

Arquiteturas e Integração de Sistemas

Miguel António Barbosa da Silva 18829
André Filipe Dias Cardoso 18848

MESTRADO EM ENGENHARIA INFORMÁTICA
ESCOLA SUPERIOR DE TECNOLOGIA
INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

Identificação dos alunos

Miguel António Barbosa da Silva 18829

André Filipe Dias Cardoso 18848

Mestrado em Engenharia Informática

Conteúdo

1	Introdução	5
1.1	Arquitetura global do sistema	5
1.2	Tecnologias a utilizar	6
1.3	BPMN	6
1.4	Setup Containers	7
2	Serviços	9
2.1	Base de Dados	9
2.2	Backend	9
2.3	RabbitMQ	10
2.4	Pilha ELK	10
2.5	Ficheiro Docker Compose	10
2.5.1	Base de dados	10
2.5.2	RabbitMQ	11
2.5.3	Pilha ELK	11
2.5.4	Backend	12
2.5.5	Rede	13
3	Sistema de Eventos	15
3.1	Endpoints Implementados	15
3.2	Documentação	16
3.3	Autenticação	16
3.4	Certificado RSA	17
3.5	Segurança dos endpoints	17
4	Proteção contra Ataques de Força Bruta	19
4.1	<i>LoginAttemptService</i>	19
4.2	<i>Listeners</i>	19
5	Logging	21
5.1	Ficheiro LogBack no spring boot	21
5.2	Tentativa de integração com RabbitMQ	22
5.3	Ver Logs na WebApp Kibana	24
6	Integração de Serviço Externo	25

1. Introdução

No mundo atual em que cada vez mais a evolução tecnológica se demonstra veloz e impactante, são necessárias soluções que permitam o seu acompanhamento. Como tal, o recurso a soluções cloud tornam-se cada vez mais atrativas tendo em conta a escalabilidade e eficiência de consumo de recursos associada, especialmente ao utilizar tecnologias como **Kubernetes**.

O objetivo principal é desenvolver um gestor de autenticação com *logs*, que seja distribuível criando uma imagem *docker*.

1.1 Arquitetura global do sistema

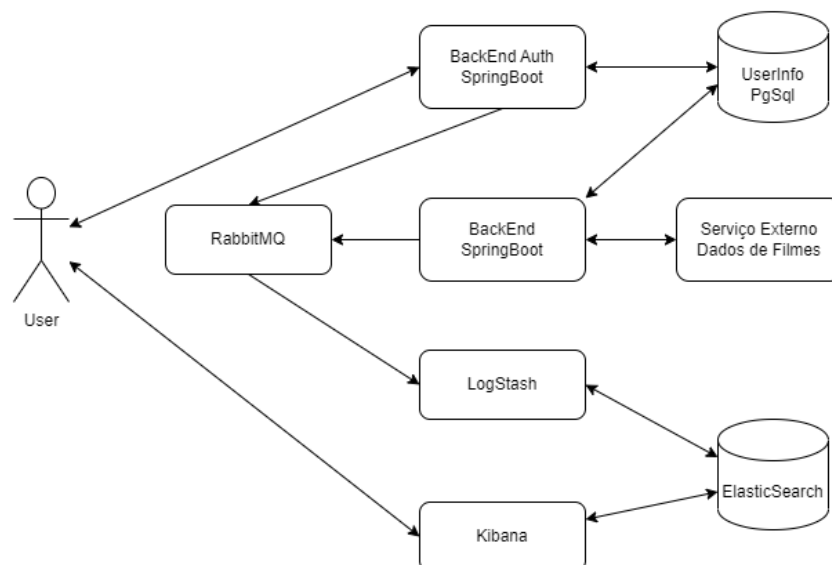


Figura 1.1: Diagrama dos micro serviços do sistema

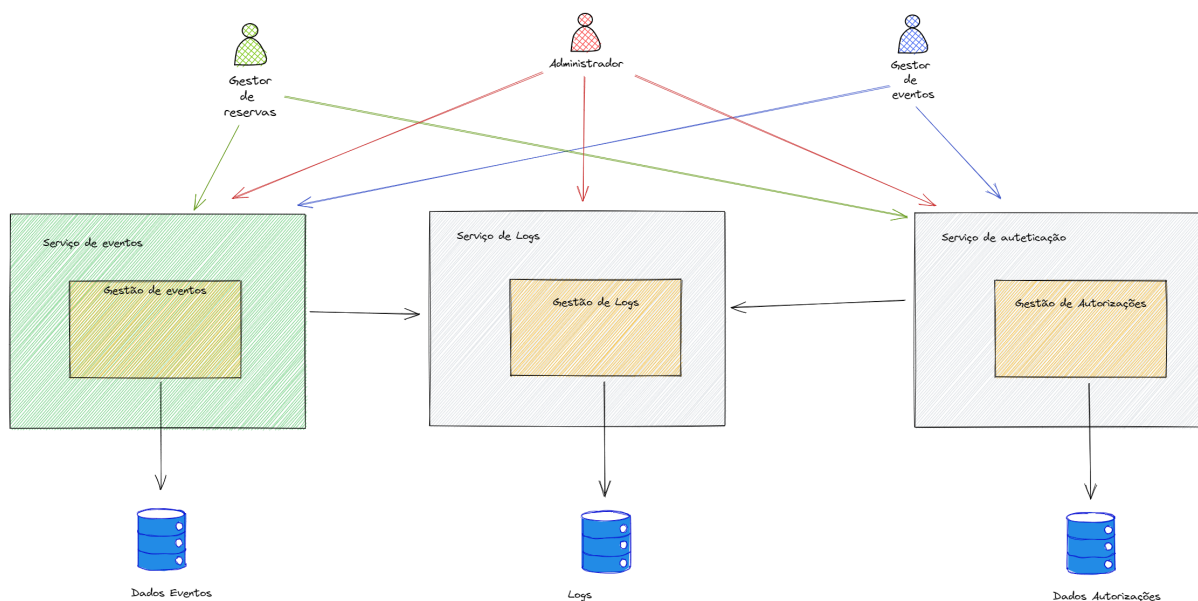


Figura 1.2: Diagrama da arquitetura do sistema

1.2 Tecnologias a utilizar

- **Springboot** - BackEnd
- **PgSql** - Database (User Info)
- **RabbitMQ** - Message Broker
- **LogStash** - Logger
- **ElasticSearch** - Logger Storage
- **Kibana** - Logger View

1.3 BPMN

Com o objetivo de ilustrar o modelo de negócio a implementar na nossa aplicação foi criado o seguinte diagrama que explicita o processo de criação de um evento e o processo de reserva de um evento através de interfaces gráficas, o que difere da implementação dos processos nos serviços.

