RBEGDR

Revista Brasileira de Estudos de Gestão e Desenvolvimento Regional

v. 05, n. 02 - 2025

Seção de Artigos

DESENVOLVIMENTO DE UM SISTEMA WEB PARA GESTÃO DE PESSOAS

DEVELOPMENT OF A WEB SYSTEM FOR PEOPLE MANAGEMENT

DESARROLLO DE UN SISTEMA WEB PARA LA GESTIÓN DE PERSONAS

Lucas Nunes Batista
https://orcid.org/0009-0006-7759-5403
UNEMAT – Universidade do Estado de Mato Grosso
e-mail: lucasnunesb6@gmail.com

Ivan Luiz Pedroso Pires
https://orcid.org/ 0000-0002-1380-082X
UNEMAT – Universidade do Estado de Mato Grosso
e-mail: ivanpires@unemat.br

Submissão em: 18/08/2025 Aceito em: 25/08/2025

RESUMO
Este trabalho apresenta o desenvolvimento de um sistema web para automatizar a separação e distribuição de holerites. A solução utiliza *back-end* em NodeJS (Express), banco de dados PostgreSQL e serviço externo em Python para processamento e criptografia de arquivos PDF. Cada funcionário acessa seus documentos via autenticação com permissões específicas. Testes ponta a ponta (E2E) validaram os principais fluxos como publicação, fracionamento e acesso seguro. Apesar da ausência de testes unitários, a arquitetura atende ao propósito e fornece base para futuras melhorias.
Palavras-chave: Automação, Holerites, Testes

ABSTRACT
This work presents the development of a web-based system to automate the separation and distribution of payslips. The solution uses a NodeJS (Express) backend, a PostgreSQL database, and an external Python service for processing and encrypting PDF files. Each employee accesses their documents through authentication with specific permission. End-to-end (E2E) tests validated key flows such as publication, file splitting, and secure access. Despite the absence of unit tests, architecture fulfills its purpose and provides a foundation for future improvements.
Keywords: Automation, Payslips, Testing

RESUMEN
Este trabajo presenta el desarrollo de un sistema web para automatizar la separación y la distribución de recibos de nómina. La solución utiliza un back-end en Node.js (Express), una base de datos PostgreSQL y un servicio externo en Python para el procesamiento y el cifrado de archivos PDF. Cada empleado accede a sus documentos mediante autenticación con permisos específicos. Las pruebas de extremo a extremo (E2E) validaron los principales flujos, como la publicación, el fraccionamiento y el acceso seguro. Apesar de la ausencia de pruebas unitarias, la arquitectura cumple su propósito y ofrece una base para mejoras futuras.
Palabras clave: Automatización, Recibos de nómina, Pruebas

1 INTRODUÇÃO

A gestão de pessoas é uma área do conhecimento voltada para o trabalho de gerir os ativos humanos de uma organização de maneira inteligente e coerente a partir da perspectiva de que os funcionários são parceiros de tomada de decisão (Oliveira e Oliveira, 2018, p. 11).

Um dos contextos no qual esse tipo de gestão acontece é o Departamento Pessoal das empresas. Este departamento é responsável por uma série de atividades, dentre elas, a produção, distribuição e documentação de arquivos como folhas de pagamento, informe de rendimentos, contratos de trabalho e exames laborais, por exemplo.

Nesse contexto, uma situação corriqueira em empresas é o desperdício de tempo e energia com a distribuição manual dos holerites aos funcionários.

Sem apoio de um *software*, é comum os analistas de departamento pessoal necessitarem imprimir várias páginas de folhas de pagamento, separar manualmente cada página em via do funcionário e via da empresa, e distribuir os documentos de setor em setor.

Este artigo apresenta um sistema de distribuição das folhas de pagamento para empregados e gestores da organização. Dessa forma, esse trabalho discorre sobre a construção desse *software* e, embora o processo completo de gestão de pessoas envolva uma pletora de funcionalidades, o objetivo do sistema construído será o controle do arquivamento, a distribuição e a gestão dos holerites. Para que isso fosse possível, o contexto do setor de Recursos Humanos de uma empresa média foi utilizado como ponto de partida para o levantamento de requisitos, modelagem de dados e concepção da arquitetura de *software* que seria utilizada.

