

PWAds

Technical Report

Milestone #1 - PEDWN 2022

Daniel Castro
8160445

Índice

1. Introdução	3
2. Contextualização	3
3. Funcionalidades Implementadas	4
3.1.RESTful API	4
3.2.Progressive Web App	4
4. Paradigmas, princípios e padrões de design	5
4.1.Progressive Web App	5
4.2.Paradigmas	5
4.3.Design Patterns	5
4.4.SOLID Principles	6
5. Aplicação dos paradigmas, princípios e padrões de design na implementação de funcionalidades	7
5.1.RESTful API	7
5.2.Progressive Web App	17
6. Lighthouse	25

1. Introdução

No âmbito da unidade curricular de PEDWM, do curso de Mestrado de Engenharia de Software, foi proposto desenvolver uma *progressive web app* (PWA) e uma aplicação de RESTful.

O propósito deste relatório é apresentar a solução desenvolvida e demonstrar que as suas funcionalidades foram desenvolvidas utilizando o conhecimento, frameworks, padrões e técnicas apresentadas pela unidade curricular.

2. Contextualização

Uma *Progressive Web Apps* (PWA) é um website que parece e se comporta como uma aplicação nativa. As PWAs foram criadas para aproveitar os recursos nativos do seu dispositivo, se exigir que o utilizador recorrer a uma loja de aplicações.

O termo PWA foi introduzido pelo desenvolvedor do Chrome Alex Russel e o designer Frances Berriman num artigo, em 2015. A ideia de PWAs não foi inventada pela Google, mas foi o Steve Jobs que introduziu o conceito pela primeira vez, em 2007 na apresentação do iPhone. Na época, parecia natural que aplicações externas aumentassem a popularidade do dispositivo.

O objetivo deste projeto foi criar um website de anúncios no formato de uma aplicação PWA, seguindo os padrões de desenvolvimento web introduzidos pela unidade curricular.

Simultaneamente, foi desenvolvido com recurso à Spring Framework um web service para servir a PWA, sobre a forma de uma RESTful API. A Spring é uma framework open source que tem como propósito facilitar o desenvolvimento de aplicações Java.

Nas secções seguintes serão apresentadas as funcionalidades desenvolvidas, os paradigmas, princípios e padrões introduzidas pela unidade curricular, e por fim a demonstração da aplicação destes na implementação das funcionalidades.

3. Funcionalidades Implementadas

Nesta secção serão listadas as funcionalidades desenvolvidas para ambas as componentes da aplicação de anúncios.

3.1. RESTful API

3.1.1. Anúncios

- 3.1.1.1. Obter todos os anúncios;
- 3.1.1.2. Obter, criar, editar, apagar um anúncio;

3.1.2. Auditoria

- 3.1.2.1. Criar registo de todos pedidos realizados à RESTful API;

3.1.3. Mensagens

- 3.1.3.1. Obter todas as mensagens;
- 3.1.3.2. Obter, criar, editar, apagar uma mensagem;
- 3.1.3.3. Obter as mensagens enviadas por um utilizador;
- 3.1.3.4. Obter as mensagens enviada para um utilizador;

3.1.4. Segurança

- i. Registo do Utilizador;
- ii. Autenticação do Utilizador;
- iii. Estabelecer e garantir acesso de acordo com permissões de Utilizador;

3.2. Progressive Web App

- 3.2.1. Registo e autenticação do utilizador;
- 3.2.2. Listar, criar, editar e apagar anúncios;
- 3.2.3. Listar mensagens enviadas e recebidas de um utilizador;
- 3.2.4. Enviar mensagens para o proprietário de um anuncio;
- 3.2.5. Notificar o utilizador que recebeu uma mensagem.

4. Paradigmas, princípios e padrões de design

Nesta secção serão apresentados os paradigmas, princípios e padrões de design apresentados na unidade curricular.

4.1. Progressive Web App

Uma web app para ser considerada uma PWA tem de incorporar um conjunto de princípios, estes são:

- 4.1.1. Discoverable
- 4.1.2. Installability
- 4.1.3. Linkability
- 4.1.4. Network independence
- 4.1.5. Progressive enhancement support
- 4.1.6. Re-engageability
- 4.1.7. Responsiveness
- 4.1.8. Secure

4.2. Paradigmas

Os paradigmas e as linguagens de programação ditam como o código do software é escrito. Os paradigmas apresentados na unidade curricular foram:

- 4.2.1. Programação Orientada a Objetos (OOP)
- 4.2.2. Programação Funcional (FP)
- 4.2.3. Programação Orientada a Aspectos (AOP)

4.3. Design Patterns

Os design patterns podem ser utilizadas em diferentes linguagens de programação para solucionar problemas comuns no desenvolvimento de software.

Todavia os design patterns introduzidos pela unidade curricular de PEDWN têm o objetivo de incrementar a reusabilidade de componentes no paradigma de programação orientada a objetos.

4.3.1. General Design Patterns

- 4.3.1.1. Singleton

- 4.3.1.2. Strategy

- 4.3.1.3. Observer

- 4.3.1.4. Decorator

- 4.3.1.5. Factory

4.3.2. IOC Container Design Patterns

- 4.3.2.1. Inversion of Control Principle (IoC)

- 4.3.2.2. Dependency Inversion Principle (DIP)

- 4.3.2.3. Dependency Injection Pattern (DI)

- 4.3.2.4. Inversion of Control Containers (IoC Containers)

4.4. SOLID Principles

- 4.4.1. Single-responsibility;

- 4.4.2. Open-closed;

- 4.4.3. Liskov substitution;

- 4.4.4. Interface Segregation Principle;

- 4.4.5. Dependency Inversion Principle;

4.5. HATEOAS Principle

O acrónimo HATEOAS vem de "Hypermedia As the Engine Of Application State" e o termo "hypermedia" no seu nome já dá uma ideia de como este principio funciona numa aplicação RESTful. Ao ser implementado, a API passa a fornecer links que indicarão aos clientes como navegar através dos seus recursos.