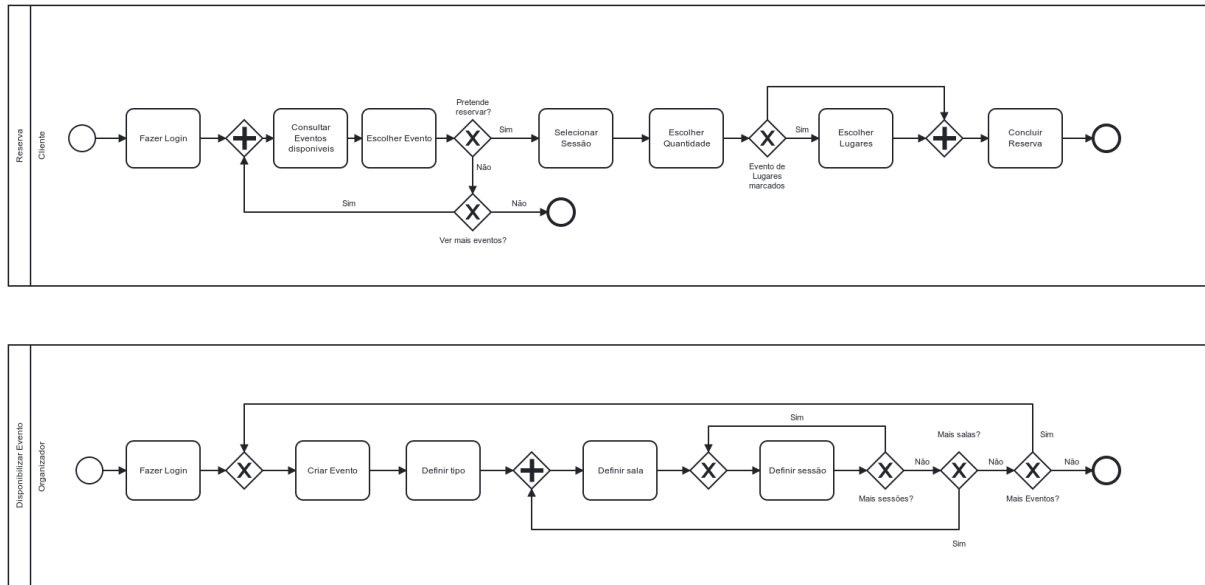


Figura 1.3: Diagrama BPMN

1.4 Setup Containers

Para a compilação do projeto são necessárias as seguintes variáveis de ambiente:

SPRING_DATASOURCE_URL
 jdbc:postgresql://localhost:5432/appreservas?currentSchema=auth
LOGSTASH_URL
 jdbc:postgresql://localhost:5432/appreservas?currentSchema=auth

Utilizando RabbitMQ é também necessária a seguinte variável de ambiente:

RABBIT_MQ_IP
 localhost

Para compilar o gestor de autenticação é necessário correr o comando da listagem 1.1 na raiz do projeto e os comandos 1.2 e 1.3 na pasta que contém o ‘docker-compose.yml’.

```
mvn clean install
```

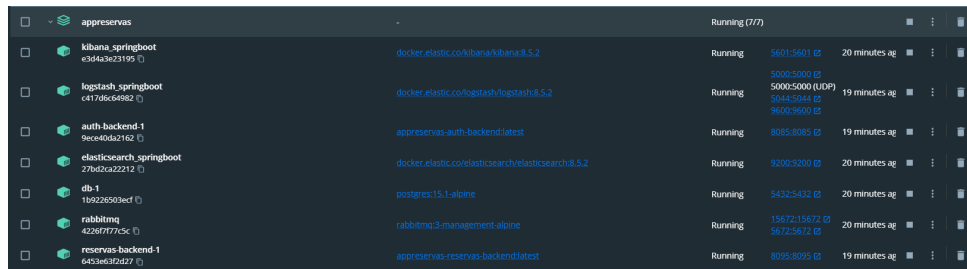
Listagem 1.1: Compilar Projeto

```
docker compose build
```

Listagem 1.2: Docker-Compose build

```
docker compose up
```

Listagem 1.3: Docker-Compose up










appreservas		Running (7/7)	
 kibana_springboot e3d4a3e23199	docker.elastic.co/kibana/kibana:8.5.2	Running	56015601 20 minutes ag
 logstash_springboot c417d6c64982	docker.elastic.co/logstash/logstash:8.5.2	Running	5000:5000 (UDP) 5044:5044 9600:9600 19 minutes ag
 auth-backend-1 9ece40da2162	appreservas-auth-backend:latest	Running	8085:8085 19 minutes ag
 elasticsearch_springboot 27b6fca22212	docker.elastic.co/elasticsearch/elasticsearch:8.5.2	Running	9200:9200 20 minutes ag
 db-1 1b226503edf	postgres:15.1-alpine	Running	5432:5432 20 minutes ag
 rabbitmq 4326f7f775c	rabbitmq:3-management-alpine	Running	15672:15672 5672:5672 20 minutes ag
 reservas-backend-1 6453a63f9d27	appreservas-reservas-backend:latest	Running	8095:8095 19 minutes ag

Figura 1.4: Containers da stack do Docker Compose a correr

2. Serviços

2.1 Base de Dados

O sistema de gestão de base de dados escolhido foi *PostgreSQL*. O *PostgreSQL* é um sistema de gestão de base de dados relacionais *open-source*, atualmente considerada a mais avançada no seu segmento de mercado.