A automação do processo de envio de folhas salariais na empresa justifica-se na redução da possibilidade de erro humano, do desperdício de tempo e da ineficiência presente no método manual e repetitivo. Além disso, espera-se que a digitalização contribua para maior agilidade na comunicação interna e melhoria na organização dos documentos.

O restante deste artigo está organizado da seguinte forma: a seção “Soluções relacionadas” apresenta uma análise sobre as ferramentas comerciais que possuem proposta semelhante ao sistema em questão; a seção “Tecnologias envolvidas” apresenta os diferentes utilitários e meios usados para a construção do sistema; na seção FolhaNet, os requisitos e arquitetura do *software* são apresentados; na seção “Desenvolvimento” são mostrados os procedimentos realizados para a codificação do *front-end* e do *back-end*; a seção “Testes” demonstra uma avaliação das principais funcionalidades do sistema e, por fim, “Resultados obtidos” apresenta uma análise dos testes realizados.

2 REFERENCIAL TEÓRICO

2.1 Soluções relacionadas

Dentro do mercado de sistemas voltados à gestão de recursos humanos há uma variedade de opções, cada qual com suas características, pontos positivos e negativos. Um exemplo de solução robusta é o EPM da Tecsmart, o qual abrange funcionalidades como controle de ponto avançado (marcação móvel e geolocalização), automação de processos operacionais, disponibilização de módulos destinados a saúde,

segurança, gestão de currículos e até mesmo análise de indicadores via ferramentas de *Business Inteligence*.

Um outro exemplo é o sistema Kairos da Dimep Sistemas, que abrange funcionalidades como dashboards com gráficos em tempo real, assinatura digital no cartão de ponto, cadastro de horários e escalas, gerenciamento de banco de horas, relatórios fiscais e gerenciais sob demanda e integração com sistemas de folha de pagamento. O processamento desses dados operacionais e a criação de holerites a partir deles geralmente é feito em sistemas contábeis, os quais exportam as folhas de pagamento como resultado.

Entretanto, a distribuição desses documentos ainda termina no processo manual de impressão e entrega para cada pessoa. Mesmo em sistemas que se integram a módulos de geração de holerites (como o Kairos), não há opção de plano que apenas envolva a coleta de marcações de ponto e automação da distribuição de documentos. Dessa maneira, há situações em que empresas menores possuem um sistema de coleta de ponto - para posterior envio à uma contabilidade para conseguirem as folhas de pagamento - e desejam apenas um módulo de arquivamento e distribuição de documentos, as quais não aproveitam todas as funcionalidades entregues mesmo nos planos mais básicos dessas soluções do mercado. Tal problema seria resolvido com uma solução mais pontual dentro de uma plataforma web que seja personalizável sob demanda da organização.

Considerando esse cenário, optou-se por buscar a construção de uma solução para esses cenários de necessidade mais específica de pequenas empresas. Desse modo, o objetivo traçado constitui-se na criação de um *software* utilitário que automatize a distribuição dos holerites e, futuramente, seja capaz de ser expandido para abranger novas funcionalidades de gestão de RH.