5. Aplicação dos paradigmas, princípios e padrões de design na implementação de funcionalidades

Nesta secção serão apresentados os paradigmas, princípios e padrões de design que foram utilizados e não utilizados na implementação das componentes desenvolvidas para a web app de anúncios.

5.1. RESTful API

Nesta subsecção serão apresentados os paradigmas, princípios e padrões de design que foram utilizados na implementação das funcionalidades e configurações da RESTful API.

5.1.1. Paradigmas

5.1.1.1. Programação Orientada a Objetos (OOP)

A RESTful API foi desenvolvida com recurso à framework Spring, que utiliza como linguagem de programação Java. Desta forma, foi determinado o uso do paradigma de programação orientada a objetos.

5.1.1.2. Programação Funcional (FP)

O paradigma de programação funcional foi utilizado com o uso das funções lambda e streams Java. As funções lambda e streams foram maioritariamente utilizadas nos controladores de anúncios, mensagens e utilizadores, para iterar objetos obtidos a partir dos seus repositórios.

Na Fig. 1 é possível observar a implementação de um endpoint do tipo GET do módulo de anúncios, que retorna a lista de todos os anúncios. Na implementação deste endpoint foram utilizadas as funções lambda e streams Java para realizar o map e transformar a lista de anúncios do repositório de anúncios.

```
@GetMapping("/all")
public CollectionModel<EntityModel<Ad>> all() {
    List<EntityModel<Ad>> ads = this.repository.findAll().stream().map(this.assembler::toModel).collect(Collectors.toList());
    return CollectionModel.of(ads, linkTo(methodOn(AdController.class).all()).withSelfRel());
}
```

1. Exemplo do Paradigma de Programação Funcional

5.1.1.3. Programação Orientada a Aspectos (AOP)

O paradigma de programação orientada a aspectos foi utilizado nos módulos de Auditoria, Segurança e Mensagens. O desenvolvimento dos componentes deste paradigma foi realizado com o suporte à framework AspectJ.

1. Auditoria

O módulo de Auditoria tem uma única funcionalidade, que é registrar as informações de todos os pedidos realizados na RESTful API. Esta funcionalidade foi implementada utilizando os conceitos do paradigma AOP.

Para tal finalidade foi criado um aspecto denominado **AuditAspect** que regista os tempos de execução e informações de pedidos realizados a todos os endpoints que pertençam aos módulos de anúncios ou mensagens.

No aspecto foi criado um pointcut, representado na Fig. 2, que aponta para a execução de todos os métodos que se encontrem dentro de um package chamado “**controller**” e que este(s) package(s) esteja(m) dentro do package “**components**”.

```
@Pointcut("execution(* com.example.springangularadsapp.components...controller...*(..))")
void allMethodsInAComponentsControllerPackage() {
}
```

2. Implementação do pointcut “allMethodsInAComponentsControllerPackage” do aspecto AuditAspect

Por fim, foi criado um advice que cuja responsabilidade é registrar as informações e o tempo de execução do(s) métodos(s) apontado(s) pelo pointcut especificado. Observável na Fig. 3, que este advice é executado com a anotação **@Around** da framework AspectJ porque é necessário iniciar a cronometragem antes e terminar depois da execução do método.

```
@Around(value = "allMethodsInAComponentsControllerPackage()")
public Object endpointAudit(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
    long start = System.currentTimeMillis();
    Object result = proceedingJoinPoint.proceed();
    long elapsedTime = System.currentTimeMillis() - start;
    String requestId = UUID.randomUUID().toString();
    String className = proceedingJoinPoint.getSignature().getDeclaringTypeName();
    String methodName = proceedingJoinPoint.getSignature().getName();
    String apiName = className + "." + methodName;
    HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.currentRequestAttributes()).getRequest();

    StatsLog statsLog = new StatsLog(requestId, request.getHeader("host"), request.getMethod(), request.getRequestURI(), apiName, Arrays.toString(request.getParameterNames()));
    this.repository.save(statsLog);

    return result;
}
```

3. Implementação do advice “endpoinAudit” do aspecto AudiAspect

2. Segurança

O módulo de segurança é composta pelas componentes de autenticação e autorização.

A componente de autenticação foi criada utilizando o módulo Spring Security da framework Spring, que permitiu implementar o protocolo de autenticação JWT.

No entanto para a componente de autenticação foi implementada utilizando conceitos de AOP. Com tal finalidade, foram impostos três níveis de autorização, sendo estes User, Moderator e Admin. No propósito de impor estes três níveis de autorização foram criadas três anotações (**UserAccess**, **ModeratorAccess** e **AdminAccess**) e o aspeto **AccessValidation** que contém o respetivo pointcut e advice para cada uma destas.