De forma a cobrir as nossas necessidades criamos uma nova base de dados, com o nome *appreservas*, que contém dois esquemas com um total de nove tabelas. O digrama de entidade relação representado abaixo contem todas as tabelas presentes na base de dados *appreservas*.

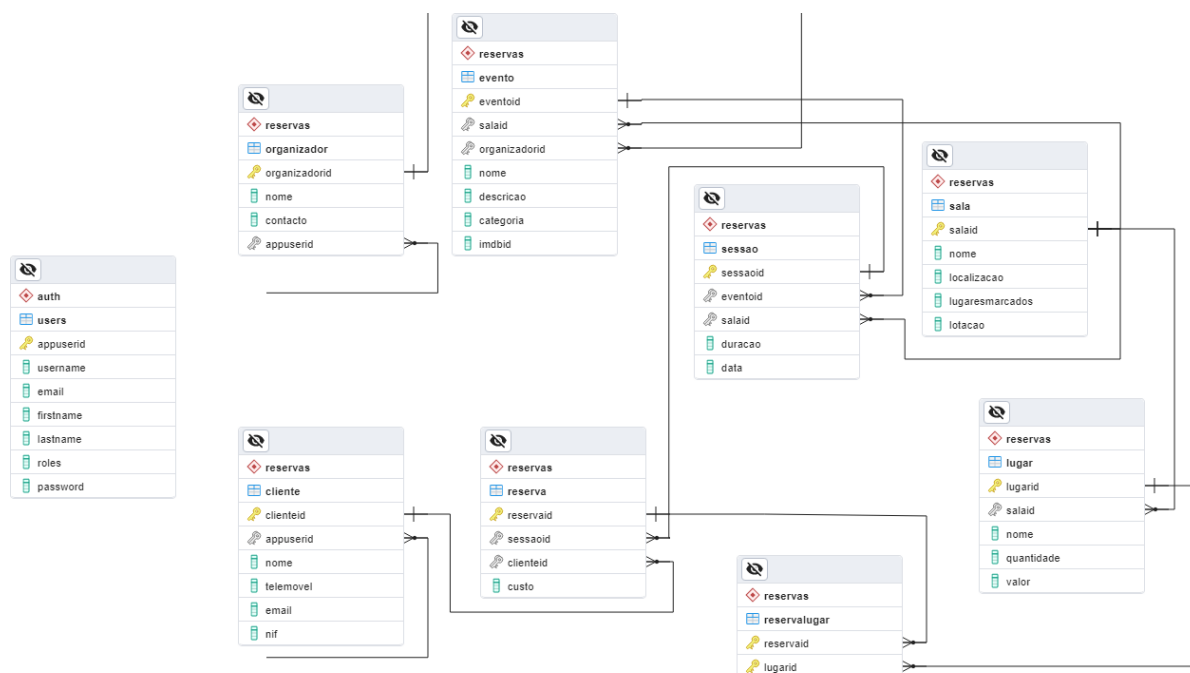


Figura 2.1: Diagrama entidade relação

2.2 Backend

O *backend* da nossa aplicação baseia-se em dois componentes: um gestor de autenticação e o servidor da aplicação. Estes funcionam independentemente o que pode ser um fator que aumenta a resiliência do eco-sistema.

Ambos são serviços REST API, desenvolvidos em **Springboot**, uma framework de JAVA gerida pela ferramenta Maven cujo objetivo passa por facilitar o processo de compilação, permitir um processo de compilação uniformizado assim como informação de qualidade acerca do projeto e ainda encorajar boas práticas de desenvolvimento. A ‘pegada’ do Maven no projeto passa pelo ficheiro **pom.xml** onde se encontram informações do projeto em questão especialmente as dependências necessárias para o desenvolvimento em Springboot.

No capítulo 3 (Sistema de Eventos) é descrito com mais detalhe as funcionalidades implementadas em cada um dos serviços.

2.3 RabbitMQ

O RabbitMQ foi *Message Broker* escolhido para fazer a comunicação entre o serviço de *backend* e a pilha ELK.

2.4 Pilha ELK

Fizemos recurso a pilha ELK para a gestão dos *logs* do serviço de gestão de identidades. A escolha da utilização desta pilha completa passa pela facilidade de integrações dos seus elementos com o *backend*, aliado ao facto que a pilha disponibiliza diversas ferramentas para a visualização e gestão dos *logs*. A pilha ELK conta com três elementos:

Elasticsearch mecanismo de busca e análise

Logstash pipeline de processamento de dados do lado do servidor que faz a ingestão de dados a partir de inúmeras fontes simultaneamente, transforma-os e envia-os para o Elasticsearch

Kibana permite visualizar dados presentes no Elasticsearch recorrendo a diagramas e gráficos, entre outros métodos de visualização.

2.5 Ficheiro Docker Compose

Com o objetivo de agilizar o processo de implementação de todos os serviços necessários à aplicação criamos um ficheiro Docker Compose onde se encontram descritos os parâmetros necessários a cada serviço de forma a funcionar e comunicar corretamente com os restantes.

