ShelfShare – Relatório Final

Mickeias Charles de Oliveira Paiva, Tallys Araujo dos Santos, Julia Ogino Andrade Lima, Filype Ottoni Campos, Renan Silvestre de Oliveira Vale, Arthur Alves Lopes, Ana Luísa Santos Correa e Isadora Andrade Santos.

Ciência da Computação

Teste de Software – Zoe Roberto Magalhaes Junior

Universidade Católica de Brasília (UCB)

Brasília – DF – Brasil

1. Introdução

1.1. Descrição do Projeto

O projeto "ShelfShare" consiste no desenvolvimento de uma plataforma web de compartilhamento de livros, projetada como uma biblioteca comunitária. O sistema permite que usuários se cadastrem, façam login, registrem seus próprios livros e solicitem o empréstimo de títulos de outros membros. O desenvolvimento incluiu a criação de um backend (API RESTful em Node.js e MySQL) e um frontend (em React), seguindo o escopo e o protótipo definidos pela equipe. O objetivo principal do projeto foi servir como um sistema completo para a aplicação e documentação de um ciclo integral de testes de software, incluindo testes unitários, de API, de interface e de carga.

1.2. Divisão de Tarefas

Mickeias Charles de Oliveira Paiva	Arquitetura e Desenvolvimento Fullstack do Software
Tallys Araújo	Desenvolvimento do Banco de Dados
Isadora Andrade Santos	Designer UX/UI
Ana Luísa Santos Correa	Designer UX/UI
Ottoni	Product Manager
Júlia	Desenvolvimento Frontend
Renan	Testador
Arthur	Testador

2. Arquitetura do Sistema

A arquitetura do ShelfShare segue o modelo Client-Server desacoplado, uma abordagem moderna padrão para aplicações web.

Neste modelo, o sistema é dividido em duas aplicações completamente independentes que se comunicam através de uma rede via requisições HTTP.

2.1. Aplicação 1: Frontend (Client)

Uma Single Page Application (SPA) em React.

O Frontend é a parte visual da aplicação, responsável por toda a interface com a qual o usuário interage.

Tecnologia Principal	React (construído com Vite).
Responsabilidade	Renderizar as telas (componentes) e gerenciar o estado da interface do usuário.
Roteamento	A navegação entre as páginas é controlada no lado do cliente pelo React Router DOM. Isso permite uma experiencia de SPA (Single Page Application) rápida, onde a página não precisa ser recarregada a cada clique.
Comunicação	Utiliza a biblioteca Axios para fazer reuisiçoes HTTP (GET,POST,etc.) para a API do Backend. É o Axios que "conversa" com o servidor para buscar ou enviar dados.
Estilização	Utiliza CSS Modules para criar estilos escopados, garantindo que o CSS de um componente não afete outros componentes.

2.2. Aplicação 2: Backend (Server)

O Backend é o "cérebro" da aplicação. Ele não possui interface visual e é responsável por toda lógica de negócio, segurança e gerenciamento de dados.

