Manual do Desenvolvedor - Sistema Simple

1. Introdução ao Projeto

O **Simple** é um sistema de gestão de pedidos de serviços municipais desenvolvido em Java com Spring Boot 3.2.0. O sistema foi projetado para facilitar a comunicação entre cidadãos e órgãos municipais, permitindo o registro, acompanhamento e resolução de solicitações de serviços públicos.

1.1 Objetivos do Sistema

- Simplificar o processo de solicitação de serviços municipais
- Melhorar a transparência no atendimento às demandas dos cidadãos
- Otimizar a gestão interna dos pedidos pelos servidores municipais
- Fornecer métricas e relatórios sobre o atendimento às solicitações

1.2 Visão Geral da Arquitetura

O Simple utiliza uma arquitetura em camadas (adaptação do padrão MVC para APIs REST), com os seguintes módulos principais:

- **Autenticação**: Gerencia usuários, perfis e controle de acesso
- **Gestão de Pedidos**: Controla o ciclo de vida das solicitações de serviços
- **Gestão de Cidadãos**: Administra dados dos cidadãos que utilizam o sistema
- **Configuração do Sistema**: Permite personalizar parâmetros operacionais

O sistema implementa diversos padrões de projeto, incluindo Repository, DTO, Builder e Injeção de Dependência, para garantir uma base de código modular, testável e de fácil manutenção.

2. Requisitos de Sistema e Ferramentas Necessárias

2.1 Requisitos de Hardware

- Processador: 2 GHz dual-core ou superior
- Memória RAM: 4 GB mínimo (8 GB recomendado)
- Espaço em disco: 1 GB para o código-fonte e dependências

2.2 Software Necessário

- **JDK**: Java Development Kit 17 ou superior
- **Maven**: 3.8.x ou superior para gerenciamento de dependências
- **Git**: Para controle de versão
- **IDE**: Recomendamos IntelliJ IDEA ou Spring Tool Suite (STS)
- **Banco de Dados**: PostgreSQL 13 ou superior
- **Docker** (opcional): Para containerização da aplicação
- **Postman** ou similar: Para testar as APIs REST

2.3 Conhecimentos Recomendados

- Java 8+ (incluindo recursos como Streams, Optional, etc.)
- Spring Framework (especialmente Spring Boot, Spring Data JPA, Spring Security)
- REST APIs e padrões de design

- SQL básico
- Git workflow

3. Configuração do Ambiente de Desenvolvimento

3.1 Instalação do JDK

```
# Para Ubuntu/Debian
sudo apt update
sudo apt install openjdk-17-jdk

# Verificar a instalação
java -version
```

3.2 Instalação do Maven

```
# Para Ubuntu/Debian
sudo apt update
sudo apt install maven

# Verificar a instalação
mvn -version
```

3.3 Instalação do PostgreSQL

```
# Para Ubuntu/Debian
sudo apt update
sudo apt install postgresql postgresql-contrib

# Iniciar o serviço
sudo systemctl start postgresql
sudo systemctl enable postgresql

# Criar banco de dados para o projeto
sudo -u postgres psql
CREATE DATABASE simple_db;
CREATE USER simple_user WITH ENCRYPTED PASSWORD 'sua_senha';
GRANT ALL PRIVILEGES ON DATABASE simple_db TO simple_user;
\q
```

3.4 Clonando o Repositório

```
git clone [URL_DO_REPOSITÓRIO]
cd simple
```

3.5 Configuração da IDE

IntelliJ IDEA

1. Abra o IntelliJ IDEA
2. Selecione “Open” e navegue até a pasta do projeto
3. Selecione “Open as Project”
4. Aguarde a importação e indexação do projeto
5. Verifique se o projeto está configurado para usar o JDK 17

Spring Tool Suite (STS)