2.5.1 Base de dados

Para implementar a base de dados criamos um serviço que tem como base a imagem do PostgreSQL presente no DockerHub. A versão da imagem utilizada é a 15.1-alpine, versão mais recente à data baseada no sistema operativo Linux Alpine. De forma a definir as credências de acesso ao SGBD fizemos recurso às variáveis de ambiente *POSTGRES_USER* e *POSTGRES_PASSWORD* onde é indicado o utilizador e password a usar para administrador. Os dois volumes indicam onde devem ser guardados os dados e qual ficheiro utilizar para a criação do esquema durante a inicialização. É ainda feito o mapeamento da porta '5432', porta predefinida dos SGBD Postgres, de forma a poder acedida fora do *container*.

```
services:
  db:
    image: postgres:15.1-alpine
    restart: always
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
    ports:
      - '5432:5432'
    volumes:
      - ./db-data:/var/lib/postgresql/data
      - # copy the sql script to create tables
        ./bd.sql:/docker-entrypoint-initdb.d/create_tables.sql
    networks:
      - auth
```

Figura 2.2: Serviço da base de dados no ficheiro Docker Compose

2.5.2 RabbitMQ

Para implementar o *Message Broker* criamos um serviço que tem como base a imagem do RabbitMQ presente no DockerHub. A versão da imagem utilizada é a 3-management-alpine, versão mais recente à data baseada no sistema operativo Linux Alpine. É também feito o mapeamento das portas '5672' e '15672' de forma a poderem acedidas fora do *container*.

```
rabbitmq:
  image: rabbitmq:3-management-alpine
  container_name: 'rabbitmq'
  ports:
    - 5672:5672
    - 15672:15672
  volumes:
    - ~/.docker-conf/rabbitmq/data:/var/lib/rabbitmq/
    - ~/.docker-conf/rabbitmq/log:/var/log/rabbitmq
  networks:
    - appreservas
```

Figura 2.3: Serviço do RabbitMQ no ficheiro Docker Compose

2.5.3 Pilha ELK

Para os serviços da pilha ELK foram criadas três entradas no ficheiro, uma por cada elemento da pilha. Todos os serviços são baseados em imagens, presentes no registo

de imagens da *elastic.co*, criadora da pilha em questão. Para o serviço de Logstash e Elasticsearch são definidos volumes que permite guardar os dados dos *logs*, assim como ler ficheiros de configuração na pasta do projeto. Além do já referido, é também feito recurso a várias variáveis de ambiente e mapeamento de portas de forma a configurar corretamente a pilha na comunicação com diferentes serviços.

```
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.3.3
    container_name: elasticsearch_springboot
    environment:
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
      - "discovery.type=single-node"
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
    volumes:
      - elasticsearch_data:/usr/share/elasticsearch/data
    networks:
      - auth
  kibana:
    image: docker.elastic.co/kibana/kibana:8.3.3
    container_name: kibana_springboot
    ports:
      - "5601:5601"
    environment:
      ELASTICSEARCH_URL: http://elasticsearch:9200
      ELASTICSEARCH_HOSTS: '["http://elasticsearch:9200"]'
    depends_on:
      - elasticsearch
    networks:
      - auth
  logstash:
    image: docker.elastic.co/logstash/logstash:8.3.3
    container_name: logstash_springboot
    volumes:
      - ./logstash/config/logstash.yml:/usr/share/logstash/config/logstash.yml:ro
      - ./logstash/pipeline:/usr/share/logstash/pipeline:ro
    ports:
      - "5044:5044"
      - "5000:5000/tcp"
      - "5000:5000/udp"
      - "9600:9600"
    environment:
      LS_JAVA_OPTS: "-Xmx256m -Xms256m"
    networks:
      - auth
    depends_on:
      - elasticsearch
```

Figura 2.4: Serviços da pilha ELK no Docker Compose

2.5.4 Backend

Nos serviços de *backend*, ao contrário dos restantes serviços, não é feita referência a uma imagem por ser necessário fazer o *build* da mesma. Para esse efeito é definido o contexto para a criação da imagem que contém o caminho para o *dockerfile* pretendido. As variáveis de ambiente utilizadas neste serviço contêm referência aos serviços definidos anteriormente de forma a indicar corretamente os endereços a aceder para comunicar com o LogStash e Base de dados. Sem a correta referência aos restantes serviços o mesmo não funciona, visto isso é necessário indicar que este serviço tem dependências através da utilização do *depends_on*.

```
services:
  auth-backend:
    build:
      context: ./auth
    ports:
      - '8085:8085'
    environment:
      -
    SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/appreservas?
    currentSchema=auth
      - LOGSTASH_URL=logstash:5000
    depends_on:
      - db
      - logstash
      - kibana
      - elasticsearch
    networks:
      - auth
```

Figura 2.5: Serviço de Backend no Docker Compose

Dockerfile

Com o objetivo de criar uma imagem para o nosso serviço de *backend* tivemos de definir um Dockerfile onde são descritos os passos a criação da imagem. Como base da nossa imagem é usado a imagem *openjdk* com a tag *17-jdk-slim*, o uso desta versão devesse a criar o ambiente com a versão de Java correta para a nossa aplicação. Além da definição da imagem base é também exportada a porta 8085 e definida a pasta onde a aplicação vai ficar no interior da imagem. Por fim é copiado o binário da nossa aplicação e defendido o *entrypoint* associado ao mesmo.

```
FROM openjdk:17-jdk-slim
WORKDIR app
EXPOSE 8085
COPY target/auth-*.jar exec.jar
ENTRYPOINT ["java", "-jar", "exec.jar"]
```

Figura 2.6: Dockerfile para o serviço de backend

2.5.5 Rede

Com o objetivo de permitir a comunicação entre os serviços foi necessária a definição de uma rede ao foi dada o nome de *auth* no modo *bridge*. A criação rede permite aos serviços referenciar os restantes através do seu nome. Sem a presença da rede seria necessário indicar o ip de cada serviço para os mesmos comunicarem entre si. O destas referencias pode ser visto nas variáveis de ambiente dos serviços *kibana* e *auth-backend* onde é referenciado os serviços *elasticsearch*, *db* e *logstash*.

3. Sistema de Eventos

3.1 Endpoints Implementados

De forma a dar resposta as funções base de um autenticador foi criado um conjunto de endpoints enumerados abaixo.

/login - Endpoint que utiliza o método POST do protocolo HTTP e tem como objetivo gerar um token de autenticação. Como argumento recebe um *username*, do tipo string, e a *password*, também do tipo string. Os argumentos têm de ser enviados pelo corpo da mensagem fazendo recurso à codificação JSON. O valor de retorno passa por um objeto JSON com o token JWT, caso seja autenticado com sucesso, ou resposta *BAD_REQUEST*, caso não seja possível autenticar o utilizador.