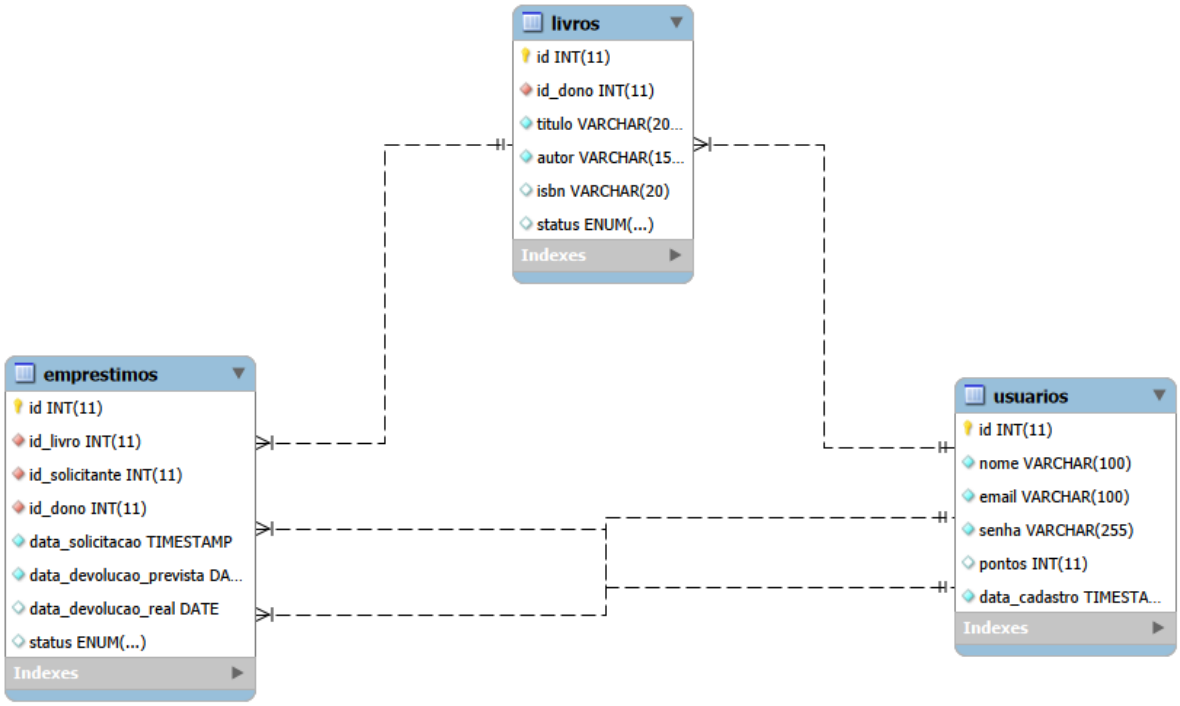
Tecnologia Principal	Node.js com o framework Express.js.
Responsabilidade	Expor uma API RESTful que o frontend (ou qualquer outro cliente, como o Postman) pode consumir.
Roteamento	Define os endpoints (URLs) que o frontend pode acessar.
Lógica	Contém as regras do sistema, como: - Validar se um email já existe; - Criptografar senhas (bcryptjs); - Gerar tokens de autenticação (jsonwebtoken); - Executar transações de empréstimo.

2.3. Persistência (Banco de Dados)

O Backend precisa de um local para armazenar os dados permanentemente.

Tecnologia	MySQL.
Responsabilidade	Armazenar os dados de forma estruturada e relacional. O banco de dados contém as tabelas principais: - `usuarios`: Armazena os dados de perfil, email e senhas criptografadas. - `livros`: Armazena os livros cadastrados, seus donos e seu status. - `empréstimos`: Mantém um histórico de todas as transações.
Comunicação	O Backend (Node.js) é o único que tem permissão para se comunicar diretamente com o banco de dados, executando queries SQL para consultar ou modificar dados.

Modelagem Lógica:



2.4. Fluxo de Autenticação

A segurança e a comunicação entre as partes são gerenciadas por JSON Web Tokens (JWT), o que é fundamental para a arquitetura:

1. Solicitação do Token: O usuário insere e-mail e senha no Frontend (React), que os envia para a rota `POST /api/usuários/login` no Backend.
2. Validação e Geração: O Backend (Node.js) verifica as credenciais no banco MySQL. Se estiverem corretas, ele gera um Token JWT (um "crachá" digital criptografado) e o envia de volta par ao Frontend.
3. Armazenamento do Token: O Frontend (React) recebe o token e o armazena no `localStorage` do navegador.
4. Requisições Autenticadas: Para todas as requisições futuras as rotas protegidas (ex: `GET /api/livros` ou `POST api/empréstimos/solicitar`), o Frontend (Axios) anexa automaticamente o token JWT no cabeçalho `Authorization: Bearer <token>`.
5. Verificação no Backend: Um middleware no Backend intercepta cada uma dessas requisições, verifica a validade do token e, se for autêntico, permite que a requisição prossiga para a lógica de negócio. Se o token for inválido ou ausente, o backend retorna um erro `401 Unauthorized`.