Nas Fig. 4, 5 e 6 visualizam-se a anotação, o pointcut e o advice criados para impor o nível o autorização de UserAccess. O pointcut implementado na Fig.4 aponta para todos os métodos que possuam a anotação “**UserAccess**”. O advice para verificar o nível de autorização requer o uso da anotação **@Around** da framework AspectJ, porque existe a necessidade de impedir a execução do método caso o utilizador não possua a autorização ao lançar uma exceção. No cenário de possuir os privilégio de utilizador utiliza-se o método `proceed` da Class `ProceedingJoinPoint` para resumir a execução do método.

```
public @interface UserAccess {  
}
```

4. Anotação UserAccess

```
@Pointcut("@annotation(com.example.springangularadsapp.security.authorization.annotation.UserAccess)")  
public void methodsWithUserAccessAnnotation() {  
}
```

5. Implementação pointcut `methodsWithUserAccessAnnotation` no aspeto `AccessValidation`

```
@Around(value = "methodsWithUserAccessAnnotation()")  
public Object checkUserPrivileges(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {  
    HttpServletRequest request = ((ServletRequestAttributes) Objects.requireNonNull(RequestContextHolder.getRequestAttributes())).getRequest();  
    if (request.isUserInRole(s: "ROLE_USER") || request.isUserInRole(s: "ROLE_MODERATOR") || request.isUserInRole(s: "ROLE_ADMIN"))  
        return proceedingJoinPoint.proceed();  
    else throw new UnauthorizedAccessException(ERole.ROLE_USER);  
}
```

6. Implementação advice `checkUserPrivileges` no aspetos `AccessValidation`

Além destes foi criado um advice único responsável por tratar o lançamento de todas as exceções da classe `UnauthorizedAccessExceptions`, nos controladores da API.

Observável na Fig. 7, que este advice simplesmente faz o attach da mensagem da exceção lançada ao response body com o response status `Unauthorized`.

```
@ControllerAdvice
public class UnauthorizedAccessResponse {
    @ResponseBody
    @ExceptionHandler(UnauthorizedAccessException.class)
    @ResponseStatus(HttpStatus.UNAUTHORIZED)
    String accessPrivilegesNotFoundHandler(UnauthorizedAccessException ex) { return ex.getMessage(); }
}
```

7. Controller Advice UnauthorizedResponse

3. Mensagens

Na componente de mensagens este paradigma permitiu criar a possibilidade de enviar uma notificação push para o destinatário de uma mensagem. Para esse fim, foi elaborado o aspecto “`SendPushNotification`” com o pointcut e o advice denominados de `saveMessagePointcut` e `sendPushNotificationToUser`.

O pointcut aponta para a execução do método de guardar uma mensagem na classe `MessageController`. Observável na Fig. 8.

```
@Pointcut("execution(* com.example.springangularadsapp.components.messages.controller.MessageController.save(..)")
public void saveMessagePointcut() {
}
```

8. Implementação pointcut do aspecto `SendPushNotification`

O advice utiliza o pointcut, juntamente com a anotação **@AfterReturning** que implica que o advice será executado quando o método de guarda mensagem retornar o resultado. Após retornar o resultado o advice é responsável por utilizar o token firebase do destinatário e a resposta do mesmo método e chamar o método de `sendMessageNotification` do serviço `FirebaseMessagingService`.

```
@AfterReturning(value = "saveMessagePointcut()", returning = "responseEntity")
public void sendPushNotificationToUser(ResponseEntity<?> responseEntity) throws IOException, FirebaseMessagingException {
    EntityModel<Message> entityModel = (EntityModel<Message>) responseEntity.getBody();
    String token = entityModel.getContent().getTo().getFirebaseToken();
    firebaseMessagingService.sendMessageNotification(entityModel, token);
}
```

9. Implementação advice do aspecto `SendPushNotification`

5.1.2. Design Patterns

5.1.2.1. General Design Patterns

Os design patterns utilizados para o contexto da RESTful API implementada foram o singleton pattern e o factory pattern.

1. Singleton

O singleton pattern foi utilizado de forma generalizada para criar instâncias únicas de aspetos, factories, mappers, assemblers, repositórios, controladores e serviços.

A própria framework Spring facilita o seu uso deste padrão através dos IoC Containers, que podem ser instanciados com anotação **@Bean** ou derivados desta, permitindo criar uma instância única de um objeto no scope definido no Container.

2. Strategy

O strategy pattern é um padrão de software comportamental, que identifica a comunicação comuns entre objetos, aumentando a flexibilidade de comunicação. Perante este facto o padrão não foi utilizado dada que a dimensão do tema e contexto não permitiu o uso de interface para a comunicação entre objetos.

3. Observer

O observer pattern é igualmente categorizado como um é um padrão de software comportamental, cujo um objeto (observable) possui uma lista de objetos dependentes (observers). Por esta razão, o padrão observer não foi utilizado por não existiu a necessidade de implementar um funcionalidade que encaixe neste contexto de uso.

4. Decorator

O decorator pattern é um padrão que para ser utilizado exige a existência de interfaces para permitir a flexibilidade de estender funcionalidades a objetos, sem o uso de herança, com este efeito, o padrão não acabou por ser utilizado devido à falta de interfaces Java a oferecer funcionalidades ao objetos criados para o contexto da aplicação.

5. Factory

O factory pattern é um padrão de software de criação que utiliza um interface para criar objetos de uma super classe. Este padrão foi utilizado dada a necessidade de

instanciar subclasses da classe abstrata **Ad**, no mapper denominado de **AdMapper**, para este propósito foi criada a classe **AdFactory**.

