RELATÓRIO FINAL - ENGENHARIA DE SOFTWARE

Grupo: Maxley Soares da Costa - 11911BCC038
João Vitor Afonso Pereira - 11911BCC037
Gean Fernandes da Silva - 11811BSI257
Henrique de Moraes Segatto - 11721BSI241
Thalita Alves de Sousa - 11511EMT033

Breve introdução ao produto (Link - descrição completa)

O Sistema de gerenciamento hospitalar (SGH) é um sistema web desenvolvido para auxiliar na gestão de pacientes e funcionários de hospitais públicos e particulares. Entre suas funcionalidades estão: Cadastro de pacientes, funcionários e médicos do hospital. Também é possível fazer o gerenciamento de consultas médicas, insumos hospitalares, leitos e a partir disso fazer relatórios para gerenciar custos do hospital. O sistema também dá total autonomia ao paciente de agendar suas consultas com o médico que mais se sente à vontade. Para uma descrição mais detalhada, veja o link acima.

Porque utilizamos a metodologia SCRUM

Alguns fatores foram levados em conta para a definição da metodologia. Entre eles estavam a grande adoção da metodologia pelo mercado e nossa falta de experiência na programação de forma profissional. O primeiro fator citado nos fez querer adotar a metodologia para que pudéssemos aprender mais e praticar algo que o mercado está adotando em larga escala. O segundo fator nos fez perceber que independente de qual metodologia fossemos adotar, a mesma teria que nos permitir reconhecer erros de forma rápida para que pudéssemos ajustar o mais rápido possível. Isso nos levou ao SCRUM já que o segundo fator pensado é justamente um “pilar” da metodologia.

Como utilizamos a metodologia SCRUM

Para que todos pudessem praticar, adotamos um rodízio nas funções do SCRUM de modo que todos pudessem passar por todas as funções. Com o passar do tempo alguns se identificaram mais com suas funções as quais decidimos tornar fixas. AS definições finais foram:

PO: Thalita
SCRUM Master: Maxley
Dev. Backend: Maxley, João Vitor
Dev. Frontend: Henrique, Gean

Inicialmente tínhamos sprints semanais que começavam todas as quintas feiras, sprint review e planning todas as quartas e nossas dailies eram diárias. Com o final do semestre chegando vimos a necessidade de mudar esses ritos dado que não estávamos conseguindo cumpri-los adequadamente. Depois dos ajustes temos agora:

Sprint - 2 Semanas.
Review - Quartas feiras às 17:30.
Planning - Quartas feiras às 18:00.
Daily - Segunda, Terça e Quinta às 18:00.

O planejamento e organização das tarefas foi feito através do trello. Para cada tarefa a ser feita que previa algum tipo de programação (back ou front) os membros do time discutiam sobre e gerava uma pontuação para a mesma dado a tabela abaixo:

PONTOS X HORAS
1 PONTO = Ajuste pontual.
2 PONTOS = Até 1 hora.
3 PONTOS = Até 2 horas.
5 PONTOS = Até um dia.
8 PONTOS = Vai levar de 2 até 3 dias.
13 PONTOS = Uma semana, metade da Sprint.
21 PONTOS = A Sprint inteira, 2 semanas.

A tabela prevê pontos para cada desenvolvedor, ou seja, um desenvolvedor pode pegar no máximo 21 pontos por sprint, ou 42 pontos um time com 2 desenvolvedores.

O porque adotamos MVC

O padrão MVC possui um design modular que separa a representação da informação da interação do cliente. Esse padrão é separado em 3 camadas Model, View e Controller o nos dá muita praticidade tanto na hora de desenvolver novas funcionalidades quanto na hora de dar manutenção nas funcionalidades existentes. Outra característica do padrão que foi levada em consideração é sua flexibilidade para criar protótipos, permitindo que o design do produto e seu código possa andar de maneira simultânea, facilitando nossa adoção da metodologia scrum.

