRequest().authenticated();  
// аутентификация нужна всем  
 http.authorizeRequests().anyRequest().authenticated();  
// http.authorizeRequests().antMatchers("/\*\*").hasIpAddress("localhost:8011/\*\*");  
 http.headers().frameOptions().disable();  
 }

Результат –

localhost:8011/user-service/users/v1 – доступ запрещен (403 код)

localhost:8090/users/v1/ping – доступ запрещен (403 код)

localhost:8090/resp\_pers/v1 – доступ запрещен (403 код)

localhost:8090/users/v1 – доступ запрещен (403 код)

1. Разрешаем создавать пользователя без аутентификации

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.csrf().disable().authorizeRequests()  
 */\*\* роазрешаем создание пользователя без токена\*\*/* .antMatchers(HttpMethod.*POST*, SecurityConstants.*SIGN\_UP\_URL*).permitAll()  
 */\*\* все остальные должны авторизироваться\*\*/* .anyRequest().authenticated();  
 http.headers().frameOptions().disable();  
}

localhost:8011/user-service/users/v1 – доступ разрешен

localhost:8090/users/v1/ping – доступ запрещен (403 код)

localhost:8090/resp\_pers/v1 – доступ запрещен (403 код)

localhost:8090/users/v1 – доступ разрешен

## Token

Идея в том, что при логине пользователь получит свой токен, по которому его будет определять юзер сервис и давать ему доступ ко всему

1. Class LoginRequestModel
2. Создание AutheficationFilter который будет срабатываться каждый раз, когда будет происходить логин

В классе есть метод

@Override  
public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)  
 throws AuthenticationException {  
 try {  
 UserLoginRequestModel creds = new ObjectMapper()  
 .readValue(request.getInputStream(), UserLoginRequestModel.class);  
 return authenticationManager.authenticate(  
 new UsernamePasswordAuthenticationToken(  
 creds.getMainEmail(),  
 creds.getPassword(),  
 new ArrayList<>()  
 )  
 );  
 } catch (IOException e) {  
 throw new RuntimeException();  
 }  
}

Этот метод вызывается каждый раз, когда происходит вызов localhost:8080/login

Метод получает того пользователя, который был отправлен в реквесте (модель пользователя - LoginRequestModel)

После того, как метод получает пользователя, он вызывает метод

@Override  
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {  
 Optional<Users> optionalUser = userRepo.findByMainEmail(email);  
 if(!optionalUser.isPresent())  
 throw new UsernameNotFoundException(email);  
 return new User(optionalUser.get().getMainEmail(), optionalUser.get().getPassword(), new ArrayList<>());  
}

Который достает пользователя с таким же e-mail из базы данных

!!!Для того, что бы получить правильный loadByUser (это метод, который наследуется от класса UserDetails (SpringSecurity), необхзодимо, что бы интерфейст/ UserService наследовал этот класс

public interface UserService extends UserDetailsService

!!!

После того, как пользователь получен – метод attemptAuthentication сравнивает обоих пользователей – того, которого получили из базы данных и того, которого получили из реквесте. Если все ОК и пользователи совпадают – вызывается метод, который создает токен для конкретного пользователя (то есть аутентификация пройдена, возвращаем токен)

@Override  
protected void successfulAuthentication(HttpServletRequest request,  
 HttpServletResponse response,  
 FilterChain chain,  
 Authentication authResult) throws IOException, ServletException {  
 String userName = ((User)authResult.getPrincipal()).getUsername();  
  
 String token = Jwts.*builder*()  
 .setSubject(userName)  
 .setExpiration(new Date(System.*currentTimeMillis*() + SecurityConstants.*EXPIRATION\_TIME*))  
 .signWith(SignatureAlgorithm.*HS512*, SecurityConstants.*TOKEN\_SECRET*)  
 .compact();  
 response.addHeader(SecurityConstants.*HEADER\_STRING*, SecurityConstants.*TOKEN\_PREFIX* + token  
 );  
}

!!! Для того, что бы при аутентификации пользователя не приходила ошибка 403 -> необходимо добавить фильтр в WebSecurity

В методе configure

Добавляем строку

.and().addFilter(new AuthenticationFilter(authenticationManager()));

Весь метод -------

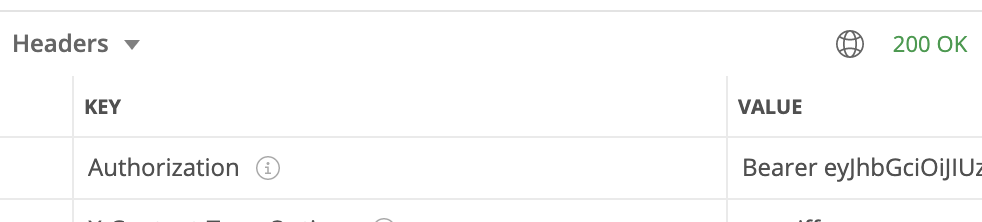
@Override  
protected void configure(HttpSecurity http) throws Exception{  
 http.csrf().disable().authorizeRequests()  
 .antMatchers(HttpMethod.*POST*, SecurityConstants.*SIGN\_UP\_URL*)  
 .permitAll().anyRequest().authenticated()  
 .and().addFilter(new AuthenticationFilter(authenticationManager()));  
}

Кроме этого, тут же необходимо метод

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);  
}

Который будет использовать attemptAuthentication когда ему нужно будет расшифровать пароль, который получен из базы данных.