A classe **AdFactory** é observável na Fig. 10 e nesta é possível observar que foi utilizado também o padrão singleton, com o uso da anotação **@Component**. Esta factory é responsável por instanciar as subclasses da super classe abstrata **Ad**.

```
@Component
public class AdFactory {
    public Ad getAdInstance(Class<? extends AdDto> entity) {
        if (entity == BasicAdDto.class) {
            return new BasicAd();
        }

        if (entity == CarAdDto.class) {
            return new CarAd();
        }
        return null;
    }
}
```

10. Implementação da **AdFactory**

5.1.2.2. IOC Container Design Patterns

1. Inversion of Control Principle (IoC)

O padrão inversion of control é utilizado para alcançar um controlador universal de anúncios, denominado de **AdController**. Este **AdController** surgiu devido à necessidade de existir um endpoint que retorne todos as sub classes da classe abstrata **Ad**. Para alcançar o inversion of control no **AdController** foi utilizada a classe abstrata **Ad**, observável na Fig.11.

```
@GetMapping("/{all}")
public CollectionModel<EntityModel<Ad>> all() {
    List<EntityModel<Ad>> ads = super.getRepository().findAll().stream().map(super.getAssembler().toModel).collect(Collectors.toList());
    return CollectionModel.of(ads, linkTo(methodOn(AdController.class).all()).withSelfRel());
}
```

11. Exemplo Inversion of Control - Método **all** do **AdController**

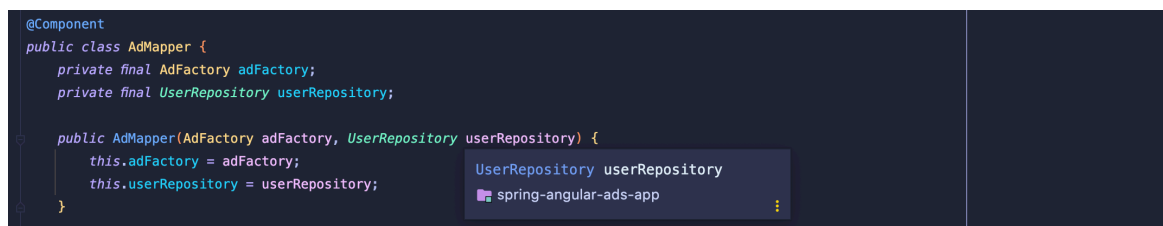
2. Dependency Inversion Principle (DIP)

O princípio de dependency inversion é alcançado através do factory pattern, que inverte a criação de instâncias da super class **Ad** através do **AdFactory**, como foi mencionado anteriormente.

3. Dependency Injection Pattern (DI)

A dependency injection é um aspeto fundamental da framework Spring, pelo qual um container Spring é injecta objetos em outros objetos ou dependências. Dado tal facto, o padrão DI foi aplicado na injeção de containers singleton como repositórios, serviços, factories, assemblers, mappers, entre outros.

Na Fig.12 é possível observar um exemplo do padrão DI implementado na classe AdMapper, que necessita de uma instância AdFatory e uma instância UserRepository.

The image shows a code editor with a dark theme. It displays the Java code for a class named AdMapper. At the top, there is an annotation @Component. The class is defined as public class AdMapper {. Inside the class, there are two private final fields: private final AdFactory adFactory; and private final UserRepository userRepository;. Below these, there is a constructor: public AdMapper(AdFactory adFactory, UserRepository userRepository) {. The constructor body contains two lines: this.adFactory = adFactory; and this.userRepository = userRepository;. To the right of the constructor body, there is a tooltip showing the type UserRepository and the variable name userRepository, with a reference to 'spring-angular-ads-app'.

12. Exemplo Dependency Injection na classe AdMapper

4. Inversion of Control Containers (IoC Containers)

Uma das mais importantes e interessante mecanismos da Spring Framework são os IoC Containers, que são responsáveis pela componente de Dependency Injection.

Um Spring IoC Container pode ser registado com a anotação @Componente e é responsável por instanciar, configurar e montar os beans da Spring Framework.

Os IoC Containers foram utilizados para criar diversos componentes utilizando a anotação @Component ou anotações derivadas desta para casos mais específicos como: repositórios (@Repository), controladores (@RestController), serviços(@Service) e configurações (@Config). De seguida serão enumerado o uso destes nas componentes implementadas.

A. Repositórios

- RoleRepository - Repositório de Roles de Utilizadores
- UserRepository - Repositório de Utilizadores
- AdRepository - Repositório de Anúncios
- MessageRepository - Repositório de Mensagens

B. Controladores

- AuthController - Controlador de Autenticação
- AdController - Controlador de Anúncios Ad
- BasicAdController - Controlador de Anúncios BasicAd
- CarAdController - Controlador de Anúncios CarAd

C. Serviços

- CustomUserDetailsService - Serviço UserDetails
- CustomFirebaseMessagingService - Serviço de Mensagens Firebase

D. Configurações

- LoadMongoDatabase - Configuração da população da Mongo Database
- WebMvcConfig - Configuração da disponibilização da PWA.
- WebSecurityConfig - Configuração de segurança.

5.1.3. SOLID Principles

5.1.3.1. Single-responsibility

As classes criadas para a lógica de negócio exercida não possuem uma grande complexidade, pois é focada em fornecer e obter informação, logo as classes são bem definidas com um propósito único.

5.1.3.2. Open-closed

O princípio open-closed é aplicado simplesmente pelo uso de classes abstratas, que implica que não é necessário alterar o código no caso de realizar mudanças.

5.1.3.3. Liskov substitution

Este princípio foi aplicado no AdController ao utilizar a classe abstrata Ad, para manipular todas as suas subclasses, é possível observar este facto na Fig. 11, que exemplifica um endpoint para obter a lista de todos os anúncios que possuam a super class Ad.