Arquitetura do Sistema (Link - images)

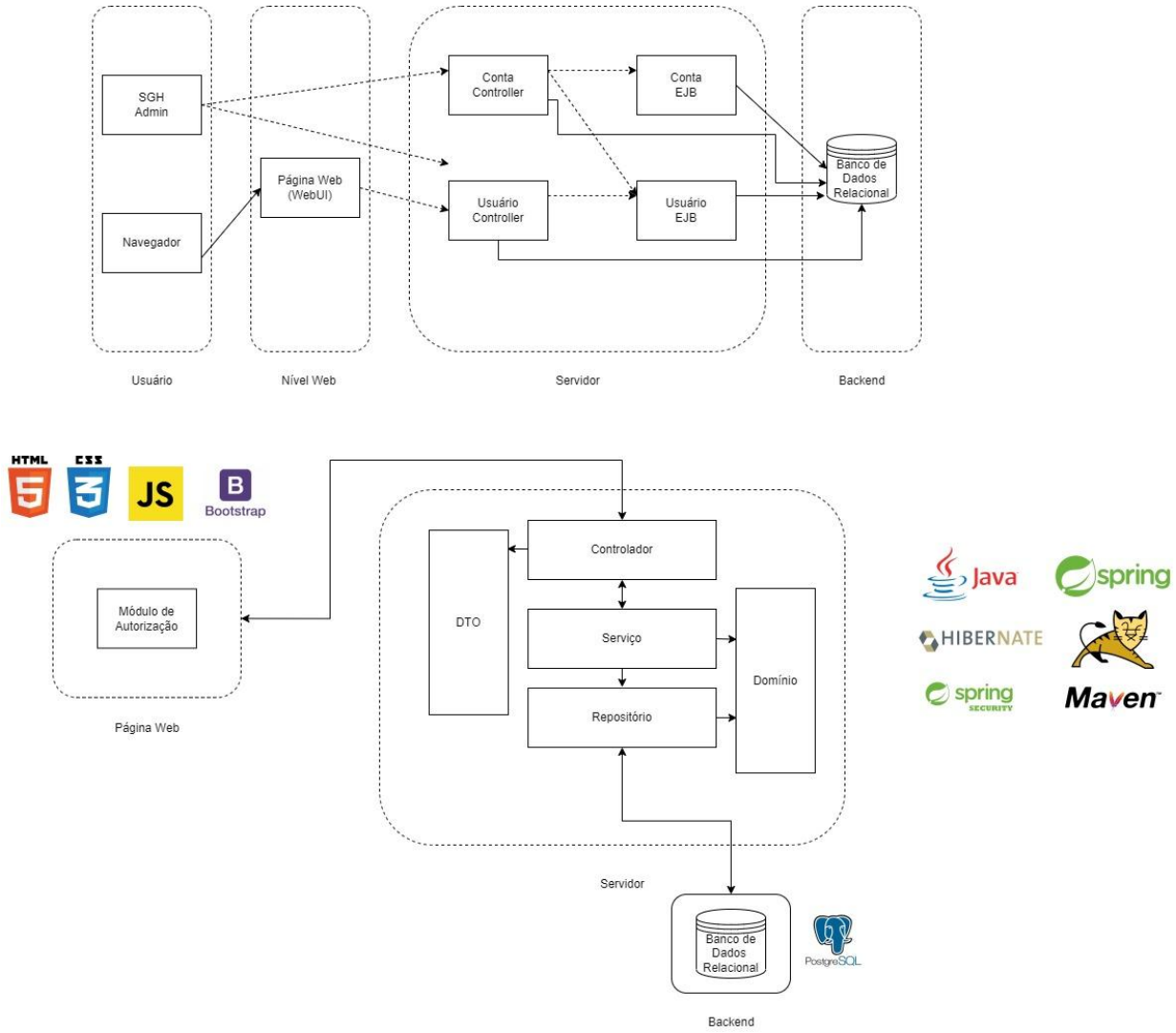

Figura 1: Arquitetura simplificada.