1. Проверка
2. В постмене localhost:8080/login
3. Результат:
   * 1. 200 ОК
     2. В headers видим token пользователя.



## Добавляем UserUuid в заголовок (header)

1. В метод successfulAuthentication добавляем строчку UserService userService = (UserService) SpringApplicationContext.getBean("userServiceImpl"); для того, что бы мы могли доставать из applicationContex пользователя (UserDto) у этого ДТО есть все необходимоые нам поляю его id, почта и все, что мы можем захотеть использовать в своем приложении.
2. Для того, что бы получить UserDto пришем метод в UserServiceImpl  
   @Override  
   public UserDto getUser(String email) {  
    Optional<Users> optionalUser = userRepo.findByMainEmail(email);  
    if(!optionalUser.isPresent())  
    throw new UsernameNotFoundException(email);  
    UserDto returnValue = new UserDto();  
    BeanUtils.*copyProperties*(optionalUser.get(), returnValue);  
    return returnValue;  
   }
3. В этом методе мы получаем пользователя из БД и добавляем его в Bean спринга, что бы в последствии можно было доставать его из любого места в приложении.
4. В метод successfulAuthentication добавляем строчку
5. UserDto userDto = userService.getUser(userName);
6. И наконец добавляем в header запроса этот идентификатор пользователя
7. response.addHeader("UserId", userDto.getUuidUser());
8. Теперь, если мы отправим запрос – получим токен и UserId далее по этому UserId мы сможем удалять, обновлять или получать пользователя, давать ему какие то привилегии или наоборот забирать их у него.

Изображение выглядит как стол

Автоматически созданное описание

## Изменяем url для login

Если мы не хотим логиниться по пути localhost:8080/login

1. */\*\*  
    \* логин пользователя теперь по ссылке .../users/login  
    \*/*protected AuthenticationFilter getAuthenticationFilter() throws Exception {  
    final AuthenticationFilter filter = new AuthenticationFilter(authenticationManager());  
    filter.setFilterProcessesUrl("/users/login");  
    return filter;  
   }

Тут мы прописываем новый путь

1. Для того, что бы новый путь не требовал авторизации – меняем в configure
2. .and().addFilter(getAuthenticationFilter());
3. Теперь метод выглядит так:

@Override  
protected void configure(HttpSecurity http) throws Exception{  
 http.csrf().disable().authorizeRequests()  
 .antMatchers(HttpMethod.*POST*, SecurityConstants.*SIGN\_UP\_URL*)  
 .permitAll().anyRequest().authenticated()  
 .and().addFilter(getAuthenticationFilter());  
}

1. Проверяем – работает!

## Авторизация

Делаем так, что бы только авторизованые пользователи могли получить доступ к приложению user-service. Используем токен для проверки авторизации

1. AuthorizationFilter
2. Каждый раз, когда происходит запрос к любой конечной точке – вызывается этот фильтр. В нем есть доступ к хеалерам, то есть мы можем достать токен и проверить его.

@Override  
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws IOException, ServletException {  
  
 String header = request.getHeader(SecurityConstants.*HEADER\_STRING*);  
 if(header==null || header.startsWith(SecurityConstants.*TOKEN\_PREFIX*)){  
 chain.doFilter(request, response);  
 return;  
 }  
 UsernamePasswordAuthenticationToken authentication = getAuthentication(request);  
 SecurityContextHolder.*getContext*().setAuthentication(authentication);  
 chain.doFilter(request, response);  
}

1. Реализовываем метод
2. private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {  
    String token = request.getHeader(SecurityConstants.*HEADER\_STRING*);  
     
    if (token != null) {  
     
    token = token.replace(SecurityConstants.*TOKEN\_PREFIX*, "");  
     
    String user = Jwts.*parser*()  
    .setSigningKey( SecurityConstants.*TOKEN\_SECRET*)  
    .parseClaimsJws( token )  
    .getBody()  
    .getSubject();  
     
    if (user != null) {  
    return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());  
    }  
    return null;  
    }  
    return null;  
   }
3. Этот метод проверяем верный ли токен
4. В классе WebSecurity дописываем
5. .addFilter(new AuthorizationFilter(authenticationManager()));
6. Вест метод –
7. @Override  
   protected void configure(HttpSecurity http) throws Exception{  
    http.csrf().disable().authorizeRequests()  
    .antMatchers(HttpMethod.*POST*, SecurityConstants.*SIGN\_UP\_URL*)  
    .permitAll().anyRequest().authenticated()  
    .and()  
    */\*\*  
    \* фильтр для всех обьектов, которые проходят через процедуру аутентификации  
    \*/* .addFilter(getAuthenticationFilter())  
    */\*\*  
    \* только тот, кто вошел в пприложение, может менять что то в нем  
    \*/* .addFilter(new AuthorizationFilter(authenticationManager()));
8. Проверяем.
   1. Логинимся
   2. Получаем токен
   3. localhost:8080/users/v1
   4. in Headers adding token
   5. 200 OK

Изображение выглядит как текст

Автоматически созданное описание

1. Убираем сохранение в кеше токена
2. В классе WebSecurity to config adding
3. .sessionManagement()  
   .sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);