Regras para Desenvolvimento do Backend

Participar da Aliança pela Mobilidade Sustentável

Ingenico do Brasil

Bahia, BA
Versão 1.0
Julho de 2024

Identificação

Título: Definição do *Backlog* e distribuição das tarefas para primeira *sprint*
Projeto: Participar da Aliança pela Mobilidade Sustentável
Data: Julho 2024
Local: Bahia, BA
Versão: 1.0

Revisões

Data	**Alterações / Comentário**	**Revisor**
29.07.2024	Criação do documento.	João Vitor N. Ramos

Equipe de Desenvolvimento

Tutor

Marcelo Silva

Tutor

Rogério de Jesus

Líder Técnico

João Vitor Nascimento Ramos

Desenvolvedores

Albert Silva de Jesus

Danilo da Conceição Santos

Everlan Santos de Rosário

Leonardo Ribeiro Barbosa Santos

Sumário

1 Regras para Escrita e Organização de Código

1.1 Informações sensíveis

Informações sensíveis, como nomes de usuário, senhas e chaves secretas, devem ser armazenadas em variáveis de ambiente e nunca expostas diretamente no código-fonte para garantir a segurança e facilitar o gerenciamento de configuração. Isso evita a exposição acidental dessas informações, além de permitir que configurações sejam facilmente alteradas sem modificar o código, especialmente em diferentes ambientes de desenvolvimento, teste e produção.

Por exemplo, o código abaixo demonstra como referenciar variáveis de ambiente em vez de expor diretamente informações sensíveis (veja a Listagem 1.1):

Listing 1.1 – Configurações do Banco de Dados e Segurança

```
spring.datasource.url=jdbc:postgresql://localhost:5432/
    shoppingStore
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.username=${POSTGRES_DATASOURCE_USER}
spring.datasource.password=${POSTGRES_DATASOURCE_PASSWORD}

api.security.token.secret=${JWT_SECRET}
```

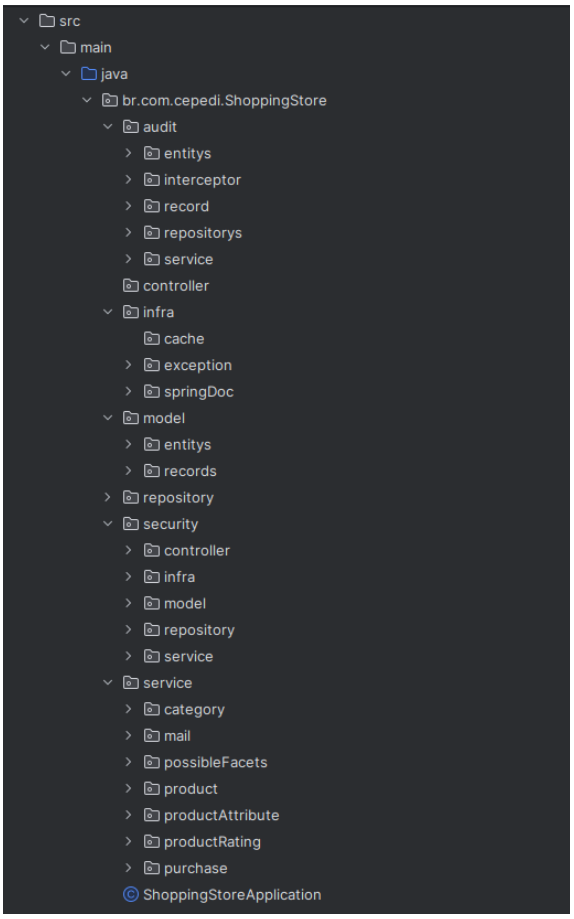
1.2 Estrutura de *Packages*

Foi estabelecida uma estrutura básica de *packages* com o intuito de organizar e manter cada classe em seu devido lugar, facilitando assim a localização.

Basicamente, toda entidade tem um *DTO* de *input* e *output* chamados de '*register*' e '*details*', que definem o que é necessário o usuário passar para realizar um registro e o que deve ser retornado ao *frontend*. Cada entidade possui um *repository*, um *service* e, em alguns casos, um *controller*. A Figura 1 ilustra essa organização.

O **repository** no Spring Boot é responsável por interagir diretamente com o banco de dados, executando operações de CRUD (Create, Read, Update, Delete).

