Análise Detalhada do Sistema Simple

1. Arquitetura Modular e Padrões de Design

1.1 Arquitetura Geral

O sistema Simple é uma aplicação monolítica desenvolvida com Spring Boot 3.2.0, seguindo uma arquitetura em camadas bem definida. A aplicação utiliza o padrão MVC (Model-View-Controller) adaptado para APIs REST, com as seguintes camadas:

- **Controladores (Controllers)**: Responsáveis por receber as requisições HTTP e delegar o processamento para os serviços.
- **Serviços (Services)**: Contêm a lógica de negócio da aplicação.
- **Repositórios (Repositories)**: Responsáveis pela persistência e recuperação de dados.
- **Entidades (Entities)**: Representam as tabelas do banco de dados.
- **DTOs (Data Transfer Objects)**: Utilizados para transferência de dados entre as camadas.

1.2 Padrões de Design Utilizados

1. **Injeção de Dependência**: Utilizado extensivamente através das anotações `@Autowired` e construtor com `@RequiredArgsConstructor` do Lombok.

```
@Service
@RequiredArgsConstructor
public class AuthService {
    private final UsuarioRepository usuarioRepository;
    private final PerfilRepository perfilRepository;
    private final PasswordEncoder passwordEncoder;
    private final JwtTokenProvider jwtTokenProvider;
    private final AuthenticationManager authenticationManager;

    // Métodos do serviço
}
```

2. **Repository Pattern**: Implementado através das interfaces que estendem `JpaRepository` do Spring Data JPA.

```
public interface PedidoRepository extends JpaRepository<Pedido, UUID> {
    Optional<Pedido> findByCodigoAcompanhamento(String codigo);
    Page<Pedido> findByCidadao(Cidadao cidadao, Pageable pageable);
    Page<Pedido> findByUsuarioCriacao(Usuario usuario, Pageable pageable);
}
```

3. **DTO Pattern**: Utilizado para separar a representação de dados externa da representação interna.

```java
@Builder
public class PedidoResponse {
    private UUID id;
    private String codigoAcompanhamento;
    private String tipoServico;
    private String cidadao;
    private String usuarioCriacao;
    private String usuarioResponsavel;
    private String etapaAtual;
    private String status;
    private LocalDateTime dataInicio;
    private LocalDateTime dataPrevisao;
    private LocalDateTime dataConclusao;
    private String observacoes;
    private BigDecimal valorTotal;
    private String origem;
    private Integer prioridade;
    private LocalDateTime criadoEm;
}
```

4. **Builder Pattern**: Implementado através da anotação `@Builder` do Lombok para construção de objetos complexos.

5. **Filter Chain Pattern**: Utilizado na implementação de segurança com JWT.

```java
@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    // Implementação do filtro
}
```

2. Componentes Independentes

Embora o sistema seja monolítico, ele é organizado em componentes funcionais bem definidos:

2.1 Módulo de Autenticação e Autorização

Responsável pelo controle de acesso ao sistema, incluindo: - Autenticação baseada em JWT - Gerenciamento de usuários e perfis - Controle de permissões baseado em roles

Principais classes: - `SecurityConfig`: Configuração de segurança - `JwtTokenProvider`: Geração e validação de tokens JWT - `JwtAuthenticationFilter`: Filtro para autenticação via JWT - `AuthService`: Serviço de autenticação - `AuthController`: Endpoints de autenticação

2.2 Módulo de Gestão de Pedidos

Núcleo do sistema, responsável pelo gerenciamento dos pedidos de serviços municipais: - Criação e acompanhamento de pedidos - Atualização de status - Consulta de pedidos por diferentes critérios

Principais classes: - `PedidoController`: Endpoints para gestão de pedidos - `PedidoService`: Lógica de negócio para pedidos - `Pedido`: Entidade que representa um pedido - `StatusPedido`: Entidade que representa os possíveis status de um pedido

2.3 Módulo de Gestão de Cidadãos

Responsável pelo cadastro e gerenciamento de cidadãos: - Cadastro de cidadãos - Consulta de cidadãos - Associação de cidadãos a pedidos

Principais classes: - `CidadaoController`: Endpoints para gestão de cidadãos - `CidadaoService`: Lógica de negócio para cidadãos - `Cidadao`: Entidade que representa um cidadão

2.4 Módulo de Configuração do Sistema

Responsável pelas configurações gerais do sistema: - Configuração de tipos de serviços - Configuração de categorias de serviços - Configuração de perfis de usuários

Principais classes: - `ConfiguracaoController`: Endpoints para configurações - `ConfiguracaoService`: Lógica de negócio para configurações - `TipoServico`: Entidade que representa um tipo de serviço - `CategoriaServico`: Entidade que representa uma categoria de serviço