5.1.3.4. Interface Segregation Principle

Este princípio foi implementado com a criação da interface IHateoasController e da classe abstrata HateoasController, com recurso ao uso de genéricos Java. Observáveis nas Fig. 13 e 14, respetivamente.

```
public interface IHateoasController<T,S> {  
  
    public CollectionModel<EntityModel<T>> all();  
  
    public EntityModel<? extends T> one(@PathVariable String id);  
  
    public ResponseEntity<?> save(@RequestBody S newEntity);  
  
    public ResponseEntity<?> update(@RequestBody S newEntity, @PathVariable String id);  
  
    public ResponseEntity<?> delete(@PathVariable String id);  
  
}
```

13. Interface IHateoasController

```
public abstract class HateoasController<T, S> implements IHateoasController<T, S> {  
  
}
```

14. Classe Abstrata HateoasController

A criação destes componentes permitiu utilizar estes como base para implementar controladores do tipo Hateoas, como por exemplo o AdHateoasController e MessageController, visíveis nas Fig. 15 e 16.

```
public abstract class AdHateoasController<T extends Ad, S extends AdDto> extends HateoasController<T, S> {  
    private final AdRepository<T> repository;  
    private final AdModelAssembler<T> assembler;  
    private final UserRepository userRepository;  
  
    public AdHateoasController(AdRepository<T> repository, AdModelAssembler<T> assembler, UserRepository userRepository) {  
        this.repository = repository;  
        this.assembler = assembler;  
        this.userRepository = userRepository;  
    }  
}
```

15. Classe Abstrata AdHateoasController

```
@RestController  
@RequestMapping("/api/messages")  
public class MessageController extends HateoasController<Message, MessageDto> {  
    private final MessageRepository messageRepository;  
    private final UserRepository userRepository;  
    private final MessageModelAssembler assembler;  
    private final MessageMapper mapper;  
}
```

16. Implementação MessageController

O próprio `AdHateoasController` também utiliza este princípio para permitir a implementação de controladores, podendo assim implementar os controladores: `AdController`, `BasicAdController` e `CarAdController`, visíveis nas Fig. 17, 18 e 19.

```
@RestController
@RequestMapping("/api/ads")
public class AdController extends AdHateoasController<Ad, AdDto> {
    public AdController(AdRepository<Ad> repository, AdModelAssembler<Ad> assembler, UserRepository userRepository) {
        super(repository, assembler, userRepository);
    }
}
```

17. Implementação `AdController`

```
@RestController
@RequestMapping("/api/ads/basic")
public class BasicAdController extends AdHateoasController<BasicAd, BasicAdDto> {
    private final AdMapper mapper;

    public BasicAdController(AdRepository<BasicAd> repository, UserRepository userRepository, AdModelAssembler<BasicAd> assembler, AdMapper mapper) {
        super(repository, assembler, userRepository);
        this.mapper = mapper;
    }
}
```

18. Implementação `BasicAdController`

```
@RestController
@RequestMapping("/api/ads/cars")
public class CarAdController extends AdHateoasController<CarAd, CarAdDto> {
    private final AdMapper mapper;

    public CarAdController(AdRepository<CarAd> repository, UserRepository userRepository, AdModelAssembler<CarAd> assembler, AdMapper mapper) {
        super(repository, assembler, userRepository);
        this.mapper = mapper;
    }
}
```

19. Implementação `CarAdController`

5.1.3.5. Dependency Inversion Principle

O princípio de dependência inversa é incorporado no modelo `Message`, no qual este usa uma layer de abstração, neste caso a classe Abstrata `Ad`, para referenciar as classes especializadas desta classe. Esta layer de abstração `Ad` é demonstrada no modelo `Message` na Fig. 20.

```
@ToString
@EqualsAndHashCode
@Document(collection = "messages")
public class Message {
    @Id
    private String id;

    @CreatedDate
    private Date createdAt;

    @LastModifiedDate
    private Date lastModifiedDate;

    @DBRef
    private User from;

    @DBRef
    private User to;

    @DBRef
    private Ad ad;

    private String message;

    public Message() {
    }
}
```

20. Modelo `Message`

5.1.4. HATEOAS Principle

O princípio HATEOAS é uma restrição da arquitetura RESTful que exige a existência de hypermedia. Este princípio foi exercido com o uso de model assemblers, estes são o AdModelAssembler e o MessageModelAssembler. Estes model assemblers criam EntityModels, para adicionar a hypermedia as responses de pedidos, visível na Fig. 21.

```
1 {
2   "_embedded": {
3     "basicAdList": [
4       {
5         "id": "6276423e714f2e70b67e017f",
6         "createdDate": "2022-05-07T09:56:14.294+00:00",
7         "lastModifiedDate": "2022-05-08T11:29:15.923+00:00",
8         "owner": { ~ 6 ~ },
9         "imageList": null,
10        "title": "Teste 2",
11        "description": "Teste Images",
12        "_links": {
13          "self": {
14            "href": "http://localhost:8080/api/ads/basic/6276423e714f2e70b67e017f"
15          },
16          "allBasicAds": {
17            "href": "http://localhost:8080/api/ads/basic/"
18          }
19        }
20      }
21    ],
22    "_links": {
23      "self": {
24        "href": "http://localhost:8080/api/ads/all"
25      }
26    }
27  }
28 }
```

21. Exemplo de response com links HATEOAS

5.2. Progressive Web App

Nesta seção serão apresentados os artefactos desenvolvidos para que a web app cumpra os critérios e possa ser considerada uma progressive web app.