3. Testes

Para garantir a qualidade, funcionalidade e estabilidade do sistema, foi executado um ciclo de testes abrangente, cobrindo seis níveis distintos de validação.

3.1. Teste Unitários

Ferramenta	Jest.
Objetivo	Validar os menores "componentes" de código (funções) do backend em total isolamento.
Descrição	Focamos em validar as regras de negócio puras na suíte `validators.test.js`. Testamos as funções `isValidEmail` (para formatos válidos e inválidos) e `isStrongPassword` (para a regra de 6 caracteres), garantindo que a lógica de validação de dados estava correta.
Resultado	Todos os 8 testes unitários passaram.

3.2. Testes de API

Ferramenta	Postman
Objetivo	Validar o "contrato" da API RESTful, garantindo que cada *endpoint* respondia corretamente a requisições HTTP.
Descrição	Criamos uma coleção no Postman para testar as rotas principais. Validamos o `POST /api/usuarios/cadastro` (retornando `201 Created`) e o `GET /api/livros`, onde confirmamos a segurança (retornando `401 Unauthorized` sem token) e o sucesso (retornando `200 OK` com um token válido).
Resultado	Todos os testes passaram.

3.3. Testes de Integração de Componentes

Ferramenta	React (Axios) e Node.js (Express)
Objetivo	Validar se os componentes de software desenvolvidos separadamente (o frontend em React e o backend em Node.js) conseguiam comunicar-se e funcionar juntos corretamente.
Descrição	O foco deste teste foi a "conversa" entre as partes. Validamos, por exemplo, o fluxo de login: o componente React `Login.jsx` (Frontend) se integra com o serviço `api.js` (Axios) para enviar uma requisição `POST` para a API (Backend), que por sua vez se integra com o banco de dados.
Resultado	A integração foi validada como funcional, o que foi posteriormente confirmado pelos testes de interface automatizados.

3.4. Teste de Interface

Ferramenta	Robot Framework (com SeleniumLibrary)
Objetivo	Validar o fluxo completo do usuário (End-to-End) através da interface gráfica.
Descrição	Criamos scripts de automação (`.robot`) para os fluxos críticos: 1. `login.robot`: Verificou se um usuário com credenciais válidas

	conseguia preencher os campos, clicar em "Entrar" e ser redirecionado para o `/dashboard`. 2. `cadastro.robot`: Verificou se um novo usuário (com e-mail único) conseguia preencher o formulário, clicar em "Cadastrar" e ser redirecionado para a página de login.
Resultado	Ambos os testes de interface passaram, servindo como a prova final de que todos os componentes estavam integrados e funcionando.

3.5. Teste de Carga

Ferramenta	Apache JMeter
Objetivo	Avaliar o desempenho e a estabilidade da API do backend sob uma carga de usuários esperada e realista.
Descrição	Focamos o teste na rota `POST /api/usuarios/cadastro`, pois ela envolve operações de escrita no banco de dados. Simulamos uma carga de 50 usuários cadastrando-se simultaneamente, com um "ramp-up" (período de subida) de 5 segundos.
Resultado	O sistema suportou a carga esperada perfeitamente. O "Summary Report" do JMeter registrou 0% de taxa de erro para os 50 usuários, com um tempo médio de resposta baixo, validando que o sistema é estável sob condições normais de uso.

3.6. Testes de Estresse

Ferramenta	Apache JMeter
Objetivo	Identificar o ponto de quebra (breakpoint) do sistema, submetendo a API a uma carga de usuários extrema e não realista.
Descrição	Usamos o mesmo cenário do teste de carga (`POST /api/usuarios/cadastro`), mas aumentamos drasticamente a simulação para 500 usuários em um "ramp-up" de 10 segundos.
Resultado	O sistema atingiu seu ponto de quebra. A taxa de erro subiu significativamente (de 0% para 21.20%), e o tempo médio de resposta aumentou consideravelmente. Este teste foi um sucesso em identificar o limite de desempenho da aplicação no ambiente de desenvolvimento, mostrando que otimizações seriam necessárias para escalar além desse ponto.