1. Abra o STS
2. Selecione “File > Import > Maven > Existing Maven Projects”
3. Navegue até a pasta do projeto e selecione o arquivo pom.xml
4. Clique em “Finish” e aguarde a importação

3.6 Configuração do application.properties

Crie ou edite o arquivo `src/main/resources/application.properties` com as seguintes configurações:

```
# Configurações do banco de dados
spring.datasource.url=jdbc:postgresql://localhost:5432/simple_db
spring.datasource.username=simple_user
spring.datasource.password=sua_senha
spring.datasource.driver-class-name=org.postgresql.Driver

# JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# Configurações do servidor
server.port=8080
server.servlet.context-path=/api

# Configurações de segurança
jwt.secret=chave_secreta_para_tokens_jwt
jwt.expiration=86400000
```

4. Estrutura do Projeto e Organização do Código

O projeto segue uma estrutura de pacotes baseada em funcionalidades e camadas:

```
src/
├── main/
│   ├── java/
│   │   └── br/
│   │       └── gov/
│   │           └── municipio/
│   │               └── simple/
│   │                   ├── SimpleApplication.java
│   │                   ├── config/
│   │                   │   ├── SecurityConfig.java
│   │                   │   └── ...
│   │                   ├── auth/
│   │                   │   ├── controller/
│   │                   │   ├── dto/
│   │                   │   ├── model/
│   │                   │   ├── repository/
│   │                   │   └── service/
│   │                   ├── pedido/
│   │                   │   ├── controller/
│   │                   │   ├── dto/
│   │                   │   ├── model/
│   │                   │   ├── repository/
│   │                   │   └── service/
│   │                   ├── cidadao/
│   │                   │   ├── controller/
│   │                   │   ├── dto/
│   │                   │   ├── model/
│   │                   │   ├── repository/
│   │                   │   └── service/
│   │                   ├── sistema/
│   │                   │   ├── controller/
│   │                   │   ├── dto/
│   │                   │   ├── model/
│   │                   │   ├── repository/
│   │                   │   └── service/
│   │                   └── common/
│   │                       ├── exception/
```

```
│   │                       ├── util/
│   │                       └── validation/
│   └── resources/
│       ├── application.properties
│       ├── application-dev.properties
│       ├── application-prod.properties
│       └── db/
│           └── migration/
└── test/
    └── java/
        └── br/
            └── gov/
                └── municipio/
                    └── simple/
                        ├── auth/
                        ├── pedido/
                        ├── cidadao/
                        └── sistema/
```

4.1 Descrição das Camadas

- **Controller**: Responsável por receber as requisições HTTP e delegar o processamento para os serviços
- **Service**: Contém a lógica de negócio da aplicação
- **Repository**: Interface com o banco de dados usando Spring Data JPA
- **Model**: Entidades JPA que representam as tabelas do banco de dados
- **DTO (Data Transfer Object)**: Objetos para transferência de dados entre camadas
- **Config**: Classes de configuração do Spring
- **Common**: Componentes compartilhados entre os módulos

4.2 Fluxo de Dados

1. O cliente faz uma requisição HTTP para um endpoint
2. O Controller recebe a requisição e converte os parâmetros em DTOs
3. O Controller chama o Service apropriado
4. O Service implementa a lógica de negócio, utilizando Repositories quando necessário
5. O Service retorna os dados processados para o Controller
6. O Controller converte os dados em DTOs de resposta e retorna ao cliente

5. Guia de Estilo e Padrões de Código

5.1 Convenções de Nomenclatura

- **Classes**: PascalCase (ex: `PedidoService`)
- **Métodos e variáveis**: camelCase (ex: `buscarPedidoPorId`)
- **Constantes**: SNAKE_CASE_MAIÚSCULO (ex: `MAX_TENTATIVAS_LOGIN`)
- **Pacotes**: minúsculas, sem underscores (ex: `br.gov.municipio.simple.pedido`)