Figura 1 – Exemplo da estrutura de *packages*



O **service** contém a lógica de negócio da aplicação, chamando os métodos do *repository* e realizando as operações necessárias antes de retornar os dados ao *controller*. O **controller** é a camada que lida com as requisições HTTP, recebendo dados do *frontend*, chamando os métodos do *service* e retornando as respostas adequadas.

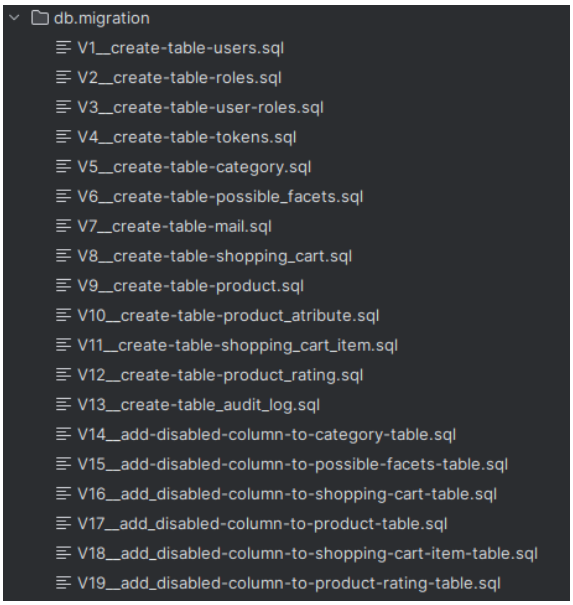
Além disso, o *package* de segurança (*security*) é isolado devido à possibilidade de construí-lo com baixo acoplamento, permitindo que ele seja reutilizado em outros projetos com mínimas modificações. Isso facilita a manutenção, além de promover a possiblidade de reutilização de código e a padronização das práticas de segurança.

1.3 *Migrations*

Cada modificação no banco de dados, como a criação de tabelas ou qualquer comando DDL, requer uma nova *migration*. Essas *migrations* são nomeadas seguindo uma sequência numérica, começando com ’S(O número correspon-

dente)__', garantindo a ordem e unicidade das instruções. Um exemplo desse padrão é ilustrado na Figura 2.

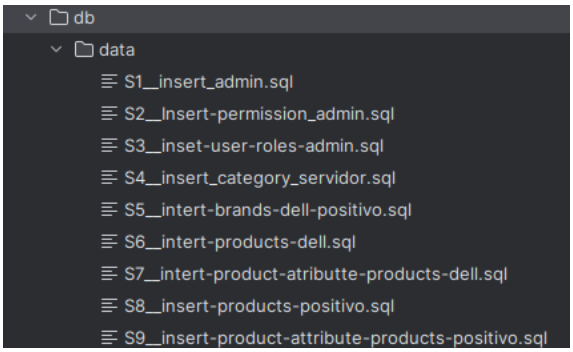
Figura 2 – Exemplo de estrutura do *package* db.migration



1.4 *Seeders*

Os *seeders* desempenham um papel crucial na inicialização e no preenchimento inicial do banco de dados com dados pré-definidos. Localizados no pacote de dados (*package data*) em `db/resources`, esses arquivos seguem uma convenção de nomenclatura que inclui uma sequência numérica, começando com 'S(O número correspondente)__'. Um exemplo visual da estrutura do diretório *package* db.data é ilustrado na Figura 3.

Figura 3 – Exemplo de estrutura do *package* db.data



Para executar os *scripts* de *seeders* durante o processo de inicialização do banco de dados, é necessário adicionar as seguintes instruções no arquivo `application.properties`:

```
spring.sql.init.mode=always
spring.sql.init.platform=postgres
spring.sql.init.data-locations=classpath:db/data/S__.sql
```

A configuração `spring.sql.init.mode=always` garante que os scripts de *seeders* sejam executados sempre que a aplicação for iniciada, garantindo a consistência dos dados. O `spring.sql.init.platform=postgres` especifica a plataforma de banco de dados a ser usada durante a inicialização. Já `spring.sql.init.data-locations` indica o local onde os scripts de *seeders* estão localizados no classpath.

É importante notar que, após a primeira população do banco de dados, é recomendável alterar o modo de inicialização para `never` (`spring.sql.init.mode=never`) para evitar a sobregravação dos dados existentes durante futuras inicializações da aplicação. Isso garante que os dados já existentes no banco permaneçam intactos.