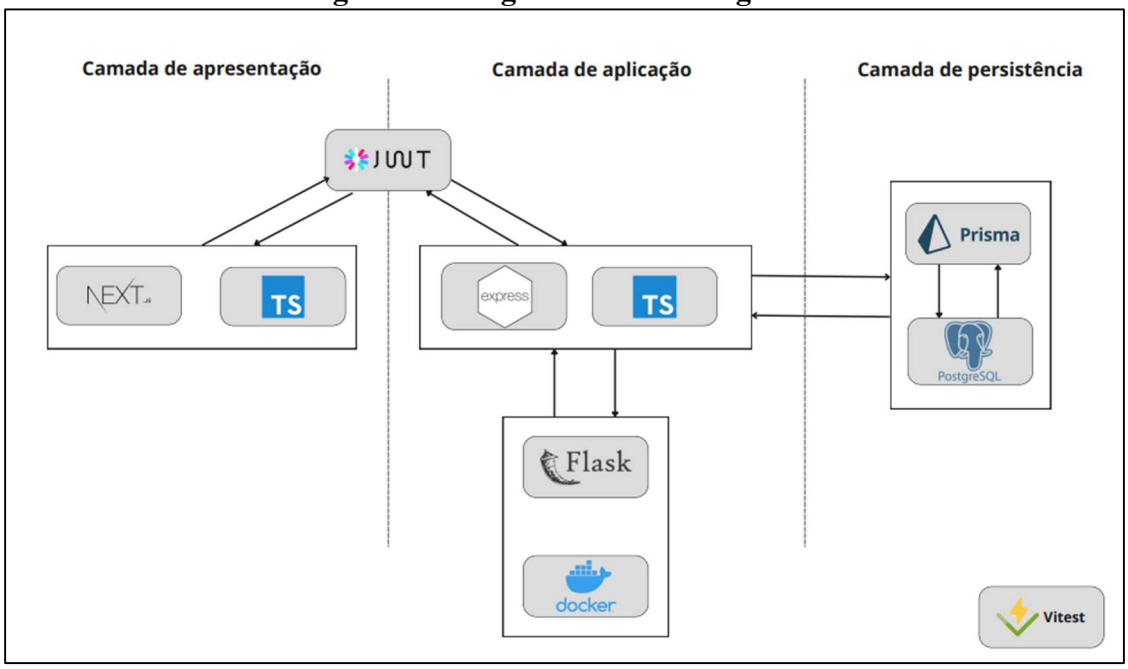
2.2 Tecnologias envolvidas

Para o desenvolvimento do *software*, foi necessário utilizar várias ferramentas de codificação, as quais estão listadas abaixo. Na Figura 1, é possível verificar em qual camada da aplicação elas atuam e como interagem entre si. Os detalhes sobre a arquitetura de camadas utilizada serão abordados na sessão Arquitetura.

- *Prisma*: ORM que facilita o acesso a bancos relacionais (Prisma, 2025). Foi escolhido pela facilidade de uso, suporte ativo e familiaridade do autor;
- *PostgreSQL*: SGBD relacional *open-source* com suporte avançado a dados (PostgreSQL, 2025). Selecionado pela robustez, comunidade ativa e experiência prévia do autor;
- *ExpressJS*: *Framework* para NodeJS com diversas ferramentas para gerenciamento de rotas e requisições (ExpressJS, 2025). Utilizado por sua flexibilidade e grande adoção no mercado;
- *Typescript: Superset* de JavaScript com tipagem estática (Typescript, 2025). Adotado por melhorar a previsibilidade e a detecção de erros em tempo de desenvolvimento;
- *JWT:* Padrão seguido para autenticação via tokens assinados (JWT, 2025). Escolhido por sua leveza e integração facilitada com o NodeJS;
- *NextJS: Framework React* focado em renderização do lado do servidor e geração de páginas estáticas (NextJS, 2025). Utilizado pela eficiência na construção de interfaces reativas e suporte da comunidade;
- *Flask:* Microframework Python para construção de aplicações web (Flask, 2025). Aplicado pela compatibilidade com o Pytesseract e por ser leve e flexível.

- *Vitest: Framework* de testes utilizado por oferecer execução rápida, feedback ágil em tempo de desenvolvimento e suporte nativo para Typescript (Vitest, 2025);
- *Docker: Docker* é um conjunto de utilitários PaaS (*Platform as a Service*) que usa virtualização em nível de sistema operacional para encapsular *software* em contêineres portáteis e isolados. Foi aplicado devido à sua utilidade para empacotar serviços necessários à execução dos testes ponta a ponta da aplicação.

Figura 1 – Diagrama de tecnologias utilizadas



Fonte: Autor, 2025

Dessa forma, com o uso dessas ferramentas, será possível construir um utilitário eficaz para importar as folhas de pagamento, armazená-las de forma segura e distribuir os holerites dos funcionários de forma automatizada.

3 PROCEDIMENTOS METODOLÓGICOS

3.1 FolhaNet

O FolhaNet é um sistema focado em atender à necessidade de gerenciar e automatizar a distribuição de folhas salariais dentro de uma empresa, de forma a evitar o esforço manual de funcionários do setor de Recursos Humanos para entregar esses documentos aos funcionários. Utilizando esse *software*, é possível, com poucos cliques, processar um arquivo PDF que contém todas as folhas de pagamento, separar o holerite de cada funcionário e enviar esse arquivo para o colaborador acessar em uma interface web. Além disso, esse sistema registra em tempo real quais holerites foram visualizados, permitindo rastreabilidade e controle aos analistas.