3. Fluxos de Dados e Lógica de Negócio

3.1 Fluxo de Autenticação

1. O usuário envia credenciais (email e senha) para o endpoint `/auth/login`
2. O `AuthController` recebe a requisição e delega para o `AuthService`
3. O `AuthService` autentica o usuário usando o `AuthenticationManager`
4. Se as credenciais forem válidas, o `JwtTokenProvider` gera um token JWT
5. O token é retornado ao usuário para uso em requisições subsequentes

```
public AuthResponse authenticate(AuthRequest request) {
    authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                    request.getEmail(),
                    request.getSenha()
            )
    );
```

```
    var user = usuarioRepository.findByEmail(request.getEmail())
            .orElseThrow();

    user.setUltimoAcesso(LocalDateTime.now());
    usuarioRepository.save(user);

    var jwtToken = jwtTokenProvider.generateToken(user);
    var refreshToken = jwtTokenProvider.generateRefreshToken(user);

    return AuthResponse.builder()
            .token(jwtToken)
            .refreshToken(refreshToken)
            .build();
}
```

3.2 Fluxo de Criação de Pedido

1. O usuário autenticado envia uma requisição para o endpoint `/pedidos` com os dados do pedido
2. O `PedidoController` recebe a requisição e delega para o `PedidoService`
3. O `PedidoService` valida os dados, cria um novo pedido e salva no banco de dados
4. O pedido é retornado ao usuário com um código de acompanhamento gerado automaticamente

```
public PedidoResponse create(PedidoRequest request) {
    Cidadao cidadao = cidadaoRepository.findById(request.getCidadaoId())
            .orElseThrow(() -> new EntityNotFoundException("Cidadão não encontrado"));

    TipoServico tipoServico = tipoServicoRepository.findById(request.getTipoServicoId())
            .orElseThrow(() -> new EntityNotFoundException("Tipo de serviço não encontrado"));

    StatusPedido statusInicial = statusPedidoRepository.findByCodigo("NOVO")
            .orElseThrow(() -> new EntityNotFoundException("Status inicial não encontrado"));

    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    Usuario usuarioLogado = usuarioRepository.findByEmail(authentication.getName())
            .orElseThrow(() -> new EntityNotFoundException("Usuário não encontrado"));

    Pedido pedido = new Pedido();
    pedido.setCidadao(cidadao);
    pedido.setTipoServico(tipoServico);
    pedido.setStatus(statusInicial);
    pedido.setUsuarioCriacao(usuarioLogado);
    pedido.setUsuarioResponsavel(usuarioLogado);
    pedido.setDataInicio(LocalDateTime.now());
```

```
    pedido.setOrigem(request.getOrigem());
    pedido.setPrioridade(request.getPrioridade());
    pedido.setObservacoes(request.getObservacoes());

    // Calcular data de previsão baseada no prazo estimado do tipo de serviço
    if (tipoServico.getPrazoEstimado() != null) {
        pedido.setDataPrevisao(LocalDateTime.now().plusDays(tipoServico.getPrazoEstimado()))
    }

    pedidoRepository.save(pedido);

    return mapToResponse(pedido);
}
```

3.3 Fluxo de Atualização de Status de Pedido

1. O usuário autenticado envia uma requisição para o endpoint `/pedidos/{id}/status/{statusId}`
2. O `PedidoController` recebe a requisição e delega para o `PedidoService`
3. O `PedidoService` atualiza o status do pedido e, se o status for "CONCLUIDO", atualiza a data de conclusão
4. O pedido atualizado é retornado ao usuário

```
public PedidoResponse updateStatus(UUID id, Long statusId) {
    Pedido pedido = pedidoRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Pedido não encontrado"));

    StatusPedido novoStatus = statusPedidoRepository.findById(statusId)
            .orElseThrow(() -> new EntityNotFoundException("Status não encontrado"));

    pedido.setStatus(novoStatus);

    // Se o status for "CONCLUIDO", atualizar a data de conclusão
    if (novoStatus.getCodigo().equals("CONCLUIDO")) {
        pedido.setDataConclusao(LocalDateTime.now());
    }

    pedidoRepository.save(pedido);

    return mapToResponse(pedido);
}
```

4. Integrações com Sistemas Externos

Não foram identificadas integrações diretas com sistemas externos no código analisado. O sistema parece ser autocontido, sem chamadas a APIs externas

ou serviços de terceiros. No entanto, a arquitetura permite a fácil adição de integrações através de:

- Implementação de clientes HTTP usando RestTemplate ou WebClient
- Implementação de clientes Feign para APIs REST
- Adição de listeners para mensageria (como RabbitMQ ou Kafka)

5. Mecanismos de Segurança e Controle de Acesso

5.1 Autenticação baseada em JWT

O sistema utiliza JSON Web Tokens (JWT) para autenticação stateless:

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtAuthFilter;
    private final AuthenticationProvider authenticationProvider;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .csrf(AbstractHttpConfigurer::disable)
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/auth/**",
                                         "/v3/api-docs/**",
                                         "/swagger-ui/**",
                                         "/swagger-ui.html",
                                         "/pedidos/codigo/**",
                                         "/configuracoes").permitAll() // Public
                        .requestMatchers("/favoritos/**").authenticated() // Requer autentic
                        .requestMatchers("/**").authenticated() // Secure API endpoints
                        .anyRequest().denyAll() // Deny any other requests by default
                )
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationP
                .authenticationProvider(authenticationProvider)
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
}
```

5.2 Autorização baseada em Roles

O sistema utiliza roles para controle de acesso a endpoints específicos:

```
@GetMapping
@PreAuthorize("hasAnyRole('ADMINISTRADOR', 'ATENDENTE', 'GESTOR', 'TECNICO')")
public ResponseEntity<Page<PedidoResponse>> findAll(Pageable pageable) {
    return ResponseEntity.ok(pedidoService.findAll(pageable));
}
```

5.3 Configuração de CORS

O sistema implementa uma configuração de CORS para permitir requisições de origens específicas:

```
@Configuration
public class CorsConfig {

    @Bean
    public CorsFilter corsFilter() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.setAllowCredentials(true);
        corsConfiguration.setAllowedOrigins(Arrays.asList("http://localhost:3000", "https://
        corsConfiguration.setAllowedHeaders(Arrays.asList("Origin", "Access-Control-Allow-Or
                "Accept", "Authorization", "Origin, Accept", "X-Requested-With",
                "Access-Control-Request-Method", "Access-Control-Request-Headers"));
        corsConfiguration.setExposedHeaders(Arrays.asList("Origin", "Content-Type", "Accept"
                "Access-Control-Allow-Origin", "Access-Control-Allow-Credentials"));
        corsConfiguration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "O

        UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new UrlBasedCorsCo
        urlBasedCorsConfigurationSource.registerCorsConfiguration("/**", corsConfiguration);

        return new CorsFilter(urlBasedCorsConfigurationSource);
    }
}
```

5.4 Armazenamento Seguro de Senhas

As senhas são armazenadas de forma segura utilizando o algoritmo BCrypt:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

6. APIs Expostas e seus Endpoints

6.1 API de Autenticação

- **POST /auth/login**: Autentica um usuário e retorna um token JWT
- **POST /auth/register**: Registra um novo usuário no sistema

6.2 API de Pedidos

- **GET /pedidos**: Retorna todos os pedidos (paginados)
- **GET /pedidos/{id}**: Retorna um pedido específico pelo ID
- **GET /pedidos/codigo/{codigo}**: Retorna um pedido pelo código de acompanhamento (endpoint público)
- **GET /pedidos/cidadao/{cidadaoId}**: Retorna os pedidos de um cidadão específico
- **GET /pedidos/meus-pedidos**: Retorna os pedidos criados pelo usuário logado
- **POST /pedidos**: Cria um novo pedido
- **PATCH /pedidos/{id}/status/{statusId}**: Atualiza o status de um pedido

6.3 API de Cidadãos

- **GET /cidadaos**: Retorna todos os cidadãos (paginados)
- **GET /cidadaos/{id}**: Retorna um cidadão específico pelo ID
- **POST /cidadaos**: Cria um novo cidadão
- **PUT /cidadaos/{id}**: Atualiza um cidadão existente

6.4 API de Tipos de Serviço

- **GET /tipos-servicos**: Retorna todos os tipos de serviço
- **GET /tipos-servicos/{id}**: Retorna um tipo de serviço específico pelo ID
- **GET /tipos-servicos/categoria/{categoriaId}**: Retorna os tipos de serviço de uma categoria específica
- **POST /tipos-servicos**: Cria um novo tipo de serviço
- **PUT /tipos-servicos/{id}**: Atualiza um tipo de serviço existente
- **PATCH /tipos-servicos/{id}/ativar**: Ativa um tipo de serviço
- **PATCH /tipos-servicos/{id}/desativar**: Desativa um tipo de serviço

6.5 API de Favoritos

- **GET /favoritos**: Retorna os favoritos do usuário logado
- **POST /favoritos**: Adiciona um tipo de serviço aos favoritos
- **DELETE /favoritos/{id}**: Remove um favorito

6.6 API de Configurações

- **GET /configuracoes**: Retorna as configurações do sistema (endpoint público)

7. Conclusão

O sistema Simple é uma aplicação monolítica bem estruturada, seguindo boas práticas de desenvolvimento com Spring Boot. A arquitetura em camadas,

com separação clara de responsabilidades, facilita a manutenção e evolução do sistema.

Os principais pontos fortes identificados são:

1. **Segurança robusta**: Implementação de autenticação JWT e autorização baseada em roles
2. **Arquitetura modular**: Componentes bem definidos com responsabilidades claras
3. **Padrões de design**: Uso consistente de padrões como Repository, DTO e Builder
4. **Documentação da API**: Implementação do OpenAPI/Swagger para documentação automática
5. **Tratamento de exceções**: Implementação de um handler global para tratamento consistente de exceções

Possíveis melhorias:

1. **Implementação de testes**: Não foram identificados testes unitários ou de integração
2. **Cache**: Embora a anotação `@EnableCaching` esteja presente, não foram identificadas implementações de cache
3. **Auditoria**: Implementação de auditoria mais completa para rastreamento de alterações
4. **Validação**: Implementação de validações mais robustas nos DTOs de entrada