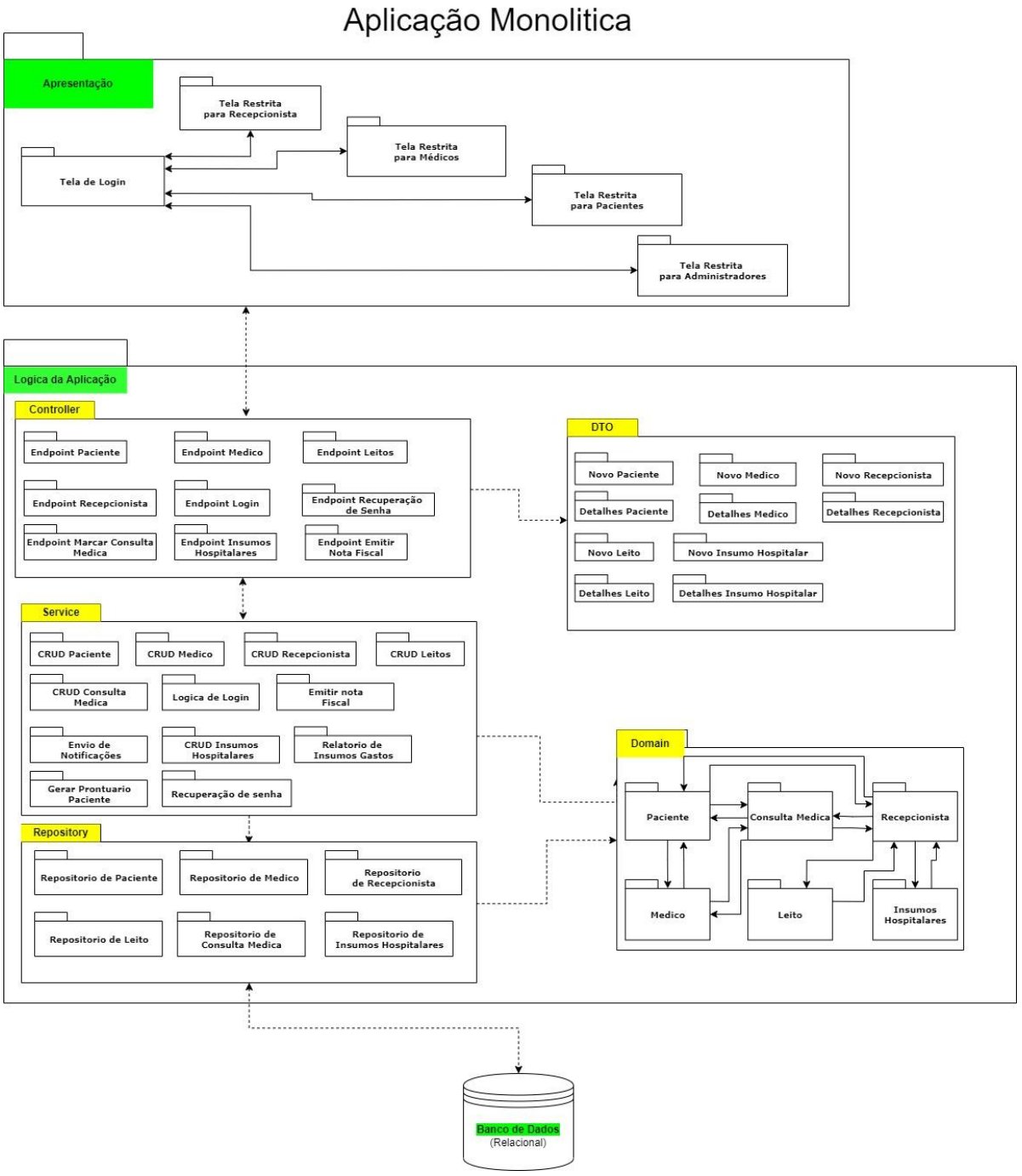

Figura 2: Arquitetura detalhada.

Testes (Link - imagens)

Com o intuito de manter a confiabilidade da aplicação foram feitos testes unitários, sendo que nesse primeiro momento foram implementados testes apenas na camada de service. Não foram feitos testes de integração (com o banco de dados, por exemplo) até o momento. Abaixo temos alguns prints dos códigos.

```java
@DisplayName("MedicoServiceTest")
class MedicoServiceTest extends ApplicationConfigTest {

    @MockBean
    private MedicoRepository medicoRepository;

    @Autowired
    private MedicoService medicoService;

    private Long idMedico = 1L;
    private NovoMedicoRequest novoMedicoRequest = new NovoMedicoRequest(
            nomeMedico: "Strange",
            especialidade: "Cirurgião",
            cpfMedico: "36267116336",
            telefoneMedico: "(34) 98888-8888",
            emailMedico: "drstrange@gmail.com",
            crm: "00000000-0/BR",
            senhaMedico: "123mudar"
    );
    private Medico dadosMedicosAtuais = new Medico(
            nomeMedico: "Stephen",
            especialidade: "Cirurgião",
            cpfMedico: "36267116336",
            telefoneMedico: "(34) 91234-5678",
            emailMedico: "drstrange@gmail.com",
            crm: "00000000-0/BR",
            senhaMedico: "123mudar"
    );
```

