API REST - Api-Starter

Pulse

Preparado por:
Helvio Oliveira Carvalho - São Luís/MA

Arquitetura de Software

A Arquitetura da *API Rest,* chamada *"api-starter"* foi desenvolvida para fornecer serviços de cadastro de produtos, manutenção de carrinho virtual e geração documento fiscal simples.

Algumas tecnologias e conceitos utilizados no projeto:

- JAVA 8
- Spring Boot 2
- MySQL 8.0
- Flyway
  - Versionamento do Banco
- JWT Token
- RESTFul
- Spring Security
  - Segurança
- Insomnia
-
  - Teste das requisições

Para a “api-starter” foram criados os seguintes principais recursos:

- [POST] http://localhost:8080/oauth/token
  - Autenticação do usuário.
- [GET] http://localhost:8080/v1/public/products
  - Listagem de todos os produtos
- [POST] http://localhost:8080/v1/private/carts
  - Cria o carrinho para o usuário logado.
  - Parâmetros:
    - totalPrice: NULL
    - codeUserDB: Código do Usuário
    - codePayment: NULL
- [POST] http://localhost:8080/v1/private/carts/product
  - Adiciona um produto ao carrinho.
  - Parâmetros:
    - codeProduct: Código do Produto
    - codeCart: Código do Carrinho
    - quantity: Quantidade do Produto
- [POST] http://localhost:8080/v1/private/carts/close
  - Fecha o carrinho.
  - Parâmetros:
    - codeCart: Código do Carrinho
    - codePayment: Código da Forma da Pagamento
    - codeShipping: Código da forma de entrega
    - totalPrice: Valor total da forma de entrega
    - deliveryTime: Tempo de entrega
- [POST] http://localhost:8080/v1/public/carts/report/{code}
  - Gera o documento fiscal.
  - Parâmetros:
    - {code}: Código do Carrinho

[POST] http://localhost:8080/oauth/token

A autenticação do usuário é feita através dos componentes do Spring Security. Resumidamente, o usuário informa seu usuário e senha e o sistema retorna um access token no padrão do JWT.

De uma forma mais detalhada, a requisição de autenticação foi desenvolvida para validar tanto as credenciais do usuário quanto as credenciais da aplicação cliente que está acessando a API Rest. Segue abaixo a configuração da autenticação do cliente:

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
    clients.inMemory()
        .withClient("apistarter-mobile-flutter")
        .secret(passwordEncoder.encode("@p1st@rt3r-m0b1l3-flutt3r"))
        .scopes("/playlists")
        .authorizedGrantTypes("password")
        .accessTokenValiditySeconds(1800)
    .and()
        .withClient("apistater-web-angular")
        .secret(passwordEncoder.encode("@p1st@t3r-w3b-@ngul@r"))
        .scopes("/playlists", "/statistic")
        .authorizedGrantTypes("password")
        .accessTokenValiditySeconds(1800);

}
```

Na imagem acima, os clientes foram criados em memória. Os dois possuem usuário (método 'withClient(....)') e senha (método secret( )).
É definido também, a duração da validade do Token para cada cliente, além de definir o escopo (scopes(....))de atuação de cada cliente. Na imagem a seguir, o usuário que autenticar nos diferentes clientes, poderá ter acesso a recursos diferentes, independente das permissões vinculadas ao usuário. O objetivo é trazer mais segurança para a API identificando a origem das requisições, além de limitar o acesso de uma gama de aplicações, se assim for o desejo do desenvolvedor.

CartController.java

```java
package com.pulse.apistarter.controller;

import java.io.FileNotFoundException;

@RestController
public class CartController {

    @Autowired
    private CartService cartService;

    @PostMapping("/v1/private/carts")
    @PreAuthorize("hasAuthority('ROLE_CART') and #oauth2.hasScope('/carts')")
    public ResponseEntity<Long> create(@Valid @RequestBody CartDTO cartDTO){
        Long cartCode = cartService.createCart(cartDTO);
        return ResponseEntity.ok(cartCode);
    }

    @PostMapping("/v1/private/carts/product")
    @PreAuthorize("hasAuthority('ROLE_CART') and #oauth2.hasScope('/carts')")
    public ResponseEntity<Long> addProduct(@Valid @RequestBody CartProductDTO cartProductDTO){
        Long cartCode = cartService.addProduct(cartProductDTO);
        return ResponseEntity.ok(cartCode);
    }

    @DeleteMapping("/v1/private/carts/product/{code}")
    @PreAuthorize("hasAuthority('ROLE_CART') and #oauth2.hasScope('/carts')")
    public ResponseEntity<Object> withdrawProduct(@NotNull @PathVariable(value = "code") Long cartProductCode){
        HttpStatus status = cartService.withdrawProduct(cartProductCode);
        return new ResponseEntity<>(status);
    }

    @PostMapping("/v1/private/carts/close")
    @PreAuthorize("hasAuthority('ROLE_CART') and #oauth2.hasScope('/carts')")
    public ResponseEntity<ReceiptDTO> closeCart(@Valid @RequestBody CloseCartDTO closeCartDTO) throws JRException, FileNotFoundException{
        ReceiptDTO receiptDTO = cartService.closeCart(closeCartDTO);

        return ResponseEntity.ok(receiptDTO);
    }
```

A validação é feita através da anotação @PreAuthorize anotada acima do método. O acesso ao método é permitido apenas quando se satifaz as duas condições. A primeira condição "hasAuthority(ROLE_CART)" verifica se o usuário, através de suas próprias credenciais, possui permissão para acessar o método. Já a segunda condição "#oauth2.hasScope(.....)" verifica se as credenciais enviadas pelo cliente possuem o escope de acesso. O usuário possuindo a "ROLE_CART" e acessando do cliente "apistarter-web-angular" teria acesso apenas ao método "create", já a partir de outro cliente não teria acesso aos métodos apresentados.

Stateless e Autenticações da Requisições