5.2 Padrões de Código

- Utilize os princípios SOLID
- Prefira injeção de dependência via construtor
- Use Lombok para reduzir código boilerplate
- Documente APIs com Swagger/OpenAPI
- Implemente validação de entrada com Bean Validation
- Use Optional para valores que podem ser nulos
- Implemente tratamento de exceções centralizado

5.3 Exemplo de Controller

```java
@RestController
@RequestMapping("/pedidos")
@RequiredArgsConstructor
public class PedidoController {

    private final PedidoService pedidoService;

    @GetMapping("/{id}")
    public ResponseEntity<PedidoResponseDTO> buscarPorId(@PathVariable Long id) {
        return pedidoService.buscarPorId(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public ResponseEntity<PedidoResponseDTO> criar(@Valid @RequestBody PedidoRequestDTO
        dto) {
        PedidoResponseDTO pedidoCriado = pedidoService.criar(dto);
        URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(pedidoCriado.getId())
                .toUri();
        return ResponseEntity.created(location).body(pedidoCriado);
    }
}
```

5.4 Exemplo de Service

```java
@Service
@RequiredArgsConstructor
public class PedidoServiceImpl implements PedidoService {

    private final PedidoRepository pedidoRepository;
    private final PedidoMapper pedidoMapper;

    @Override
    public Optional<PedidoResponseDTO> buscarPorId(Long id) {
        return pedidoRepository.findById(id)
                .map(pedidoMapper::toResponseDTO);
    }

    @Override
    @Transactional
    public PedidoResponseDTO criar(PedidoRequestDTO dto) {
        Pedido pedido = pedidoMapper.toEntity(dto);
        pedido.setStatus(StatusPedido.ABERTO);
        pedido.setDataCriacao(LocalDateTime.now());

        Pedido pedidoSalvo = pedidoRepository.save(pedido);
        return pedidoMapper.toResponseDTO(pedidoSalvo);
    }
}
```

6. Fluxo de Trabalho de Desenvolvimento (Workflow)

6.1 Fluxo Git

O projeto utiliza o Gitflow como metodologia de controle de versão:

- **main**: Contém o código em produção

- **develop**: Branch de desenvolvimento, base para features
- **feature/xxx**: Branches para novas funcionalidades
- **hotfix/xxx**: Branches para correções urgentes em produção
- **release/x.x.x**: Branches para preparação de releases

6.2 Processo de Desenvolvimento

1. **Planejamento**:
   - Entender os requisitos da tarefa
   - Criar ou atualizar o design técnico, se necessário
2. **Implementação**:
   - Criar uma branch a partir de `develop`: `git checkout -b feature/nova-funcionalidade`
   - Implementar a funcionalidade com testes
   - Fazer commits frequentes com mensagens descritivas
3. **Testes Locais**:
   - Executar testes unitários e de integração
   - Verificar a cobertura de testes
   - Realizar testes manuais, se necessário
4. **Revisão de Código**:
   - Criar um Pull Request (PR) para `develop`
   - Solicitar revisão de outros desenvolvedores
   - Corrigir problemas identificados na revisão
5. **Integração**:
   - Após aprovação, fazer merge do PR para `develop`
   - Verificar se a integração contínua passou com sucesso
6. **Release**:
   - Quando um conjunto de funcionalidades estiver pronto, criar uma branch `release/x.x.x`
   - Realizar testes finais e correções na branch de release
   - Fazer merge para `main` e `develop`

6.3 Padrão de Commits

Utilize commits semânticos para facilitar a compreensão do histórico:

```
<tipo>(<escopo>): <descrição>

[corpo]

[rodapé]
```

Tipos comuns: - **feat**: Nova funcionalidade - **fix**: Correção de bug - **docs**: Alterações na documentação - **style**: Formatação, ponto-e-vírgula, etc; sem alteração de código - **refactor**: Refatoração de código - **test**: Adição ou correção de testes - **chore**: Atualizações de tarefas de build, configurações, etc