/register - Endpoint que utiliza o método POST do protocolo HTTP e permite o registo de um novo utilizador na plataforma. Como argumento recebe *username*, *password*, *password*, *firstname*, *lastname*, *lastname* e *email*. Todos os campos do tipo string. Os argumentos têm de ser enviados pelo corpo da mensagem fazendo recurso à codificação JSON. O valor de retorno pode ser uma mensagem com o código 201 a indicar que foi criado com sucesso, ou resposta *BAD_REQUEST*, caso não seja possível registar o utilizador.

/forgotpassword - Endpoint que utiliza o método POST do protocolo HTTP e permite ao utilizador gerar uma nova *password* para a sua conta. Como argumento recebe um *email* e o *username*, ambos do tipo string. Os argumentos têm de ser enviados pelo corpo da mensagem fazendo recurso à codificação JSON. O valor de retorno passa uma mensagem a informar que a nova *password* foi enviada para *email* do utilizador, caso os dados coincidam com algum utilizador registado, ou resposta *BAD_REQUEST*, caso não seja possível detetar o utilizador em questão.

/changepassword - Endpoint que utiliza o método POST do protocolo HTTP e permite ao utilizador mudar a sua *password* de acesso. Como argumento recebe *currentpassword*, do tipo string, e *newpassword*, também do tipo string. Os argumentos têm de ser enviados pelo corpo da mensagem fazendo recurso à codificação JSON. A rota necessita ser acedida por um utilizador já autenticado, para isso é necessário enviar um token JWT válido nos *headers* da mensagem. O valor de retorno passa por uma mensagem que informa o sucesso da ação, caso a mudança de password seja feita com sucesso, ou resposta *BAD_REQUEST*, caso não seja possível alterar a password.

Já o serviço de eventos necessita de vários endpoints para suportar o CRUD do negócio. Assim, com ajuda da ferramenta Speedment foi possível gerar, através da conexão à base de dados, todo o serviço REST que processava as ações de CRUD necessárias. Cada tabela tem o seu endpoint que respeita o padrão '/nometabela' através da qual são executadas todas as ações identificando o método GET e LIST para leitura, POST para inserção, PUT para atualização e DELETE para eliminação.

Precisamos apenas de identificar as tabelas relevantes e como desejávamos utilizar e restringir a informação de cada tabela e coluna. Toda essa informação é então armazenada

num ficheiro JSON que pode ser a qualquer momento reutilizado para gerar novamente esse CRUD ou até fazer modificações.

```
mvn speedment:tool;
```

Listagem 3.1: Abrir ferramenta de configuração Speedment

```
mvn speedment:generate;
```

Listagem 3.2: Gerar código CRUD com o Speedment

3.2 Documentação

No que toca à documentação, ela é gerada pelo próprio Springboot utilizando a dependência *springdoc-openapi-ui* e pode ser acedida nos *endpoints* que se seguem:

`/swagger-ui.html` - UI de Swagger com informação dos endpoints

`/api-docs` - JSON com informação dos endpoints

A documentação é referente para cada serviço individualmente, pelo que segundo as configurações finais para o serviço de autenticação acede-se pelo domínio ‘localhost:8085’ e ‘localhost:8095’ para o serviço de eventos. É perceptível que em alguns casos o endpoint de SwaggerUI não é capaz de reconhecer imediatamente o endpoint da documentação openAPI (‘/api-docs’) pelo que é preciso passar esse endpoint manualmente no topo da página.

3.3 Autenticação

O serviço de autenticação faz uso de um conjunto de credenciais (*username* e *password*) de modo a gerar um *token* **temporário** que permite distinguir os utilizadores e os seus respetivos acessos.

Este *token* é do tipo **JWT** e contém informação do utilizador, tal como:

scope - Autorização do utilizador (equivalente ao *role*)

username - Nome único do utilizador (relevante para início de sessão)

user_id - ID único do utilizador (relevante para transações do sistema)

Além disto, tem também informação para quem emitiu o token, a data de emissão e a data de expiração de validade do token.

3.4 Certificado RSA

Com o objetivo de tornar o *token* de autenticação mais seguro, geramos um certificado RSA com chave publica e chave privada. O JWT é gerado com base na chave privada e pode ser validado com recurso à chave publica.

Para gerar o certificado fizemos recurso à biblioteca OpenSSL que disponibiliza um conjunto de ferramentas de propósito criptográfico geral que permitem tornar comunicações mais seguras. Um novo certificado RSA pode ser obtido com recurso ao comando:

```
openssl genrsa -out keypair.pem 2048
```

Listagem 3.3: Comando para um novo certificado RSA de 2048 bits

Uma vez obtido o *keypair* podemos gerar as chaves publicas e privadas com recurso aos comandos:

```
rsa -in keypair.pem -pubout -out public.pem
```

Listagem 3.4: Comando para gerar a chave pública

```
pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in keypair.pem  
↪ -out private.pem
```

Listagem 3.5: Comando para gerar a chave privada sobre o formato PKCS#8

3.5 Segurança dos endpoints

O controlo de acessos é feito a partir do método `securityFilterChain` onde é retornada uma configuração da gestão de autorizações feita através do padrão *BuilderDesign*. Para este gestor em específico as configurações estão como se segue:

<code>csrf</code>	disabled
<code>authorizeRequest</code>	...
<code>oauth2ResourceServer</code>	JWT
<code>sessionCreationPolicy</code>	Stateless
<code>userDetailsService</code>	Serviço Personalizado

Quanto aos acessos às rotas específicas (**`authorizeRequest`**):

Autenticação - “/changepassword”, “/home”

Anónimo - restantes rotas (inclui rotas de documentação OpenAPI)

4. Proteção contra Ataques de Força Bruta

De modo a impedir este tipo de ataques, é necessário implementar um serviço capaz de bloquear um determinado utilizador que tente (sem sucesso) iniciar sessão de forma abusiva.

Uma das informações mais importantes para uma proteção deste género é o IP do utilizador que faz o *request*. Tendo acesso ao IP em questão, podemos estabelecer um número limite de acessos até este ser bloqueado durante um determinado período de tempo.