5.2.1. Web Manifest

O web manifest é um ficheiro JSON que contém as informações sobre a web app num formato específico e estruturado que permite os search engines realizarem facilmente a indexação da web app (discoverable). O manifesto é responsável por fornecer informações sobre a web app como o nome, icons, temas, linguagem, start_url, entre outras.

O start_url é importante porque contém o url que deve ser usado no arranque da aplicação, o que é útil se a aplicação for iniciada através do home screen (installability). Além demais, a utilização de um único url para navegar pela aplicação facilita a sua partilha e descoberta, sem recurso de uma loja de aplicações (linkability/discoverable).

```

{
  "id": "/",
  "start_url": "/",
  "name": "PWAds",
  "description": "An PWA Ads App developed by Daniel Castro.",
  "scope": "/",
  "gcm_sender_id": "370072803732",
  "background_color": "#ee8d32",
  "orientation": "landscape",
  "display": "standalone",
  "theme_color": "#ee8d32",
  "lang": "en-US",
  "short_name": "pwads-app",
  "icons": [
    {
      "src": "./icons/windows11/SmallTile.scale-100.png",
      "sizes": "71x71"
    },
    {
      "src": "./icons/windows11/SmallTile.scale-125.png",
      "sizes": "89x89"
    },
    {
      "src": "./icons/windows11/SmallTile.scale-150.png",
      "sizes": "107x107"
    },
    {

```

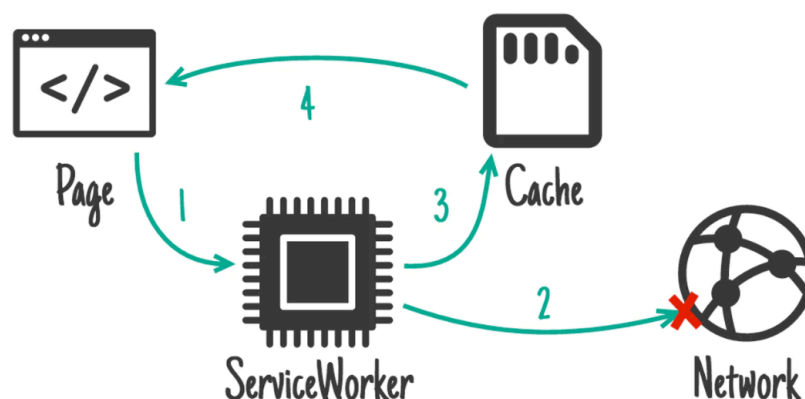
22. Manifest da aplicação PWAds

5.2.2. Service Worker

O service worker é um worker especial baseado em eventos, que serve como orquestrado da web app.

Este worker tem responsabilidades de tornar a web app resiliente a interrupções de conexão à internet (network independence), sendo esta característica um requisito para a instabilidade da aplicação (installability). Além destes, o service worker é responsável por realizar o fornecimento de recursos que a aplicação necessita, utilizando o protocolo HTTPS (secure).

O service worker da web app de anúncios para cumprir o princípio de network independence foi adotado a estratégia de “network first, falling back to cache”.



23. Representação da estratégia “Network first, falling back to cache”.

A estratégia de “network first, falling back to cache” significa que no momento que o service worker recebe um request, primeiramente tentar obter o recurso pretendido através da network, após a sua conclusão é adicionado à cache e depois devolvido. Caso não foi possível obter o recurso o service worker tentará obter o recurso a partir da cache e devolve o recurso. Como ultimo recurso se não foi possível obter o recurso através da network ou da cache, o service worker irá devolver uma página html de fallback.

```
self.addEventListener('type: fetch', listener: (event: Event) => {
  event.respondWith(caches.open(CACHE_NAME).then((cache: Cache) => {
    // Go to the network first
    return fetch(event.request.url).then((fetchedResponse: Response) => {
      if (event.request.method === 'GET') {
        cache.put(event.request, fetchedResponse.clone());
      }
      // Return the network response
      return fetchedResponse;
    }).catch(() => {
      if (event.request.method !== 'GET') {
        return Promise.reject( reason: 'no-match' );
      }
      // If the network is unavailable, get from cache
      return cache.match(event.request.url).then((cachedResponse: Response | undefined) => {
        // Return a cached response if we have one
        if (cachedResponse) {
          return cachedResponse;
        }
      }).catch((error) => {
        console.log('Fetch failed; returning offline page instead.', error);
        return cache.match( request: '/html/fallout.html' );
      });
    });
  }));
});
```

24. Implementação da estratégia no service worker.

As razões para a adoção desta estratégia deve-se à necessidade imposta pelo contexto da lógica de negócios, visto que trata-se de uma temática de anúncios o que sujeita a aplicação tentar obter os dados mais recentes possíveis.

5.2.3. Notifications API e Firebase Messaging

Uma progressive web app necessita de interagir com os utilizadores sem estes estarem a utilizar o browser (re-engageability).

No propósito de incorporar este comportamento foi implementada a funcionalidade de no momento em que uma mensagem seja enviada, o destinatário seja notificado da existência da mensagem. Este envio de notificações é composta por uma componente na progressive web app e na RESTful API.

Antes demais, foi necessário criar um projeto Firebase na Firebase Console. Para tal foram seguidos os tutoriais de [“Adicionar o Firebase ao projeto JavaScript”](#) e o [“Adicionar o SDK Admin do Firebase ao servidor”](#), para configurar e adicionar as credenciais do projeto Firebase na aplicação pwa e na RESTful API.