Exemplo:

```
feat(pedido): implementa endpoint de cancelamento de pedidos

- Adiciona validação de status atual
- Registra motivo do cancelamento
- Notifica o cidadão por email

Closes #123
```

7. Como Executar e Depurar a Aplicação

7.1 Executando via Maven

```
# Executar a aplicação
mvn spring-boot:run

# Executar com perfil específico
mvn spring-boot:run -Dspring-boot.run.profiles=dev
```

7.2 Executando via IDE

IntelliJ IDEA

1. Abra a classe `SimpleApplication.java`
2. Clique no ícone de execução (triângulo verde) ao lado do método `main`
3. Para configurar perfis, edite a configuração de execução e adicione `-Dspring.profiles.active=dev` em VM options

Spring Tool Suite

1. Clique com o botão direito no projeto
2. Selecione "Run As > Spring Boot App"
3. Para configurar perfis, edite a configuração de execução e adicione `-Dspring.profiles.active=dev` em VM arguments

7.3 Depuração

IntelliJ IDEA

1. Defina pontos de interrupção (breakpoints) clicando na margem esquerda do editor
2. Clique no ícone de depuração (inseto) ao lado do método `main`
3. Use as ferramentas de depuração para inspecionar variáveis, avaliar expressões, etc.

Spring Tool Suite

1. Defina pontos de interrupção clicando na margem esquerda do editor
2. Clique com o botão direito no projeto
3. Selecione "Debug As > Spring Boot App"

7.4 Logs e Monitoramento

- Os logs são gerados no console e no arquivo configurado
- Para ajustar o nível de log, modifique o arquivo `application.properties`:

```
# Configurações de log
logging.level.root=INFO
logging.level.br.gov.municipio.simple=DEBUG
logging.file.name=logs/simple.log
```

- Para monitoramento em tempo real, o projeto inclui o Spring Boot Actuator:
  - Endpoints de saúde: http://localhost:8080/api/actuator/health
  - Métricas: http://localhost:8080/api/actuator/metrics

8. Como Executar Testes

8.1 Executando Testes via Maven

```
# Executar todos os testes
mvn test
```

```
# Executar testes de um módulo específico
mvn test -Dtest=br.gov.municipio.simple.pedido.*

# Executar um teste específico
mvn test -Dtest=br.gov.municipio.simple.pedido.service.PedidoServiceTest

# Gerar relatório de cobertura de testes
mvn verify
```

8.2 Executando Testes via IDE

IntelliJ IDEA

1. Navegue até a pasta de testes
2. Clique com o botão direito na pasta ou classe de teste
3. Selecione “Run Tests” ou “Debug Tests”

Spring Tool Suite

1. Navegue até a pasta de testes
2. Clique com o botão direito na pasta ou classe de teste
3. Selecione “Run As > JUnit Test”

8.3 Estrutura de Testes

- **Testes Unitários**: Testam componentes isoladamente, com mocks para dependências
- **Testes de Integração**: Testam a interação entre componentes
- **Testes de API**: Testam os endpoints REST de ponta a ponta