1.5 Auditoria

Na auditoria será utilizada a estratégia por *IpAddressInterceptor* combinada com a programação orientada a aspectos (AOP), o que garante menor escrita de código e maior modularidade. A utilização de AOP permite que preocupações transversais, como auditoria, sejam implementadas de forma isolada, sem a necessidade de poluir o código principal das aplicações.

Dessa forma, a auditoria se torna um componente reutilizável e facilmente gerenciável, podendo ser ajustada ou expandida conforme necessário sem grandes impactos no restante da aplicação.

1.6 Controle de Exceções

O controle de exceções é gerenciado pelo arquivo *ErrorHandler*, localizado no *package* infra/*exception*, e é marcado com a anotação `@RestControllerAdvice`. Este componente desempenha um papel crucial na aplicação, interceptando e tratando exceções que podem ocorrer durante sua execução. Ao utilizar a anotação `@RestControllerAdvice`, o *ErrorHandler* é globalmente aplicado em toda a aplicação Spring Boot, garantindo um tratamento consistente de exceções em todos os endpoints, evitando a repetição de código para o tratamento de erros em cada controlador individualmente.

Por meio do *ErrorHandler*, é possível definir diferentes comportamentos para diferentes tipos de exceções, como retornar mensagens de erro personalizadas e códigos de status HTTP apropriados. Abaixo segue um exemplo (veja a Listagem 1.2).

Listing 1.2 – Exemplo de ErrorHandler

```java
@RestControllerAdvice
public class ErrorsHandler {
    private static final Logger logger = LoggerFactory.getLogger(
        ErrorsHandler.class);
    @ExceptionHandler(EntityNotFoundException.class)
    public ResponseEntity<Object> Error404() {
        logger.error("EntityNotFoundException occurred.");
        return ResponseEntity.notFound().build();
    }
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Object> Error400(
        MethodArgumentNotValidException exception) {
        logger.error("MethodArgumentNotValidException occurred.",
            exception);
        List<FieldError> errors = exception.getFieldErrors();
        return ResponseEntity.badRequest().body(errors.stream().map
            (DataExceptionValidate::new).toList());
    }
    @ExceptionHandler(BadCredentialsException.class)
    public ResponseEntity<Object> handleBadCredentialsError() {
        logger.error("BadCredentialsException occurred.");
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body(
            "Invalid credentials");
    }
    private record DataExceptionValidate(String value, String
        message) {
        public DataExceptionValidate(FieldError error) {
            this(error.getField(), error.getDefaultMessage());
        }
    }
}
```

1.7 Cacheable

O uso da anotação `@Cacheable` deve ser utilizado em consultas que retornam grandes volumes de dados é essencial para otimizar o desempenho e reduzir a carga no sistema. Ao armazenar os resultados dessas consultas em cache, evita-se a necessidade de consultas repetitivas ao banco de dados, reduzindo o tempo de resposta e minimizando o consumo de recursos do servidor. Isso não apenas melhora a experiência do usuário, proporcionando tempos de resposta mais rápidos, mas também aumenta a escalabilidade da aplicação, permitindo lidar com um maior número de solicitações simultâneas de forma mais eficiente.

1.8 Entidades

1.8.1 Construtores de Entidades

Para garantir consistência e facilitar o desenvolvimento, todas as entidades devem fazer uso de recursos como Lombok, a anotação `@Entity`, e o `@Table` para especificar o nome da tabela no banco de dados. Além disso, devem ser configurados para receber os objetos DTO correspondentes em seus construtores, como ilustrado abaixo (veja a Listagem 1.3).