Nos tópicos seguintes, serão abordados os requisitos que orientaram o desenvolvimento do FolhaNet, a sua arquitetura e a metodologia adotada durante o seu desenvolvimento.

3.2 Requisitos

Dentre as várias etapas relacionadas à engenharia de *software*, há o levantamento de requisitos, o qual se concentra em dois tipos principais: requisitos funcionais e requisitos não funcionais. Os requisitos funcionais determinam as operações que o sistema deve ser capaz de realizar, enquanto os não funcionais descrevem atributos de qualidade, desempenho, segurança e outras restrições relevantes (Morais e Zanin, 2020, p. 96). O contexto de uma empresa de pequeno porte com estrutura de RH convencional foi utilizado como base para o levantamento dos requisitos. É possível visualizar os requisitos funcionais no Quadro 1.

Quadro 1 – Requisitos funcionais do sistema

Identificação	Descrição
RF1	O usuário deve poder acessar seus holerites para visualizar seu salário, benefícios e descontos mensais.
RF2	O supervisor deve poder acessar os holerites da equipe para revisar as informações salariais.
RF3	O sistema deve ser capaz de receber um arquivo *PDF* que contenha todos os holerites, separar as folhas de pagamento por usuário e por ID.
RF4	Deve haver uma inscrição nos holerites relativa à competência de cada um visando a visualização destes conforme o seu período.
RF5	O sistema deve oferecer um *dashboard* com funções C.R.U.D. (*Create, Read, Update, Delete*) tanto para usuários quanto para setores.
RF6	Deve existir um componente de pesquisa por nome, *e-mail* ou setor no *dashboard* de usuários, facilitando a localização de registros.
RF7	Os campos de formulários dos usuários devem possuir validações padrão, como formato de *e-mail*, código de folha e padrão seguro de senha.
RF8	Caso o *upload* do arquivo *PDF* único traga novos funcionários, o sistema deve gerar um pré-cadastro desses usuários com o nome e código de folha presentes em seus holerites.

Fonte: Autor, 2025.

Além disso, foram levantados os requisitos não funcionais, os quais estão presentes no Quadro 2.

Quadro 2 – Requisitos não funcionais do sistema

Identificação	Descrição
RNF1	O sistema deve ser executado em ambiente de produção no formato Javascript, a partir da compilação dos arquivos Typescript.
RNF2	O *software* deve assegurar que a fila de envio de *e-mails* continue funcionando, mesmo que algum *e-mail* inexistente esteja presente na lista.
RNF3	A aplicação deve utilizar um banco de dados relacional.
RNF4	O sistema deve utilizar autenticação e autorização baseada no padrão JWT (JSON Web Token).
RNF5	A estilização do projeto deve ser realizada com uma tecnologia que permita a compilação completa do CSS durante a *build* do projeto.
RNF6	O sistema deve adotar um *framework* de *back-end* que ofereça flexibilidade arquitetural, permitindo o uso de diferentes ferramentas, pacotes e padrões de *design*
RNF7	O *software* deve ser versionado utilizando Git, permitindo um controle mais detalhado sobre o histórico de alterações do projeto.

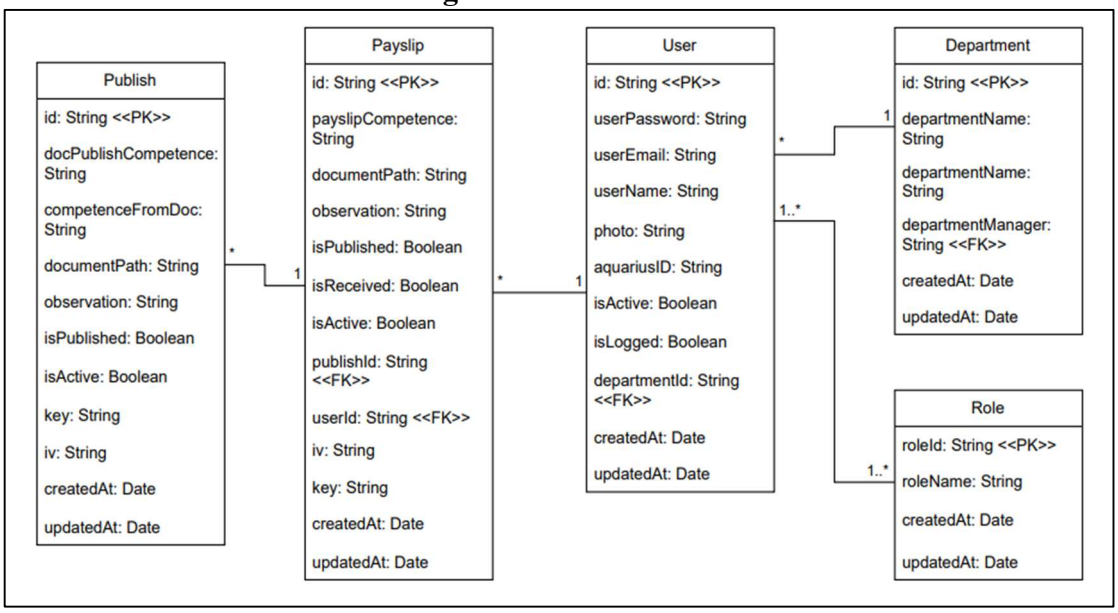
RNF8	O sistema deve ser disponibilizado em uma *intranet* corporativa por meio de um servidor *web* Nginx.