Figura 3: Dados de médicos mokados.

```java
    @Test
    @DisplayName("Deve alterar dados do medico quando id do medico valido")
    public void deveAlterarDadosDoMedicoQuandoIdDoMedicoValido() {
        //Given
        Mockito.when(medicoRepository.findById(ArgumentMatchers.eq(idMedico))).thenReturn(Optional.of(dadosMedicosAtuais));
        Mockito.when(medicoRepository.save(ArgumentMatchers.any())).thenReturn(novoMedicoRequest.toModel());

        //When
        Medico medicoAlterado = medicoService.alterarMedico(idMedico, novoMedicoRequest);

        //Then
        Assertions.assertEquals( expected: "Strange", medicoAlterado.getNomeMedico());
        Assertions.assertEquals( expected: "(34) 98888-8888", medicoAlterado.getTelefoneMedico());
    }

    @Test
    @DisplayName("Deve retornar null quando id do medico invalido")
    public void deveRetornarNullQuandoIdMedicoInvalido() {
        //Given
        Mockito.when(medicoRepository.findById(ArgumentMatchers.any())).thenReturn(Optional.empty());

        //When
        Medico medicoAlterado = medicoService.alterarMedico(idMedico, novoMedicoRequest);

        //Then
        Assertions.assertEquals( expected: null, medicoAlterado);
    }
```

Figura 4: Exemplos de testes do service de médico.