Listing 1.3 – Exemplo de Entidade Patient com Lombok e Anotações JPA

```
@Entity
@Table(name = "patients")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = "id")
@ToString
public class Patient {

    ...atributos

    public Patient(DataRegisterPatient data) {
        this.name = data.name();
        this.email = data.email();
        this.phoneNumber = data.phoneNumber();
        this.cpf = data.cpf();
        this.address = new Address(data.dataAddress());
        this.activated = true;
    }
```

1.8.2 Construtores de Entidades que possuem composição

Para entidades que têm composição, ou seja, precisam de dados adicionais que não são fornecidos pelo DTO, os valores relevantes devem ser passados juntamente com o DTO, como exemplificado abaixo (veja a Listagem 1.4):

Listing 1.4 – Exemplo de Construtor de Entidade Product com Dados Adicionais

```
public Product(DataRegisterProduct data, Category category){
    this.name = data.name();
    this.description = data.description();
    this.price = data.price();
    this.sku = data.sku();
    this.imageUrl = data.imageUrl();
    this.quantity = data.quantity();
    this.manufacturer = data.manufacturer();
    this.featured = data.featured();
    this.category = category;
    this.disabled = false;
}
```

Segue um exemplo de um DTO de registro (veja a Listagem 1.5):

Listing 1.5 – Exemplo de DTO para Registro de Categoria

```
public record DataRegisterCategory(
        @NotBlank
        String name,
        @NotBlank
        String description
) {
}
```

E também um exemplo de um DTO de detalhes (veja a Listagem 1.6):

Listing 1.6 – Exemplo de DTO para Detalhes de Categoria

```
public record DataCategoryDetails (
        Long id,
        String name,
        String description
) {
    public  DataCategoryDetails(Category category){         this(
       category.getId(),category.getName(),category.getDescription
       ());
    }

}
```

1.9 Mensagens de erro nos DTO's

Padronizar as mensagens de erro nos DTOs é essencial para garantir consistência e facilitar a manutenção do código. Armazenar essas mensagens em

um arquivo como `ValidationMessages.properties` ajuda na gestão e tradução delas conforme necessário. Veja o exemplo no código abaixo:

```
validation.required.name=O nome é obrigatório.
validation.required.email=E-mail é obrigatório.
validation.length.min.name=O nome deve ter pelo menos {0} caracteres.
validation.pattern.email=E-mail inválido. Insira um e-mail válido.
```

Essa prática promove uma experiência consistente para os usuários e simplifica a manutenção do sistema.

1.10 Repositorys

Para cada entidade, é necessário um repositório correspondente. Pode ser utilizado a sintaxe JPQL ou seguir a convenção de nomenclatura do JPA. Deve ser evitado a convenção JPA caso o nome resultante seja muito longo. Aqui estão exemplos tanto da conversão JPA quanto do uso de JPQL. Um exemplo consta na Listagem 1.7.

Listing 1.7 – Exemplos de Repositórios

```
1  public interface PatientRepository extends JpaRepository<Patient,
       Long> {
2      // Exemplo da conven  o JPA
3      Page<Patient> findAllByActivatedTrue(Pageable pageable);
4
5      // Exemplo da utiliza  o de JPQL
6      @Query("""
7              SELECT p.activated FROM Patient p WHERE p.id = :id
8              """)
9      Boolean findActivatedById(@Param("id") Long id);
10 }
```

1.11 Services

Todas as entidades devem possuir um serviço responsável por implementar sua regra de negócios. Esses serviços devem estar localizados em um *package* específico para a entidade, com o nome da entidade seguido de `Service`. Por exemplo, para a entidade *Patient*, o serviço correspondente será `PatientService` e ficará dentro do *package* `patient` dentro do *package* `service`.

A implementação de um serviço deve seguir um fluxo específico para cada operação: o serviço recebe um DTO de entrada, comunica-se com o repositório para executar a operação desejada e, em seguida, retorna um DTO contendo os detalhes da entidade. Além disso, o serviço deve ser anotado com `@Service`, injetar os repositórios necessários e possuir listas de validações que serão injetadas por inversão de dependência.

Abaixo, apresentamos um exemplo prático com várias operações de um serviço para a entidade *Patient*. Um exemplo está na Listagem 1.8.

Listing 1.8 – Exemplo de Serviço para a Entidade *Patient*

```
@Service
public class PatientService {

    @Autowired
    private PatientRepository repository;
    @Autowired
    private List<ValidationUpdatePatient> validationUpdatePatient;
    @Autowired
    private List<ValidationDisabledPatient>
       validationDisabledPatients;

    public DataDetailsPatient register(DataRegisterPatient data) {
        Patient patient = new Patient(data);
        repository.save(patient);
        return new DataDetailsPatient(patient);
    }
    public Page<DataDetailsPatient> list(Pageable pageable) {
        return repository.findAllByActivatedTrue(pageable).map(
           DataDetailsPatient::new);
    }
    public DataDetailsPatient details(Long id) {
        Patient patient = repository.getReferenceById(id);
        return new DataDetailsPatient(patient);
    }
    public DataDetailsPatient update(Long id, DataUpdatePatient
       data) {
        validationUpdatePatient.forEach(v -> v.validation(id, data)
           );
        Patient patient = repository.getReferenceById(id);
        patient.updateData(data);
        return new DataDetailsPatient(patient);
    }
    public void disabled(Long id) {
        validationDisabledPatients.forEach(v -> v.validation(id));
        Patient patient = repository.getReferenceById(id);
```

```
32          patient.logicalDelete();
33      }
34  }
```