Fonte: Autor, 2025.

Além dos requisitos funcionais e não funcionais, foi necessário reconhecer os requisitos de dados. Nesse sentido, a aplicação possui cinco entidades principais, sendo elas *User, Role, Department, Publish e Payslip*. O modelo de dados pode ser visualizado na Figura 2.

- A entidade User representa um funcionário e possui os seguintes relacionamentos: Relacionamento "N para M" (muitos para muitos) com a entidade Role, ou seja, o usuário pode ter um ou mais papéis e uma *Role* pode ter vários usuários; Relacionamento "1 para 1" com a entidade *Department*, podendo o usuário pertencer a apenas um departamento; Relacionamento "1 para N" com a entidade *Publish*, caso tenha permissão "RH", permitindo que um usuário com essa permissão publique vários documentos; Relacionamento "1 para N" com a entidade *Payslip*, ou seja, um usuário pode ter vários holerites.
- A entidade Department representa um setor e tem um relacionamento "1" para "N" com a entidade *User*, significando que um único departamento pode ter vários funcionários.
- A entidade *Role* representa um papel ou permissão dentro do *software* e tem um relacionamento "N para M" com a entidade *User*.
- A entidade Publish representa uma publicação (um documento com vários holerites) e tem um relacionamento "1" para "N" com a entidade *Payslip*, o que significa que uma publicação pode ter várias folhas salariais associadas.
- A entidade *Payslip* representa uma folha de pagamento no sistema e possui os seguintes relacionamentos: Relacionamento "1" para "1" com a entidade *User*, ou seja, cada folha de pagamento pertence a um único usuário; Relacionamento "1" para "1" com a entidade *Publish*, ou seja, cada folha de pagamento está associada a uma única publicação.

Figura 2 – Modelo de dados



Fonte: Autor, 2025.

3.3 Arquitetura

A abordagem escolhida para organizar os componentes da aplicação foi a arquitetura em camadas. Nesse paradigma, busca-se particionar a complexidade do sistema em componentes menores - as camadas - que reúnem uma ou várias classes. As camadas são organizadas de forma hierárquica, de modo que uma camada somente pode usar serviços - como métodos, objetos, classes - da camada imediatamente inferior.

As vantagens principais dessa maneira de projetar o *software* envolvem a facilidade de trocar uma camada por outra e a flexibilidade no reuso dos utilitários de uma camada em outra mais superior (Valente, 2020). Neste trabalho, optou-se pela construção do *software* com a abordagem de arquitetura em três níveis. Tais níveis são:

- Camada de apresentação: responsável pela interação com o usuário. Trata da exibição de informações, coleta e processamento de entradas e eventos de interface tais como cliques, marcações, entre outros (Valente, 2020). Neste trabalho, essa camada é representada pelos componentes *React* executados dentro do *framework* NextJS. A biblioteca *React* gerencia a renderização da página e o NextJS controla o roteamento;
- Camada de aplicação: camada responsável por implementar as regras de negócio da aplicação (Valente, 2020). No contexto deste trabalho, esse componente arquitetural é representado por toda lógica encapsulada no *back-end* voltada tanto à manipulação de informações quanto à interação com o banco de dados;
- Banco de dados: é a camada que persiste os dados processados pelo sistema (Valente, 2020). No *software* em questão, é representada pelo banco PostgreSQL;