1. RESTFul API

Após a realização do tutorial foi obtida a classe CustomFirebaseMessaging, com o registo da aplicação com as credenciais configuradas na Firebase Console. Com a aplicação registada foi criado um serviço denominado CustomFirebaseMessagingService, com o método sendMessageNotification. Este método é responsável por criar uma notificação com uma instância de Message e o token de destinatário.

```
@Component
public class CustomFirebaseMessaging {
    private FirebaseApp app;

    FirebaseMessaging firebaseMessaging() throws IOException {
        GoogleCredentials googleCredentials = GoogleCredentials
            .fromStream(new ClassPathResource("/firebase/pwads-app-firebase-adminsdk-s1o5f-5d2c784e03.json").getInputStream());
        FirebaseOptions firebaseOptions = FirebaseOptions
            .builder()
            .setCredentials(googleCredentials)
            .build();

        try {
            app = FirebaseApp.initializeApp(firebaseOptions, name: "my-app");
        } catch (Exception ignored) {}

        return FirebaseMessaging.getInstance(app);
    }
}
```

25. Configuração Firebase Messaging App in RESTful API

```
private final CustomFirebaseMessaging firebaseMessaging;

public CustomFirebaseMessagingService(CustomFirebaseMessaging firebaseMessaging) {
    this.firebaseMessaging = firebaseMessaging;
}

public String sendMessageNotification(EntityModel<Message> model, String token) throws FirebaseMessagingException, IOException {
    Message msg = model.getContent();
    HashMap<String, String> map = new HashMap<>();
    map.put("embedded", model.getContent().toString());
    map.put("_links", model.getLinks().toString());
    Notification notification = Notification
        .builder()
        .setTitle(msg.getFrom().getUsername())
        .setBody(msg.getMessage())
        .build();

    com.google.firebase.messaging.Message message = com.google.firebase.messaging.Message
        .builder()
        .setToken(token)
        .setNotification(notification)
        .putAllData(map)
        .build();

    return firebaseMessaging.firebaseMessaging().send(message);
}
```

26. Método que cria notificações e envia para utilizador

O envio de notificação é realizado quando uma mensagem é guardada na RESTful API com sucesso. Este envio é alcançado com o uso de AOP, sendo assim foi criado o aspecto SendPushNotification.

O aspecto possui um pointcut que aponta para o método que guarda mensagens do MessageController e executa o advice sendPushNotificationToUser que envia as notificações push, após o momento em que o método retorne o valor.

```
@Component
@Aspect
public class SendPushNotification {
    private final CustomFirebaseMessagingService firebaseMessagingService;
    private final Logger logger = LoggerFactory.getLogger(getClass());

    public SendPushNotification(CustomFirebaseMessagingService firebaseMessagingService) {
        this.firebaseMessagingService = firebaseMessagingService;
    }

    @Pointcut("execution(* com.example.springangularadsapp.components.messages.controller.MessageController.save(..))")
    public void saveMessagePointcut() {
    }

    @AfterReturning(value = "saveMessagePointcut()", returning = "responseEntity")
    public void sendPushNotificationToUser(ResponseEntity<?> responseEntity) throws IOException, FirebaseMessagingException {
        EntityModel<Message> entityModel = (EntityModel<Message>) responseEntity.getBody();
        String token = entityModel.getContent().getTo().getFirebaseToken();
        firebaseMessagingService.sendMessageNotification(entityModel, token);
    }
}
```

27. Aspecto SendPushNotification

2. Progressive Web App

Na progressive web app a implementação das push notification com recurso à aplicação Firebase é realizada com o script firebase-notifications.js e o service worker.

O service worker é responsável por instanciar de forma semelhante à aplicação firebase que na RESTful API, com as credenciais de aplicação para Javascript e inicializar o modulo messaging do firebase.

```
//firebase-notifications-sw.js

importScripts('https://www.gstatic.com/firebasejs/6.6.2/firebase-app.js');
importScripts('https://www.gstatic.com/firebasejs/6.6.2/firebase-messaging.js');

const vapidKey =
    'BCX_it2GStqQGsv1IJRzylZCjEvo-adRhY47KUKv0BGhd0nNH2xyRN90oojy0Da7s66vZ-FwbEckdj90';

const firebaseConfig = {
    apiKey: 'AIzaSyBH0o09K98VCIAfX2ng8nKj0-_2pbAIPUK',
    authDomain: 'pwads-app.firebaseio.com',
    projectId: 'pwads-app',
    storageBucket: 'pwads-app.appspot.com',
    messagingSenderId: '63582499105',
    appId: '1:63582499105:web:810f385ca66e61fc3a4e60',
    measurementId: 'G-MR0N6J3ZSX',
};

// Initialize Firebase
firebase.initializeApp(firebaseConfig);

// Initialize Firebase Cloud Messaging and get a reference to the service
const messaging = firebase.messaging();
```

28. Configuração firebase messaging no service worker

Isto é importante porque é necessário criar os tokens firebase para o cliente, em que o firebase necessita para estabelecer a comunicação entre os clientes e o servidor. O gerador de tokens messaging firebase é o método `getToken` do service worker. O service worker também possui a configuração dos listeners “message” e “push”.

```
self.addEventListener( type: 'push', listener: (event :Event ) => {
  let payload = event.data ? event.data.text() : 'no payload';
  payload = JSON.parse(payload);
  let username = payload.notification.title;
  let message = payload.notification.body;

  // Customize notification here
  const notificationTitle = 'PWAds - New message received';
  const notificationOptions = {
    body: `The user ${username} sent you the following message: ${message}`,
    icon: '/icons/ios/100.png',
  };

  self.registration.showNotification(notificationTitle, notificationOptions);
});

// service-worker.js
self.addEventListener( type: 'message', listener: (event :MessageEvent<any> ) => {
  console.log(`The client sent me a message: ${event.data}`);
  getToken(event);
});
```

29. Implementação listeners push e message

O listener “message” é configurado para que o service worker funcione como um web worker, que quando chamado este chame a função `getToken` para gerar os tokens e devolver. Esta implementação é devido ao modulo messaging do firebase que necessite que a comunicação seja HTTPS.