- **register**: Recebe um `DataRegisterPatient`, cria uma nova entidade `Patient`, salva no repositório e retorna os detalhes do paciente registrado.
- **list**: Retorna uma página de `DataDetailsPatient` contendo todos os pacientes ativados, paginados conforme os parâmetros fornecidos.
- **details**: Recebe um `Long id`, obtém a referência do paciente pelo ID e retorna os detalhes do paciente.
- **update**: Recebe um `Long id` e um `DataUpdatePatient`, valida as atualizações, atualiza os dados do paciente e retorna os detalhes do paciente atualizado.
- **disabled**: Recebe um `Long id`, valida a desativação, realiza a exclusão lógica do paciente.

1.11.1 Services de Classes com composição

Quando uma entidade possui composição de outras entidades, as operações de busca e validação das entidades de composição são realizadas dentro do serviço da entidade fraca (entidades que dependem de outras entidades para existir). Isso garante que todas as operações relacionadas à composição sejam tratadas de forma centralizada e consistente.

No exemplo da Listagem 1.9, o serviço de agendamento (`Appointment`) está realizando o registro de um novo agendamento. Antes de persistir o agendamento, ele executa algumas validações e busca as entidades de composição (como o paciente e o médico) necessárias para criar o agendamento.

Listing 1.9 – Serviço de Agendamento

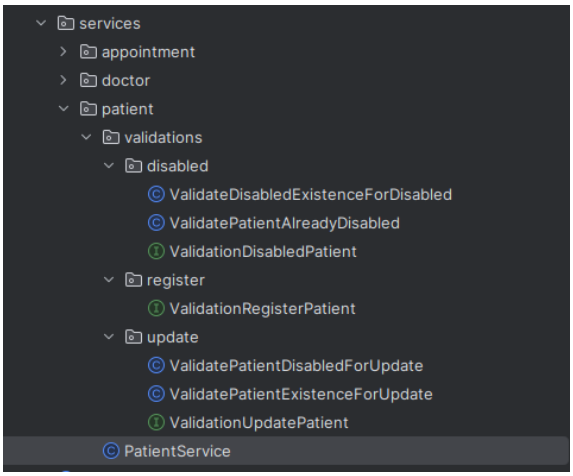
```
1  public DataDetailsAppointment register(DataRegisterAppointment data
       ){
2      validators.forEach(validator -> validator.validation(data));
3      Patient patient = repositoryPatient.getReferenceById(data.
           idPatient());
4      Doctor doctor = chooseDoctor(data);
5      Appointment appointment = new Appointment(null, doctor, patient
           , data.date(),null);
6      repository.save(appointment);
```

```
7       return new DataDetailsAppointment(appointment);
8   }
```

1.11.2 Localização das Classes de Validação

Dentro do *package* `service/nomeDaEntidade`, onde `nomeDaEntidade` é o nome escolhido pelo desenvolvedor, deve haver um *package* `validations`, que conterá *subpackages* específicos para cada tipo de validação, como `register`, `update` e `disabled`. Conforme ilustrado na Figura 4.

Figura 4 – Exemplo de onde devem estar as validações



1.11.3 Interface de validação

Cada *subpackage* terá uma interface responsável por definir a regra do método de validação, como mostrado no exemplo a seguir (Listagem 1.10):

Listing 1.10 – Exemplo de interface de validação

```
1 public interface ValidationUpdatePatient {
2 void validation(Long id, DataUpdatePatient data);
3 }
```

As classes que implementam esta interface devem realizar a validação e lançar uma exceção do tipo `ValidationException` caso a validação falhe. Essa validação será capturada pelo `ErrorsHandler` e retornada ao front-end.