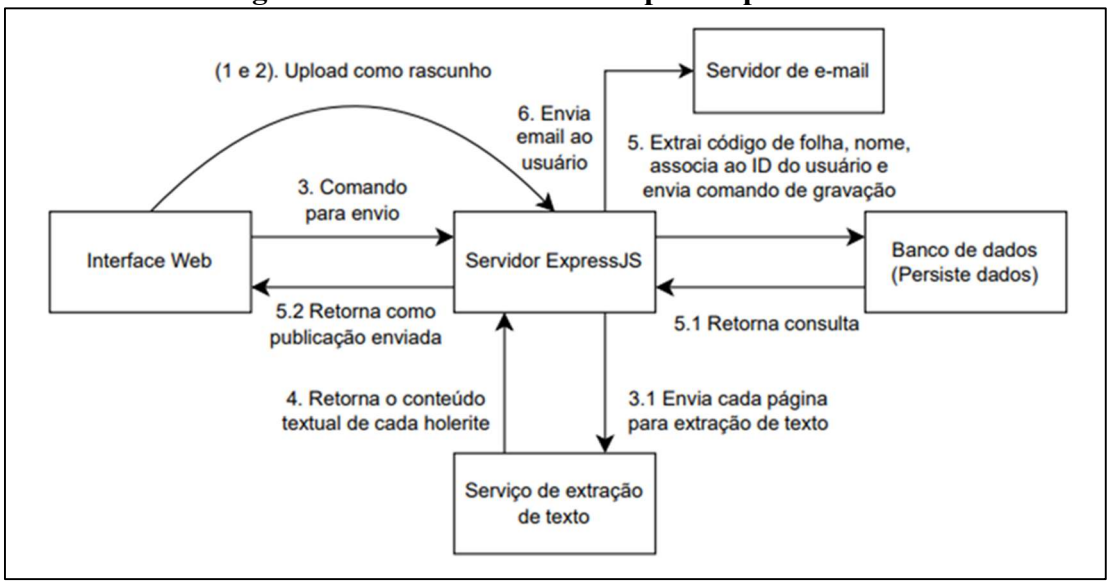
A partir do exposto, é possível compreender qualquer fluxo de eventos dentro da aplicação. A título de ilustração, focaremos no principal processo dentro do aplicativo: o carregamento de um arquivo contendo vários holerites e a distribuição de cada folha separada para o seu devido proprietário (funcionário). As etapas desse processo estão descritas abaixo e um esquema que os ilustra pode ser visualizado na Figura 3. Passos intermediários são indicados como subtópicos (3.1, 3.2, etc).

- 1. O analista de recursos humanos faz o upload de um arquivo PDF contendo os holerites. Essa operação de inserir o arquivo é efetuada por meio de uma chamada a uma rota do *back-end* executado no servidor;
- 2. O servidor em ExpressJS recebe o documento e o guarda como rascunho ainda não publicado (ou seja, ele ainda não foi separado em vários documentos para envio);
- 3. Por meio de um clique do usuário, a rota de separação de arquivos é acionada no *back-end*, o qual atua separando o documento PDF de origem em vários holerites em PDF individuais.
- 4. O serviço de separação aciona um script em Python que é executado dentro de um servidor Flask. Esse *script* captura cada holerite individual, extrai o texto utilizando uma biblioteca de OCR (*Optical Character Recognition*) denominada *pytesseract* e envia o conteúdo textual para o servidor ExpressJS.
- 5. O servidor recebe o texto e captura o código de folha e nome do funcionário, e, junto ao ID universal deste, associa o holerite ao colaborador e grava na base de dados. Em casos em que não há o identificador de folha no banco, um pré-cadastro do usuário é criado e a folha salarial é associada a ele. Os holerites gravados com seus

respectivos donos são publicados para visualização. Cada documento é armazenado em formato criptografado.

- 6. Uma notificação via e-mail é enviada aos usuários.