8.4 Exemplo de Teste Unitário

```java
@ExtendWith(MockitoExtension.class)
class PedidoServiceTest {

    @Mock
    private PedidoRepository pedidoRepository;

    @Mock
    private PedidoMapper pedidoMapper;

    @InjectMocks
    private PedidoServiceImpl pedidoService;

    @Test
    void deveCriarPedidoComSucesso() {
        // Arrange
        PedidoRequestDTO requestDTO = new PedidoRequestDTO();
        requestDTO.setDescricao("Teste");

        Pedido pedido = new Pedido();
        pedido.setDescricao("Teste");

        Pedido pedidoSalvo = new Pedido();
        pedidoSalvo.setId(1L);
        pedidoSalvo.setDescricao("Teste");
        pedidoSalvo.setStatus(StatusPedido.ABERTO);

        PedidoResponseDTO responseDTO = new PedidoResponseDTO();
        responseDTO.setId(1L);
        responseDTO.setDescricao("Teste");
        responseDTO.setStatus(StatusPedido.ABERTO);
```

```
        when(pedidoMapper.toEntity(requestDTO)).thenReturn(pedido);
        when(pedidoRepository.save(any(Pedido.class))).thenReturn(pedidoSalvo);
        when(pedidoMapper.toResponseDTO(pedidoSalvo)).thenReturn(responseDTO);

        // Act
        PedidoResponseDTO resultado = pedidoService.criar(requestDTO);

        // Assert
        assertNotNull(resultado);
        assertEquals(1L, resultado.getId());
        assertEquals("Teste", resultado.getDescricao());
        assertEquals(StatusPedido.ABERTO, resultado.getStatus());

        verify(pedidoRepository).save(any(Pedido.class));
    }
}
```

8.5 Exemplo de Teste de Integração

```
@SpringBootTest
@AutoConfigureMockMvc
class PedidoControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ObjectMapper objectMapper;

    @MockBean
    private PedidoService pedidoService;

    @Test
    void deveCriarPedidoComSucesso() throws Exception {
        // Arrange
        PedidoRequestDTO requestDTO = new PedidoRequestDTO();
        requestDTO.setDescricao("Teste");

        PedidoResponseDTO responseDTO = new PedidoResponseDTO();
        responseDTO.setId(1L);
        responseDTO.setDescricao("Teste");
        responseDTO.setStatus(StatusPedido.ABERTO);

        when(pedidoService.criar(any(PedidoRequestDTO.class))).thenReturn(responseDTO);

        // Act & Assert
        mockMvc.perform(post("/pedidos")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(requestDTO)))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.descricao").value("Teste"))
                .andExpect(jsonPath("$.status").value(StatusPedido.ABERTO.name()));
    }
}
```

9. Como Estender ou Modificar o Sistema

9.1 Adicionando uma Nova Entidade

1. **Criar a classe de modelo (entidade JPA)**

```
@Entity
@Table(name = "categorias")
@Data
```

```java
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Categoria {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String nome;

    @Column(length = 500)
    private String descricao;

    @Column(name = "data_criacao")
    private LocalDateTime dataCriacao;

    @Column(name = "ativo")
    private boolean ativo = true;
}
```

2. **Criar o Repository**

```java
@Repository
public interface CategoriaRepository extends JpaRepository<Categoria, Long> {
    Optional<Categoria> findByNome(String nome);
    List<Categoria> findByAtivo(boolean ativo);
}
```

3. **Criar os DTOs**

```java
@Data
@Builder
public class CategoriaRequestDTO {
    @NotBlank(message = "O nome é obrigatório")
    @Size(min = 3, max = 100, message = "O nome deve ter entre 3 e 100 caracteres")
    private String nome;

    @Size(max = 500, message = "A descrição deve ter no máximo 500 caracteres")
    private String descricao;
}

@Data
@Builder
public class CategoriaResponseDTO {
    private Long id;
    private String nome;
    private String descricao;
    private LocalDateTime dataCriacao;
    private boolean ativo;
}
```

4. **Criar o Mapper**

```java
@Mapper(componentModel = "spring")
public interface CategoriaMapper {
    Categoria toEntity(CategoriaRequestDTO dto);
    CategoriaResponseDTO toResponseDTO(Categoria categoria);
    List<CategoriaResponseDTO> toResponseDTOList(List<Categoria> categorias);
}
```