1.11.4 Classes que Implementam as Interfaces de Validação

As classes devem ser marcadas com a *annotation* `@Component` para serem carregadas previamente pelo *Spring Boot* e seguir as regras da interface imple-

mentada, conforme exemplificado a seguir (Listagem 1.11). Dessa forma, elas serão injetadas na lista de validações, conforme demonstrado no exemplo do serviço. É importante escolher nomes específicos e descritivos para as classes, de modo a evitar confusões com outras classes do projeto.

Listing 1.11 – Classe de Validação para Atualização de Paciente

```
@Component
public class ValidatePatientExistenceForUpdate implements
    ValidationUpdatePatient {
     @Autowired
     private PatientRepository repository;
     public void validation(Long id, DataUpdatePatient data) {
         if (!repository.existsById(id)) {
             throw new ValidationException("The required patient
                 does not exist");
         }
     }
}
```

1.12 Controllers

Os controllers devem ser marcados com a anotação `@RestController` para indicar que são controladores REST. Isso é importante porque os controladores REST são especificamente projetados para lidar com requisições HTTP e retornar respostas no formato adequado para comunicação com APIs. Além disso, é necessário o uso de `@RequestMapping` para mapear as requisições para métodos específicos do controller. É recomendado que o `@RequestMapping` inclua uma versão da API para garantir uma gestão mais eficaz da evolução da API. Também é essencial incluir `@SecurityRequirement(name = "bearer-key")` para indicar que a autenticação será realizada por meio de um token de autorização do tipo "Bearer".

É recomendável incluir um Logger nos controllers para registrar informações relevantes sobre as operações realizadas. Isso facilita a depuração e o monitoramento do sistema, permitindo que os desenvolvedores identifiquem problemas e acompanhem o comportamento da aplicação em tempo real.

Os controllers devem ter os serviços necessários injetados por meio da anotação `@Autowired`. Isso permite que os controllers chamem os métodos dos serviços para realizar operações de negócios.

Os DTOs devem ser passados como `@RequestBody` nas requisições HTTP para que os dados sejam recebidos e processados pelo controller. O uso de `@Valid` junto com `@RequestBody` é recomendado para que as validações definidas nos DTOs sejam aplicadas automaticamente pelo Spring. No método de registro (`register`), é recomendável usar `UriComponentsBuilder` para construir a URI do

recurso criado e incluí-lo na resposta. Isso permite que os clientes saibam onde encontrar o recurso recém-criado.

Os métodos que realizam modificações nos dados devem ser marcados com
`@Transactional` para garantir a consistência dos dados e a atomicidade das operações. Eles devem retornar os detalhes do recurso modificado para que os clientes possam confirmar o sucesso da operação e receber informações atualizadas.

Por fim, é recomendado que os *controllers* implementem métodos a fim de utilizar todos os métodos públicos dos serviços que estão chamando, garantindo assim uma cobertura completa das funcionalidades fornecidas pelos serviços.

No *Listing* 1.12 está um exemplo de *controller* implementando esses princípios:

Listing 1.12 – Exemplo de controller

```
@RestController
@RequestMapping("v1/patients")
@SecurityRequirement(name = "bearer-key")
public class PatientControllerV1 {
    private static final Logger log = LoggerFactory.getLogger(
        PatientControllerV1.class);
    @Autowired
    private PatientService service;
    @PostMapping
    @Transactional
    public ResponseEntity<DataDetailsPatient> register(@RequestBody
         @Valid DataRegisterPatient data, UriComponentsBuilder
        uriBuilder) {
         log.info("Registering new patient...");
         DataDetailsPatient details = service.register(data);
         URI uri = uriBuilder.path("/patients/{id}").buildAndExpand(
            details.id()).toUri();
         log.info("New patient registered with ID: {}", details.id()
            );
         return ResponseEntity.created(uri).body(details);
    }
    @GetMapping
    public ResponseEntity<Page<DataDetailsPatient>> listPatients(
        @PageableDefault(size = 10, sort = {"name"}) Pageable
        pageable) {
         log.info("Fetching list of patients...");
         Page<DataDetailsPatient> page = service.list(pageable);
         log.info("List of patients fetched successfully.");
         return ResponseEntity.ok(page);
```

```
23      }
24      @GetMapping("/{id}")
25      public ResponseEntity<DataDetailsPatient> detailsDoctor(
            @PathVariable Long id) {
26          log.info("Fetching details of patient with ID: {}", id);
27          DataDetailsPatient details = service.details(id);
28          log.info("Details of patient with ID {} fetched
                successfully.", id);
29          return ResponseEntity.ok(details);
30      }
31      @PutMapping("/{id}")
32      @Transactional
33      public ResponseEntity<DataDetailsPatient> update(@PathVariable
            Long id, @RequestBody @Valid DataUpdatePatient data) {
34          log.info("Updating patient with ID: {}", id);
35          DataDetailsPatient details = service.update(id,data);
36          log.info("Patient with ID {} updated successfully.", id);
37          return ResponseEntity.ok(details);
38      }
39      @DeleteMapping("/{id}")
40      @Transactional
41      public ResponseEntity<Object> disabled(@PathVariable Long id) {
42          log.info("Disabling patient with ID: {}", id);
43          service.disabled(id);
44          log.info("Patient with ID {} disabled successfully.", id);
45          return ResponseEntity.noContent().build();
46      }
47  }
```