4.1 *LoginAttemptService*

Este serviço define o comportamento do autenticador nos casos relevantes, fazendo uso de uma *cache* onde armazena um dicionário com o IP do *request* e o número de tentativas falhadas como um par chave-valor respetivamente.

Login com sucesso - `loginSucceeded`

Uma tentativa bem sucedida de início de sessão provoca a remoção da chave do dicionário.

Login sem sucesso - `loginFailed`

Ao falhar a iniciar sessão primeiramente tentamos aceder à respetiva chave do dicionário, e atualiza-se o valor de tentativas falhadas.

Endereço bloqueado - `isBlocked`

Ao receber um acesso à API, o servidor verifica se o endereço atual se encontra na lista de tentativas falhadas e se este já excedeu o número de tentativas falhadas permitidas; em caso positivo, o pedido é abortado e retorna uma mensagem de erro.

4.2 *Listeners*

Um *listener* é um componente que atua no momento que deteta a publicação de um evento específico. No contexto da Autenticação, são utilizados dois eventos: **AuthenticationFailureBadCredentialsEvent** e **AuthenticationSuccessEvent**.

São criados dois componentes que implementam as duas interfaces referidas com o intuito de chamar as ações de início de sessão do serviço *LoginAttemptService*.

Events

Para que o fluxo da aplicação decorra conforme o desejado resta apenas a publicação de eventos. É então definido o **AuthenticationEvent** que estende o **ApplicationEvent** de modo a que seja compatível com ambos os eventos usados.

Utilizando um **ApplicationEventPublisher** é possível publicar o evento em questão e possibilitar o *flow* desejado para a prevenção de *brute-force*.

5. Logging

5.1 Ficheiro LogBack no spring boot

De forma comunicar os logs para a pilha ELK foi necessário configurar um ficheiro de logback no backend em Spring Boot. Lá indicamos os parâmetros para efetuar a conexão com o LogStash. A conexão é feita via protocolo TCP para a porta 5000 do container do Log Stash

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/base.
    ↪ xml"/>
  <springProperty name="LOGSTASH_HOST" source="logstash.host"/>
  <appender name="logstash" class="net.logstash.logback.appender.
    ↪ LogstashTcpSocketAppender">
    <destination>${LOGSTASH_HOST}</destination>
    <encoder class="net.logstash.logback.encoder.
      ↪ LoggingEventCompositeJsonEncoder">
      <providers>
        <mdc />
        <context />
        <logLevel />
        <loggerName />
        <pattern>
          <pattern>
            {
              "app": "auth-log"
            }
          </pattern>
        </pattern>
        <threadName />
        <message />
        <logstashMarkers />
        <stackTrace />
      </providers>
    </encoder>
  </appender>
  <root level="info">
    <appender-ref ref="logstash" />
  </root>
</configuration>
```

Listagem 5.1: Configurações de LogBack

Além de configurar o ficheiro de *logback* é necessário configurar o LogStash para isso é feito recurso ao ficheiro *logstash.conf* onde indicamos os ‘inputs’ e ‘outputs’ dos dados.

```
input {
  tcp{
    port => 5000
    codec => json
  }
}

output {
```

```

    elasticsearch {
        hosts => "elasticsearch:9200"
        index => "springboot-%{app}"
    }
}

```

Listagem 5.2: Configurações de logstash.conf

5.2 Tentativa de integração com RabbitMQ

De forma a utilizar um *message broker* para a comunicação com os serviços foi feita a tentativa de configuração do Spring Boot e Logstash para receberem os logs através de AMQT, protocolo utilizado para a comunicação por RabbitMQ. A configuração do Backend foi realizada com sucesso, após configurar o ficheiro de LogBack de acordo os parâmetros do RabbitMQ.

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="AMQP" class="org.springframework.amqp.rabbit.
        ↪ logback.AmqpAppender">
        <layout>
            <pattern><![CDATA[ %d %p %t [%c] - <%m-%n ]]></pattern>
        </layout>
        <exchangeName>EnterpriseApplicationLog</exchangeName>
        <host>${RABBIT_MQ_IP}</host>
        <port>5672</port>
        <username>guest</username>
        <password>guest</password>
        <exchangeType>topic</exchangeType>
        <applicationId>AmqpAppenderTest</applicationId>
        <routingKeyPattern>logs-test</routingKeyPattern>
        <generateId>true</generateId>
        <charset>UTF-8</charset>
        <durable>false</durable>
        <deliveryMode>NON_PERSISTENT</deliveryMode>
    </appender>

    <root level="info">
        <appender-ref ref="AMQP" />
    </root>
</configuration>