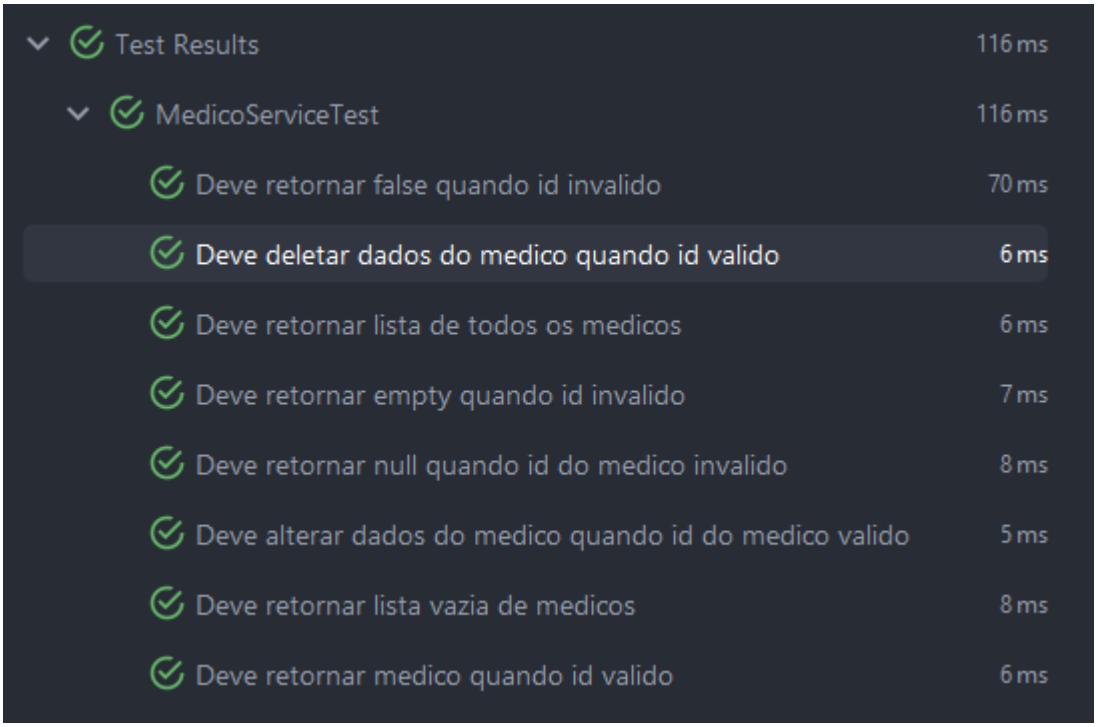

Figura 5: Log de testes.

```java
@Service
public class MedicoService {

    @Autowired
    private MedicoRepository medicoRepository;

    public Medico cadastrarMedico(Medico medico) {
        return medicoRepository.save(medico);
    }

    public List<Medico> listarTodosMedicos() {
        return medicoRepository.findAll();
    }

    public Optional<Medico> detalharMedico(Long idMedico) {
        return medicoRepository.findById(idMedico);
    }

    public Medico alterarMedico(Long idMedico, NovoMedicoRequest novoMedicoRequest) {
        Optional<Medico> medicoEncontrado = medicoRepository.findById(idMedico);

        if (medicoEncontrado.isEmpty()) {
            return null;
        }

        Medico medico = novoMedicoRequest.toModel();
        medico.setIdMedico(medicoEncontrado.get().getIdMedico());

        return cadastrarMedico(medico);
    }
```

Figura 6: Cobertura de testes parte 1.

```
    public boolean deletarMedico(Long idMedico) {
        Optional<Medico> medicoEncontrado = medicoRepository.findById(idMedico);

        if (medicoEncontrado.isEmpty()) {
            return false;
        }

        medicoRepository.deleteById(idMedico);
        return true;
    }


}
```

Figura 7: Cobertura de testes parte 2.

Problemas encontrados e como resolvemos

O primeiro problema encontrado foi a disparidade de conhecimento sobre a stack tecnológica que utilizamos. Alguns desenvolvedores sabiam mais do que outros, então resolvemos fazer pair programming (programação em pares), assim conseguimos disseminar de maneira mais uniforme o conhecimento.

O segundo problema veio quando paramos de obedecer os ritos do SCRUM, isso gerou uma dispersão no time e cada um fazia sua parte. No momento de junção das partes percebemos que algumas definições eram necessárias para que a integração das partes fizessem sentido. Para resolver isso o grupo se juntou e decidiu ajustar os ritos para que fosse possível a participação de todos.

O terceiro problema veio no final do projeto quando vimos um certo gap de conhecimento para que fosse possível integrar back e front, o que gerou algumas necessidades de refatoração no código para resolvermos problemas de cors. Esse problema ainda não foi totalmente resolvido, pois os membros do time ainda estão estudando para que seja possível entender a causa raiz e consequentemente resolvê-la.

O que faríamos diferente

Durante nosso período de desenvolvimento, notamos que os testes começaram a se adequar ao código de modo que alterações no código implicam em alterações nos testes. Para evitar isso, nossa intenção é fazer uso do TDD, onde os testes seriam escritos antes da funcionalidade, e uma vez que o teste esteja sólido e a ideia totalmente estruturada na cabeça, poderíamos desenvolver a funcionalidade.

Observando nossa estrutura de projetos no github notamos que é extremamente necessário separar códigos backend e frontend em repositórios diferentes, pois

mesmo que o intuito inicial seja apenas fazer um trabalho de uma disciplina da faculdade, a navegação por pastas fica comprometida uma vez que a IDE de desenvolvimento escolhida pensa que pastas separadas sejam módulos de um mesmo projeto, quebrando constantemente por achar que se trata de um monolito modular. Outro fator que nos levou a considerar a separação dos projetos backend e frontend foi que um simples git pull ou git push estava gerando alguns conflitos que poderiam ser resolvidos na separação de contexto.

Outra coisa que aprendemos foi que backend e frontend não devem andar em paralelo (sem contratos estabelecidos) na construção das API's pois no momento de integração vimos que muito código precisa ser refeito para ajustar a comunicação. Para evitar isso, estabelecemos contratos entre back e front que devem ser respeitados à risca e enquanto o backend não fica pronto o frontend pode utilizar dados mokados. Outra abordagem é o front pegar uma tarefa de API somente quando o backend tiver entregado todo o fluxo. Nota que essa última abordagem ainda exige um contrato rígido entre back e front.