1.13 Testes

1.13.1 Configurações do ambiente de teste

Ter um ambiente de teste bem configurado é fundamental para garantir a qualidade, confiabilidade e a estabilidade do sistema. Um aspecto crítico é o uso de um banco de dados de teste que replica o ambiente de produção. Isso ajuda a garantir que os testes reflitam com precisão o comportamento do sistema em produção, identificando problemas antes que eles impactem os usuários finais.

Um arquivo de propriedades específico para o ambiente de teste, como o `application-test.properties`, permite configurar facilmente o ambiente de teste com o mesmo SGBD (Sistema de Gerenciamento de Banco de Dados) e as mesmas configurações do ambiente de produção. Isso simplifica o processo de

teste e ajuda a manter a consistência entre os ambientes de desenvolvimento, teste e produção.

```
spring.config.activate.on-profile=test
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/shoppingStore_test
spring.datasource.username=${POSTGRES_DATASOURCE_USER}
spring.datasource.password=${POSTGRES_DATASOURCE_PASSWORD}
spring.jpa.hibernate.ddl-auto=create
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.show-sql=false
server.port=8081
```

Este exemplo demonstra como configurar o ambiente de teste para usar o PostgreSQL como banco de dados de teste. As configurações incluem o driver JDBC, a URL do banco de dados, o nome de usuário e senha, além de outras configurações específicas do Hibernate. Utilizar um ambiente de teste adequado, como esse exemplo ilustra, é essencial para garantir testes confiáveis e eficazes.

1.14 Padrões de teste

É recomendável aplicar a ordem randômica em testes de unidade para evitar dependências entre eles e garantir a independência dos resultados. Isso pode ser alcançado aplicando a anotação `@TestMethodOrder(MethodOrderer.Random.class)`. A vantagem dessa abordagem é que os testes serão executados em ordens diferentes a cada execução, ajudando a identificar possíveis falhas relacionadas à ordem de execução.

Ao escrever testes, é importante fornecer nomes significativos para os métodos de teste usando a anotação `@DisplayName`. Isso torna os testes mais legíveis e compreensíveis, facilitando a identificação dos cenários de teste e dos casos de uso cobertos pelos testes.

O uso do *Faker* é altamente recomendado sempre que possível para gerar dados de teste de forma dinâmica e realista. O *Faker* permite criar dados fictícios de maneira rápida e fácil, o que é especialmente útil para cenários de teste que requerem uma grande quantidade de dados. Isso ajuda a aumentar a cobertura dos testes e a garantir que diferentes cenários sejam testados de forma abrangente.Veja o exemplo do (Listing 1.13):

Listing 1.13 – Exemplo de teste de unidade utilizando o Faker

```
@TestMethodOrder(MethodOrderer.Random.class)
@DisplayName("Test entity Product")
class ProductTest {
    private final Faker faker = new Faker();
    @Test
    @DisplayName("Inequality test")
    public void testProductInequality() {
        Product product1 = new Product();product1.setId(faker.
            number().randomNumber());product1.setName(faker.commerce
            ().productName());
        Product product2 = new Product();product2.setId(faker.
            number().randomNumber());product2.setName(faker.commerce
            ().productName());
        assertNotEquals(product1, product2);
    }
}
```