```

Listagem 5.3: Configurações de LogBack para AMQT

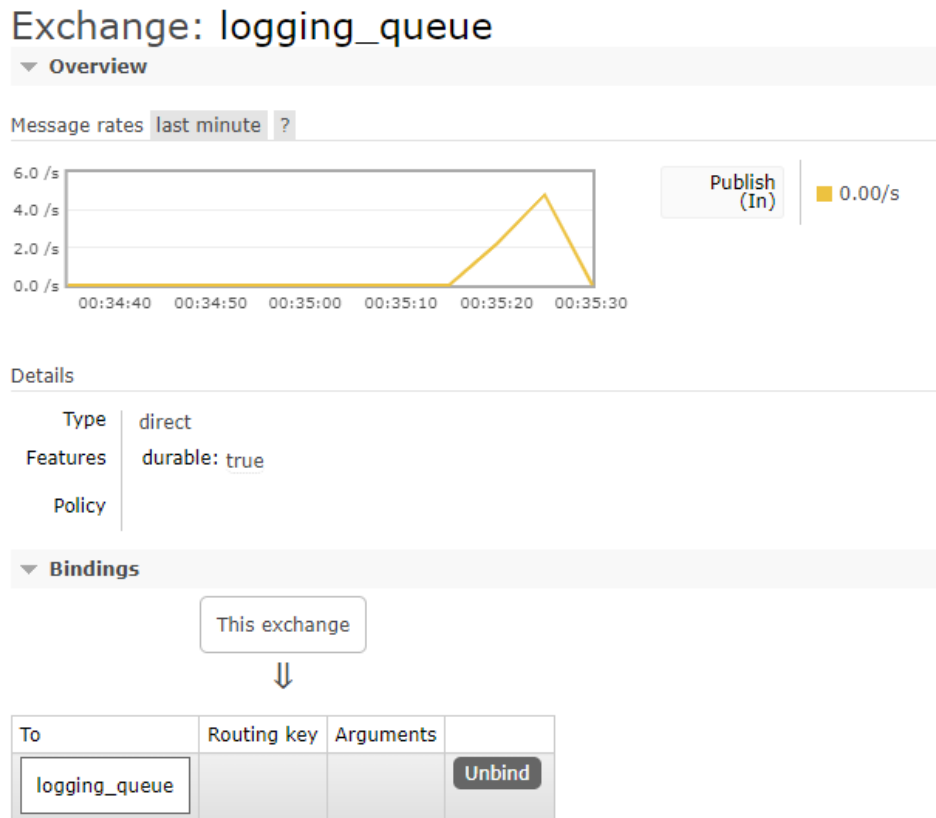


Figura 5.1: Exchange com RabbitMQ

Na configuração do RabbitMQ deparamonos com problemas relacionados com o redirecionamento das mensagens recebidas e o encaminhamento para as *queues*. As mensagens enviadas pelo Backend chegam todas ao *exchange* do RabbitMQ, no entanto o encaminhamento para as *queues* ligadas não se sucede.

```
input {
  rabbitmq {
    id => "rabbitmq_logs"
    # connect to rabbit
    host => "rabbitmq"
    port => 5672
    vhost => "/"
    # Create a new queue
    queue => "ApplicationLog"
    durable => "false"
    exchange => "EnterpriseApplicationLog"
    key => "logs-test"
    # No ack will boost your perf
    ack => false
  }
}

output {
  elasticsearch {
    hosts => "elasticsearch:9200"
    index => "single_index"
  }
}
```

Listagem 5.4: Configurações de logstash.conf para RabbitMQ

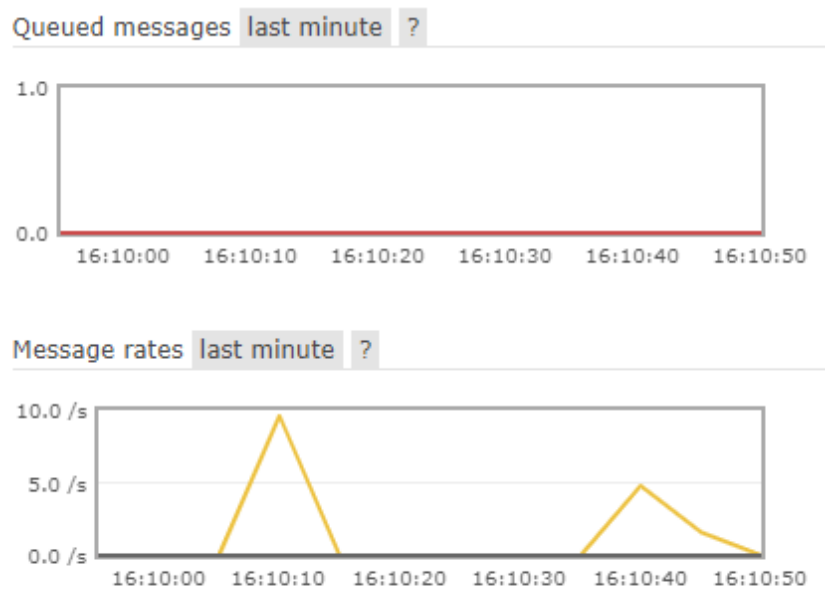


Figura 5.2: Mensagens recebidas e não encaminhadas para as queues

5.3 Ver Logs na WebApp Kibana

Para visualizar os logs na WebApp Kibana foi necessário criar uma *DataView* associada ao *Index* dos Logs recebidos pelo LogStash. A criação *DataView* pode ser feita na secção de *StackManagement*. Uma vez criada a data podemos visualizar os Logs presentes no *index* associado à mesma na página de *Discover* nas *Analytics*.