```
@Override
public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/v1/public/carts/**").permitAll()
        .antMatchers("/v1/public/products").permitAll()
        .anyRequest().authenticated()
        .and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .csrf().disable();
}

@Override
public void configure(ResourceServerSecurityConfigurer resources) throws Exception {
    resources.stateless(true);
}
```

Na imagem acima, é definido que todas as requisições feitas para a API deverão ser autenticadas, com exceção da requisição de autenticação, que por definição do próprio Spring Security, não fica impedida de consumo. Ainda, é definido que a política de sessão será STATELESS.

Solução CORS

```java
package com.ingaia.apistarter.cors;

import java.io.IOException;

@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
public class CorsFilter implements Filter{

    @Autowired
    private ApistarterProperty apistarterProperty;

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
            throws IOException, ServletException {

        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) resp;

        response.setHeader("Access-Control-Allow-Origin", apistarterProperty.getAllowedOrigin());
        response.setHeader("Access-Control-Allow-Credentials", "true");

        if ("OPTIONS".equals(request.getMethod()) && apistarterProperty.getAllowedOrigin().equals(request.getHeader("Origin"))) {
            response.setHeader("Access-Control-Allow-Methods", "POST, GET, DELETE, PUT, OPTIONS");
            response.setHeader("Access-Control-Allow-Headers", "Authorization, Content-Type, Accept");
            response.setHeader("Access-Control-Max-Age", "3600");

            response.setStatus(HttpServletResponse.SC_OK);
        } else {
            chain.doFilter(req, resp);
        }

    }

    @Override
    public void destroy() {
    }

    @Override
    public void init(FilterConfig arg0) throws ServletException {
    }
```

A classe CORSFilter.class foi criada para resolver o problema de requisições JavaScript quando a origem é diferente do host, ou seja, se o cliente estiver em outro domínio/servidor (protocolo diferente - HTTP, HTTPS - endereço e/ou porta diferente). No applications.properties, no atributo "apistarter.allowed-origin" é possível configurar a origem permitida para ser permitida pelo CORSFilter.class.

Tratamento de erros com Exception

```java
package com.ingaia.apistarter.exceptionhandler;

import java.util.ArrayList;

@ControllerAdvice
public class ApiStarterExceptionHandler extends ResponseEntityExceptionHandler{

    @Autowired
    private MessageSource messageSource;

    @Override
    protected ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        String mensagemUsuario = messageSource.getMessage("mensagem.invalida", null, LocaleContextHolder.getLocale());
        String mensagemDesenvolvedor = ex.getCause() != null ? ex.getCause().toString() : ex.toString();
        List<Erro> erros = Arrays.asList(new Erro(mensagemUsuario, mensagemDesenvolvedor));
        return handleExceptionInternal(ex, erros, headers, HttpStatus.BAD_REQUEST, request);
    }

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        List<Erro> erros = criarListaDeErros(ex.getBindingResult());
        return handleExceptionInternal(ex, erros, headers, HttpStatus.BAD_REQUEST, request);
    }


    @ExceptionHandler({UnauthorizedAccessException.class})
    public ResponseEntity<Object> handleUnauthorizedAccessException(UnauthorizedAccessException ex,
            WebRequest request){

        String mensagemUsuario = messageSource.getMessage("acesso.nao.autorizado", null, LocaleContextHolder.getLocale());
        String mensagemDesenvolvedor = ex.toString();
        List<Erro> erros = Arrays.asList(new Erro(mensagemUsuario, mensagemDesenvolvedor));
        return handleExceptionInternal(ex, erros, new HttpHeaders(), HttpStatus.UNAUTHORIZED, request);
    }
```

Os erros provenientes das regras de negócio do sistema são tratados através exceptions e da classe "ApiStarterExceptionHandler.class". Se ocorrer algum erro durante uma determinada rotina, é retornado uma exception que está declarada nesta classe, e devolve-se para o cliente mensagens de erros e o statusCode.

CACHE

```java
package com.ingaia.apistarter;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
@EnableConfigurationProperties(ApistarterProperty.class)
@EnableCaching
public class ApiStarterApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiStarterApplication.class, args);
    }

}
```

```java
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    @Cacheable("returnProducts")
    @Transactional(readOnly = true)
    public List<Product> returnProducts() {
        return productRepository.findAll();
    }
```

A política de cache foi habilitada utilizando a anotação @EnableCaching na classe "ApiStarterApplication.class" e habilitada apenas no método "returnProducts", que retorna os produtos. O uso do cache objetiva a melhora do desempenho da API.

AUTENTICAÇÃO - TESTE API REST

- [POST] http://localhost:8080/oauth/token
  - CLIENT ANGULAR (SCOPE “/products /users /carts”)
- FORM URL ENCODED
  - client = apistater-web-angular
  - username = joao@gmail.com ("ROLE_PRODUCT / ROLE_USER / ROLE_CART ")
  - password = Maria
  - grant_type = password
- BASIC AUTH
  - username = apistater-web-angular
  - password = @p1st@t3r-w3b-@ngul@r
- HEADER
  - Content-Type = application/x-www-form-urlencoded

OUTRO USUÁRIO:

- username = joao@gmail.com ("ROLE_USER / ROLE_CART ")
- password = Maria