O listener “push” é responsável por receber as notificações do firebase e instanciar a notificação da maneira apropriada.

Por fim, o script firebase-notifications tem um função que possui a responsabilidade pedir ao utilizador se tem permissão para ativar as notificações. Após a aprovação esta verifica se o service worker está ativo e nesse caso realiza um `postMessage` para obter o token de messaging Firebase.

```

function requestPermission() {
  // [START messaging_request_permission]
  Notification.requestPermission().then((permission : NotificationPermission ) => {
    if (permission === 'granted') {
      console.log('Notification permission granted.');
```

```

      if ('serviceWorker' in navigator) {
        navigator.serviceWorker.addEventListener( type: 'message', listener: (event : MessageEvent) => {
          if (localStorage.getItem( key: 'firebase-token') === null) {
            localStorage.setItem('firebase-token', event.data);
          }
        });
        navigator.serviceWorker.ready
          .then(function (registration : ServiceWorkerRegistration ) {
            if (localStorage.getItem( key: 'firebase-token') === null) {
              registration.active.postMessage( message: 'token');
```

```

              console.log(
                'Registration successful, scope is:',
                registration.scope
              );
            }
          })
          .catch(function (err) {
            console.log('Service worker registration failed, error:', err);
          });
      }
    } else {
      console.log('Unable to get permission to notify.');
```

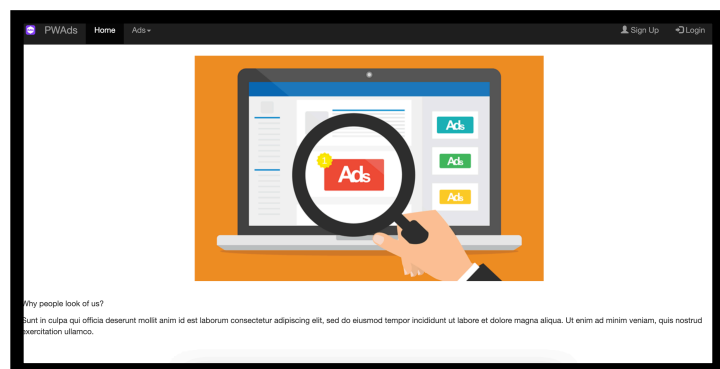
30. Implementação da estratégia de cache

5.2.4. CSS e Bootstrap

Na ambição de tornar a web app responsiva foram criados scripts de style css, juntamente com os scripts da biblioteca Bootstrap. Pode ser observável a responsividade da web app nas Fig. 31 e 32.



31. Layout Móvel



32. Layout Desktop

5.2.5. Progressive Enhancement Support

Infelizmente este princípio não foi incorporado totalmente, porque exigiria programar a web app para fornecer uma experiência aceitável em diversos browsers e de versões diferentes. Dado este facto, o princípio não foi implementado completamente e apenas foi testada a sua experiência no browser Chrome e Firefox.

6. Lighthouse

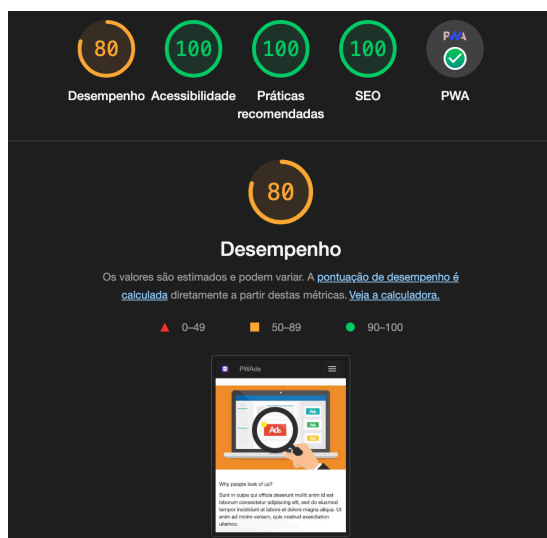
Nesta secção será apresentado dois relatórios obtidos a partir da ferramenta do browser Chrome denominada de Lighthouse. O Lighthouse é uma ferramenta automatizada open-source que verifica a qualidade das web apps, principalmente em recurso de Progressive Web Apps.

O Lighthouse executa uma série de testes automatizados na página dependendo das configurações de teste. Em ambos os relatórios produzidos foram realizados testes na seguintes categorias: desempenho, progressive web app, práticas recomendadas, acessibilidade, SEO e PWA. Os relatórios diferem no dispositivo em que os testes foram realizados um foi no dispositivo móvel e outro no desktop.

A execução dos testes Lighthouse foram realizados numa janela de navegação anónima, por sugestão da própria ferramenta.



33. Resultados Lighthouse Desktop



34. Resultados Lighthouse Móvel

Observando, os resultados é possível afirmar que a web app alcançou a cotação máxima nas categorias de : acessibilidade, práticas recomendadas, SEO e PWA. O que significa que a web app de anúncios possui todas as características de uma progressive web app.