Figura 3 – Processo executado para separar os holerites



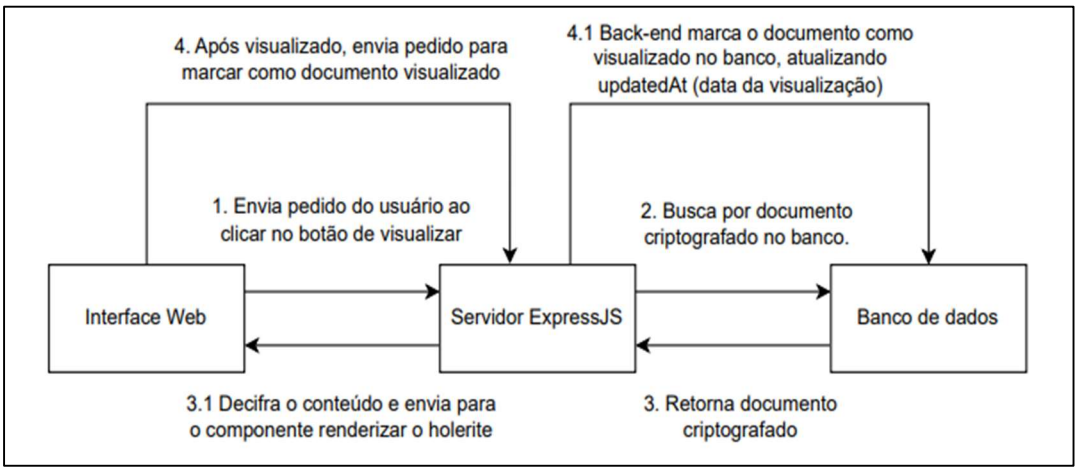
Fonte: Autor, 2025.

A partir dessa explicação, é possível verificar a presença das três camadas em todo o processo. A camada de apresentação renderiza os componentes e captura os eventos (cliques) de upload e de envio realizados pelo analista. A camada de aplicação realiza todo o processamento relativo a separar os holerites, extrair o texto de cada um, associar a folha ao funcionário correto e acionar os serviços de gravação no banco. O banco de dados, por sua vez, persiste as informações processadas em cada etapa.

Um outro procedimento na aplicação que exemplifica o funcionamento da arquitetura é o processo de um usuário visualizar seu holerite. Os passos estão descritos abaixo e um esquema que os ilustra pode ser visualizado na Figura 4. Passos intermediários são indicados como subtópicos.

- 1. O funcionário acessa o componente que renderiza os seus holerites e seleciona um documento por meio de um clique em um símbolo de "olho aberto";
- 2. Uma chamada é feita a uma rota do *back-end* que captura o identificador do documento que se deseja visualizar e decifra-o de forma que o arquivo criptografado se torne visualizável;
- 3. O retorno contendo o *blob* binário é enviado para o usuário e renderizado pelo componente *React*;
- 4. De forma paralela, assim que o usuário clica no símbolo de visualização, uma outra rota de API é acionada. Essa rota troca a coluna “isReceived” associado ao documento dentro da base de dados, além de atualizar o campo de “updatedAt”. Isso permite identificar se o holerite foi visualizado e, se sim, em qual data e hora.

Figura 4 – Processo para o usuário visualizar o holerite



Fonte: Autor, 2025.

Novamente, a partir deste processo apresentado acima, é possível verificar o papel de cada camada para a execução correta da funcionalidade. A camada de apresentação renderiza o botão de visualizar e envia a chamada para a API da camada de aplicação. Esta última, por sua vez, busca o arquivo na camada de persistência (banco de dados), decifra o documento criptografado e envia o conteúdo para a camada de aplicação renderizar para o usuário. Além disso, ela grava se o holerite foi visualizado e em que data isso ocorreu a partir de outra interação com o banco.

3.4 Desenvolvimento

O processo de desenvolvimento da aplicação foi iniciado a partir dos requisitos funcionais e não funcionais levantados durante a interação inicial com os *stakeholders*. Após isso, os requisitos foram segmentados em funcionalidades que, juntas, cumprem a exigência feita pelo(s) requisito(s) correspondente(s). A seguir, os tópicos relativos à construção do *back-end* e do *front-end* serão abordados.