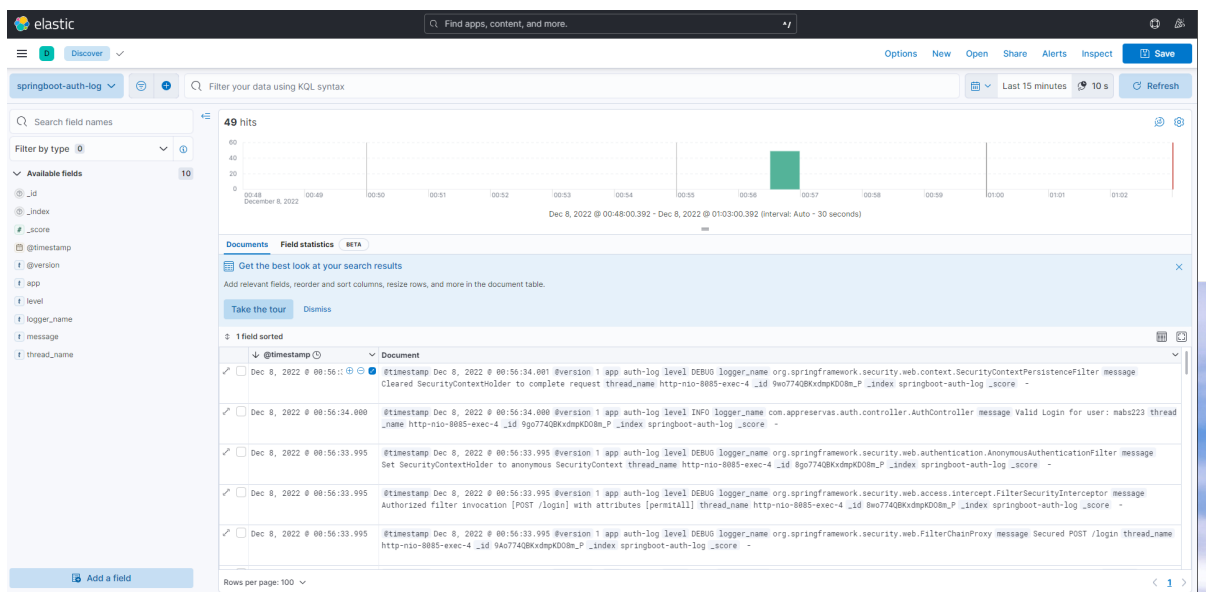


Figura 5.3: DataView no Kibana

6. Integração de Serviço Externo

Para fornecer mais informação aos utilizadores em eventos do tipo filme fizemos a integração de um serviço externo com o nome de Movie Details criado por GoodMovies. O serviço em questão está presente na RapidAPI, uma plataforma que disponibiliza publicamente varia API que ajudam desenvolvedores a tornar os seus serviços mais completos.

A escolha da API Movie Details deveu-se ao facto que a API disponibiliza um endpoint, que pode ser acedido com o método GET do protocolo HTTP, que devolve várias informações sobre um programa de televisão, serie ou filme dando como argumento o seu identificador na plataforma IMDB. Inicialmente tínhamos como ideia utilizar um serviço oficial disponibilizado pela plataforma do IMDB, no entanto, não encontramos forma de utilizar o serviço para obter dados acerca de um evento por ser necessário utilizar serviços da AWS para aceder aos dados, sendo o método atual mais simples.

Para fazer uso do serviço foi necessário fazer um registo na plataforma RapidAPI de forma a obter uma *key* de acesso para ao serviço em questão. Para testar o uso do serviço fizemos recurso à ferramenta Postman construindo um Request em que providenciando os dados necessários.

- url do endpoint: `https://movie-details1.p.rapidapi.com/imdb_api/movie`
- header com a chave de acesso: X-RapidAPI-Key
`c3a1fc6601msh79a6cdec3c2c55p13e6b5jsn853f7aef5e54`
- header com o host: X-RapidAPI-Host `movie-details1.p.rapidapi.com`
- argumento do identificador IMDB: `id=tt1375666`

Os dados que obtemos de volta, caso o request conclua com sucesso, são:

- | | | |
|----------------|------------------|---------------|
| • id | • imdb_type | • description |
| • title | • runtime | • aka |
| • rating | • genres | • version |
| • rating_count | • countries | • image |
| • release_year | • languages | • imdb_date |
| • popularity | • director_names | • actors |
| | • creator_names | |

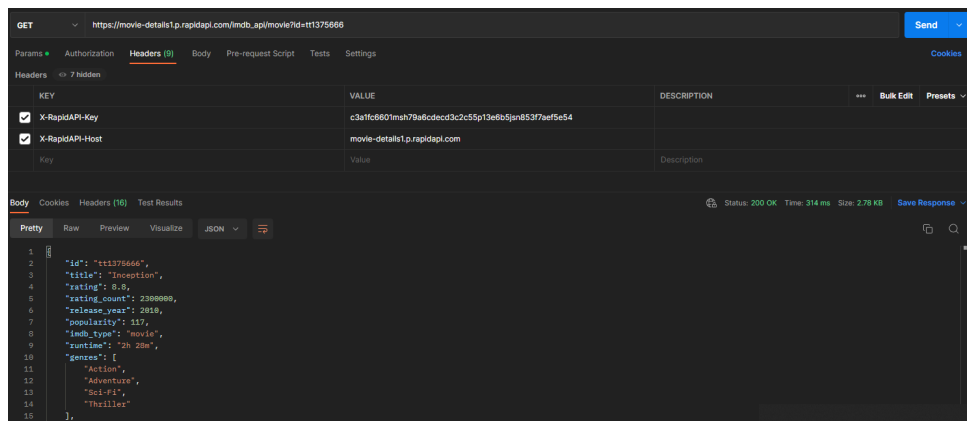


Figura 6.1: Exemplo de request no Postman

O serviço foi integrado no Backend SpringBoot de gestão de eventos e reservar através da criação de um *Service* que formula a request descrita anteriormente.

```
public class MovieService{

    private final RestTemplate restTemplate;

    @Autowired
    public MovieService() {
        this.restTemplate = new RestTemplate();
    }

    public String getMoviePlainJSON(String movieId) {
        HttpHeaders headers = new HttpHeaders();

        headers.set("X-RapidAPI-Key", "
        ↪ c3a1fc6601msh79a6cdec3c2c55p13e6b5jsn853f7aef5e54");
        headers.set("X-RapidAPI-Host", "movie-details1.p.rapidapi.com"
        ↪ );

        HttpEntity<String> entity = new HttpEntity<>(headers);

        String url = "https://movie-details1.p.rapidapi.com/imdb_api/
        ↪ movie?id=" + movieId;
        try{
            var response = this.restTemplate.exchange(url, HttpMethod.
            ↪ GET, entity, String.class);
            if(response.getStatusCode() == HttpStatus.OK) {
                return response.getBody();
            } else {
                return null;
            }
        }catch(Exception e){
            return null;
        }
    }
}
```

Listagem 6.1: MovieService implementado

7. Conclusão

Realizar este trabalho com ferramentas completamente desconhecidas à data foi um desafio interessante e elucidante no que toca à versatilidade de programação em JAVA quando utilizado em conjunto com frameworks como Maven e Springboot.

Isto ampliou-se quando aliado ao foco deste trabalho: a *containerização*. Num mundo em que a evolução dos sistemas roda em volta da migração para a *cloud*, a preparação e conhecimentos adquiridos na criação e orquestração de containers tornam-se um bom ponto de partida.

No futuro seria possível e interessante a integração de uma API Gateway e acima de tudo um *message broker* como o RabbitMQ com sucesso.