5. **Criar o Service**

```java
public interface CategoriaService {
    List<CategoriaResponseDTO> listarTodas();
    Optional<CategoriaResponseDTO> buscarPorId(Long id);
```

```
    CategoriaResponseDTO criar(CategoriaRequestDTO dto);
    CategoriaResponseDTO atualizar(Long id, CategoriaRequestDTO dto);
    void excluir(Long id);
}

@Service
@RequiredArgsConstructor
public class CategoriaServiceImpl implements CategoriaService {

    private final CategoriaRepository categoriaRepository;
    private final CategoriaMapper categoriaMapper;

    @Override
    public List<CategoriaResponseDTO> listarTodas() {
        return categoriaMapper.toResponseDTOList(categoriaRepository.findAll());
    }

    @Override
    public Optional<CategoriaResponseDTO> buscarPorId(Long id) {
        return categoriaRepository.findById(id)
                .map(categoriaMapper::toResponseDTO);
    }

    @Override
    @Transactional
    public CategoriaResponseDTO criar(CategoriaRequestDTO dto) {
        Categoria categoria = categoriaMapper.toEntity(dto);
        categoria.setDataCriacao(LocalDateTime.now());
        Categoria categoriaSalva = categoriaRepository.save(categoria);
        return categoriaMapper.toResponseDTO(categoriaSalva);
    }

    @Override
    @Transactional
    public CategoriaResponseDTO atualizar(Long id, CategoriaRequestDTO dto) {
        Categoria categoria = categoriaRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Categoria não
        encontrada"));

        categoria.setNome(dto.getNome());
        categoria.setDescricao(dto.getDescricao());

        Categoria categoriaSalva = categoriaRepository.save(categoria);
        return categoriaMapper.toResponseDTO(categoriaSalva);
    }

    @Override
    @Transactional
    public void excluir(Long id) {
        Categoria categoria = categoriaRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Categoria não
        encontrada"));

        categoria.setAtivo(false);
        categoriaRepository.save(categoria);
    }
}
```

6. Criar o Controller

```
@RestController
@RequestMapping("/categorias")
@RequiredArgsConstructor
public class CategoriaController {

    private final CategoriaService categoriaService;

    @GetMapping
```

```
    public ResponseEntity<List<CategoriaResponseDTO>> listarTodas() {
        return ResponseEntity.ok(categoriaService.listarTodas());
    }

    @GetMapping("/{id}")
    public ResponseEntity<CategoriaResponseDTO> buscarPorId(@PathVariable Long id) {
        return categoriaService.buscarPorId(id)
                .map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping
    public ResponseEntity<CategoriaResponseDTO> criar(@Valid @RequestBody
         CategoriaRequestDTO dto) {
        CategoriaResponseDTO categoriaCriada = categoriaService.criar(dto);
        URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(categoriaCriada.getId())
                .toUri();
        return ResponseEntity.created(location).body(categoriaCriada);
    }

    @PutMapping("/{id}")
    public ResponseEntity<CategoriaResponseDTO> atualizar(
            @PathVariable Long id,
            @Valid @RequestBody CategoriaRequestDTO dto) {
        return ResponseEntity.ok(categoriaService.atualizar(id, dto));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> excluir(@PathVariable Long id) {
        categoriaService.excluir(id);
        return ResponseEntity.noContent().build();
    }
}
```

7. **Adicionar Testes**

9.2 Adicionando um Novo Endpoint a um Controller Existente

1. **Adicionar o método no Service**

```
// Adicionar na interface PedidoService
List<PedidoResponseDTO> buscarPorStatus(StatusPedido status);

// Implementar no PedidoServiceImpl
@Override
public List<PedidoResponseDTO> buscarPorStatus(StatusPedido status) {
    List<Pedido> pedidos = pedidoRepository.findByStatus(status);
    return pedidos.stream()
            .map(pedidoMapper::toResponseDTO)
            .collect(Collectors.toList());
}
```

2. **Adicionar o método no Repository**

```
// Adicionar no PedidoRepository
List<Pedido> findByStatus(StatusPedido status);
```