1.14.1 Testes de Repository

Além das anotações mencionadas anteriormente, os testes de repositório devem incluir algumas outras anotações importantes para garantir que os testes sejam executados em um ambiente de teste isolado e controlado. Estas anotações são:

- `@DataJpaTest`: Essa anotação configura o ambiente de teste para testar camadas de persistência JPA. Ela carrega apenas as partes relevantes da aplicação relacionadas à JPA.
- `@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)`: Esta anotação impede que o Spring Boot substitua automaticamente o banco de dados de teste pelo da aplicação. Isso garante que os testes sejam executados no banco de dados configurado no **application-test.properties**.
- `@ExtendWith(SpringExtension.class)`: Essa anotação habilita a integração do Spring com os testes JUnit 5.
- `@ActiveProfiles("test")`: Define o perfil ativo como "test", permitindo a configuração específica para o ambiente de teste.

Essas anotações garantem que os testes de repositório sejam executados em um ambiente de teste isolado e controlado, utilizando o banco de dados de

teste configurado e evitando a persistência de dados no banco de dados de produção. Além disso, é importante limpar a tabela utilizada no teste ao final de cada caso de teste para manter a consistência e a independência dos testes.

No Listing 1.14 está um exemplo de teste de repositório com as anotações mencionadas.

Listing 1.14 – Exemplo de teste de repositório com anotações

```
@DataJpaTest
@TestMethodOrder(MethodOrderer.Random.class)
@ExtendWith(SpringExtension.class)
@ActiveProfiles("test")
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.
    Replace.NONE)
class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private TestEntityManager entityManager;

    @Test
    @DisplayName("Test find by username")
    public void testFindByUsername() {
        // C digo do teste
    }

    // Outros m todos de teste...
}
```

1.14.2 Testes de Service

Os testes de *service* devem utilizar *mocks* para isolar o *service* que está sendo testado. Para isso, são usadas as seguintes anotações:

- `@ExtendWith(MockitoExtension.class)`
- `@InjectMocks`
- `@Mock`

Um exemplo de teste de serviço com *mocks* usando o *Mockito* é apresentado no Listing 1.15:

Listing 1.15 – Exemplo de teste de serviço com mocks usando o Mockito

```
@ExtendWith(MockitoExtension.class)
class AppointmentServiceTest {

    @InjectMocks
    private AppointmentService appointmentService;

    @Mock
    private AppointmentRepository appointmentRepository;

    @Test
    void testRegisterAppointment() {
        Appointment appointment = new Appointment();
        appointment.setId(1L);

        Mockito.when(appointmentRepository.save(Mockito.any(
            Appointment.class))).thenReturn(appointment);

        DataDetailsAppointment result = appointmentService.register
            (new DataRegisterAppointment());

        assertNotNull(result);
        assertEquals(1L, result.getId());
    }

    // Outros m todos de teste...
}
```

1.14.3 Testes de Controllers

Os testes de controllers são importantes para garantir o correto funcionamento das rotas e da lógica de controle da aplicação. Para testar os controllers, geralmente utilizamos o framework MockMvc, que simula requisições HTTP sem a necessidade de iniciar um servidor real.

Para configurar e executar testes de controllers com o MockMvc, utilizamos as seguintes anotações:

- `@SpringBootTest`: Essa anotação carrega a aplicação Spring Boot durante os testes, permitindo a inicialização do contexto da aplicação e a injeção de dependências.
- `@AutoConfigureMockMvc`: Essa anotação configura o MockMvc para ser injetado automaticamente no teste, permitindo a simulação de requisições

HTTP e a validação das respostas.

Um exemplo de configuração e execução de testes de controllers com o MockMvc pode ser visto no Listing 1.16:

Listing 1.16 – Exemplo de teste de controllers com o MockMvc

```
@SpringBootTest
@AutoConfigureMockMvc
class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testGetUserById() throws Exception {
        mockMvc.perform(get("/users/{id}", 1))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(1));
    }

    // Outros m todos de teste...
}
```

Neste exemplo, estamos testando o endpoint GET para obter um usuário por ID. Utilizamos o método `perform` do MockMvc para realizar uma requisição HTTP GET para o endpoint especificado. Em seguida, usamos as expectativas (`andExpect`) para validar o status da resposta (200 OK) e o conteúdo do JSON retornado.