3. **Adicionar o endpoint no Controller**

```
@GetMapping("/status/{status}")
public ResponseEntity<List<PedidoResponseDTO>> buscarPorStatus(
        @PathVariable StatusPedido status) {
    return ResponseEntity.ok(pedidoService.buscarPorStatus(status));
}
```

4. **Adicionar Testes**

9.3 Modificando uma Funcionalidade Existente

1. **Identificar os componentes afetados**
2. **Atualizar os testes primeiro (TDD)**
3. **Modificar o código**
4. **Executar os testes para garantir que a funcionalidade continua funcionando**
5. **Atualizar a documentação, se necessário**

10. Processo de Build e Implantação

10.1 Build com Maven

```
# Compilar e empacotar a aplicação
mvn clean package

# Pular testes
mvn clean package -DskipTests

# Compilar com perfil específico
mvn clean package -P prod
```

O arquivo JAR resultante estará na pasta `target/`.

10.2 Implantação em Ambiente de Desenvolvimento

```
# Executar o JAR
java -jar target/simple-0.0.1-SNAPSHOT.jar

# Executar com perfil específico
java -jar -Dspring.profiles.active=dev target/simple-0.0.1-SNAPSHOT.jar
```

10.3 Implantação com Docker

1. **Criar um Dockerfile**

```
FROM openjdk:17-jdk-slim

WORKDIR /app

COPY target/simple-0.0.1-SNAPSHOT.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]
```

2. **Construir a imagem Docker**

```
docker build -t simple:latest .
```

3. **Executar o container**

```
docker run -p 8080:8080 -e SPRING_PROFILES_ACTIVE=prod simple:latest
```

10.4 Implantação em Ambiente de Produção

Usando Systemd (Linux)

1. **Criar um arquivo de serviço**

```
sudo nano /etc/systemd/system/simple.service
```

2. **Adicionar a configuração do serviço**

```
[Unit]
Description=Simple - Sistema de Gestão de Pedidos de Serviços Municipais
After=syslog.target network.target

[Service]
User=simple
Group=simple
WorkingDirectory=/opt/simple
ExecStart=/usr/bin/java -jar /opt/simple/simple.jar --spring.profiles.active=prod
SuccessExitStatus=143
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

3. **Habilitar e iniciar o serviço**

```
sudo systemctl enable simple.service
sudo systemctl start simple.service
```

Usando Docker Compose

1. **Criar um arquivo docker-compose.yml**

```
version: '3'

services:
  app:
    image: simple:latest
    ports:
      - "8080:8080"
    environment:
      - SPRING_PROFILES_ACTIVE=prod
      - SPRING_DATASOURCE_URL=jdbc:postgresql://db:5432/simple_db
      - SPRING_DATASOURCE_USERNAME=simple_user
      - SPRING_DATASOURCE_PASSWORD=sua_senha
    depends_on:
      - db
    restart: always

  db:
    image: postgres:13
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=simple_db
      - POSTGRES_USER=simple_user
      - POSTGRES_PASSWORD=sua_senha
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: always

volumes:
  postgres_data:
```

2. **Iniciar os serviços**

```
docker-compose up -d
```

11. Troubleshooting e FAQs

11.1 Problemas Comuns e Soluções

Aplicação não inicia

Problema: A aplicação não inicia e apresenta erros no console.

Soluções: 1. Verifique se o banco de dados está acessível 2. Verifique se as configurações no `application.properties` estão corretas 3. Verifique se todas as dependências foram baixadas corretamente 4. Verifique os logs para identificar o erro específico

Erro de conexão com o banco de dados

Problema: Erro “Could not create connection to database server”

Soluções: 1. Verifique se o serviço do PostgreSQL está em execução 2. Verifique se as credenciais no `application.properties` estão corretas 3. Verifique se o banco de dados existe 4. Verifique se o firewall permite conexões na porta do banco de dados

Erro de autenticação

Problema: Erro “Bad credentials” ao tentar fazer login

Soluções: 1. Verifique se o usuário existe no banco de dados 2. Verifique se a senha está correta 3. Verifique se o usuário está ativo 4. Verifique se o token JWT está configurado corretamente

11.2 FAQs

Como adicionar um novo perfil de usuário?

1. Adicione o novo perfil no enum `Perfil`
2. Atualize a lógica de autorização no `SecurityConfig`
3. Atualize os testes relacionados à segurança

Como alterar o tempo de expiração do token JWT?

Modifique o valor da propriedade `jwt.expiration` no arquivo `application.properties`. O valor é em milissegundos.

Como configurar o envio de e-mails?

Adicione as seguintes configurações no `application.properties`:

```
spring.mail.host=smtp.example.com
spring.mail.port=587
spring.mail.username=seu_email@example.com
spring.mail.password=sua_senha
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

E implemente um serviço de e-mail:

```
@Service
@RequiredArgsConstructor
public class EmailServiceImpl implements EmailService {

    private final JavaMailSender mailSender;

    @Value("${spring.mail.username}")
```

```
    private String from;

    @Override
    public void enviarEmail(String para, String assunto, String conteudo) {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setFrom(from);
        message.setTo(para);
        message.setSubject(assunto);
        message.setText(conteudo);
        mailSender.send(message);
    }
}
```

Como implementar paginação em uma listagem?

Modifique o Repository para estender `PagingAndSortingRepository` e atualize o método no Controller:

```
@GetMapping
public ResponseEntity<Page<PedidoResponseDTO>> listar(
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "10") int size,
        @RequestParam(defaultValue = "id") String sort) {

    Pageable pageable = PageRequest.of(page, size, Sort.by(sort));
    Page<Pedido> pedidos = pedidoRepository.findAll(pageable);

    Page<PedidoResponseDTO> pedidosDTO = pedidos.map(pedidoMapper::toResponseDTO);
    return ResponseEntity.ok(pedidosDTO);
}
```

Como implementar filtros de busca avançados?

Utilize Specification do Spring Data JPA:

```
@Repository
public interface PedidoRepository extends JpaRepository<Pedido, Long>,
        JpaSpecificationRepository<Pedido> {
}

// No service
public Page<PedidoResponseDTO> buscarComFiltros(PedidoFiltroDTO filtro, Pageable pageable)
        {
    Specification<Pedido> spec = Specification.where(null);

    if (filtro.getStatus() != null) {
        spec = spec.and((root, query, cb) -> cb.equal(root.get("status"),
        filtro.getStatus()));
    }

    if (filtro.getDataInicio() != null && filtro.getDataFim() != null) {
        spec = spec.and((root, query, cb) ->
            cb.between(root.get("dataCriacao"), filtro.getDataInicio(),
        filtro.getDataFim()));
    }

    // Mais filtros...

    Page<Pedido> pedidos = pedidoRepository.findAll(spec, pageable);
    return pedidos.map(pedidoMapper::toResponseDTO);
}
```

11.3 Logs e Monitoramento

Como aumentar o nível de log para depuração?

Modifique o arquivo `application.properties`:

```
logging.level.root=INFO
logging.level.br.gov.municipio.simple=DEBUG
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Como monitorar o desempenho da aplicação?

1. Configure o Spring Boot Actuator no `pom.xml`:

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

2. Configure os endpoints no `application.properties`:

```
management.endpoints.web.exposure.include=health,info,metrics,prometheus
management.endpoint.health.show-details=always
```

3. Adicione o Micrometer Prometheus para métricas:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

4. Acesse os endpoints de monitoramento:
    - http://localhost:8080/api/actuator/health
    - http://localhost:8080/api/actuator/metrics
    - http://localhost:8080/api/actuator/prometheus

Este manual foi criado para auxiliar os desenvolvedores do projeto Simple. Para dúvidas ou sugestões, entre em contato com a equipe de desenvolvimento.