3.4.1 *Back-end*

O desenvolvimento do *back-end* foi realizado de forma gradual, sendo que as funcionalidades basilares do sistema, tais como autenticação e operações de manipulação das entidades básicas (*User, Department* e *Role*), foram priorizadas em um primeiro momento. Após isso, os requisitos funcionais foram classificados conforme o grau de importância para o problema de negócio e, assim, seguiu-se a implementação de cada um. A organização dos diretórios do projeto foi definida de forma que refletisse a arquitetura escolhida. Nesse sentido, o projeto está estruturado da seguinte maneira:

- *Originalpdfs, payslips e photos*: pastas destinadas ao armazenamento de arquivos binários, sendo eles os relatórios de origem dos holerites, as folhas de pagamento em si e as imagens de perfil dos usuários, respectivamente;
- *Prisma*: este diretório contém o esquema de dados do projeto, além de uma subpasta que contém todos os arquivos de migração realizadas no banco de dados;
- *Config*: possui o arquivo de configuração do utilitário multer, uma ferramenta que auxilia o upload dos arquivos em PDF;

• *Controllers*: contém todas as classes encarregadas de recepcionarem as requisições HTTP e acionar os devidos serviços para respondê-las;
• *Middlewares*: contém os arquivos que interceptam as requisições HTTP antes de estas chegarem aos controllers. Utilizadas para verificar se o usuário está autenticado e autorizado a acessar o recurso;
• *Prisma*: contém o arquivo de configuração que instancia o utilitário prisma para uso interno na aplicação;
• *Services*: armazena as classes que lidam diretamente com a manipulação do banco de dados por meio do uso do utilitário do prisma.
• *Utils*: contém arquivos úteis para integração com bibliotecas de envio de e-mails, além de erros personalizados;
• *Arquivo "routes"*: arquivo que contém todas as rotas da aplicação;
• *Arquivo "server"*: arquivo que efetivamente instancia o servidor ExpressJS, além de servir para configurar o tratamento global de erros, a política de *Cross Origin*, a fonte das variáveis ambiente, os diretórios de arquivos binários e a fonte de rotas da aplicação.

Todo o aplicativo foi construído a partir do conceito de que uma rota está conectada a um controller, o qual, por sua vez, chama um serviço que faz a operação necessária na camada de persistência para responder corretamente à requisição feita pela camada de apresentação. Como exemplo, pode-se discorrer sobre uma rota de listagem de setores (*GET '/departments'*). A rota, ao ser chamada, invoca o método "*handle*" de um controller chamado "*ReadAllDepartmentsController*" o qual consiste em uma classe que tratará a requisição realizada.

Após isso, o controller instancia o serviço "*ReadAllDepartmentsService*" e invoca o método "*execute*" deste. Esse método acionará o utilitário Prisma para fazer a busca na base de informações. A resposta deste método – que contém os dados dos setores – é armazenada em uma variável que, posteriormente, é retornada dentro de um corpo em JSON para a camada solicitante. A classe do serviço atua, em praticamente todas as vezes, invocando o utilitário do ORM Prisma para fazer a busca (ou qualquer que seja a operação desejada) na base de dados.

Todos as funcionalidades derivadas dos requisitos foram implementadas, uma a uma, por meio desta abordagem. Dessa forma, foi possível separar a lógica responsável pelas requisições das regras de processamento relacionadas com a manipulação do banco de dados.

3.4.2 *Front-end*

A construção do *front-end* foi feita gradualmente, partindo das telas essenciais de cadastro e login dos usuários e posteriormente avançando para as páginas relacionadas ao problema de negócio. Tal como citado, essa camada do sistema foi construída utilizando o *framework* NextJS, o qual influenciou a organização dos arquivos.

Portanto, o projeto está organizado com a seguinte estrutura:
• Diretório "*.next*": contém os arquivos do *framework* NextJS;
• *Public*: contém os arquivos estáticos da aplicação, como logotipos;
• *Styles*: contém os estilos de CSS globais da aplicação;
• Components: possui os componentes *React* que são reutilizados em diferentes telas do aplicativo;

• *Contexts*: armazena os arquivos de contexto do *front-end*, os quais possuem funções com chamadas ao *back-end* que podem ser invocadas em qualquer componente do projeto;

• *Pages*: são os componentes reativos que constroem as diferentes telas que o usuário acessa ao utilizar o *software*;

• Arquivo "*app*": permite injetar o CSS global e configurar os providers que iniciam os diferentes contextos do sistema;

• Arquivo "*document*": utilizado para customizar os atributos de *head* e *body* do HTML inicial que é retornado no primeiro carregamento via renderização do lado do servidor (SSR);

• *Index*: é o componente que renderiza a primeira página a ser acessada. No caso, a página de login;

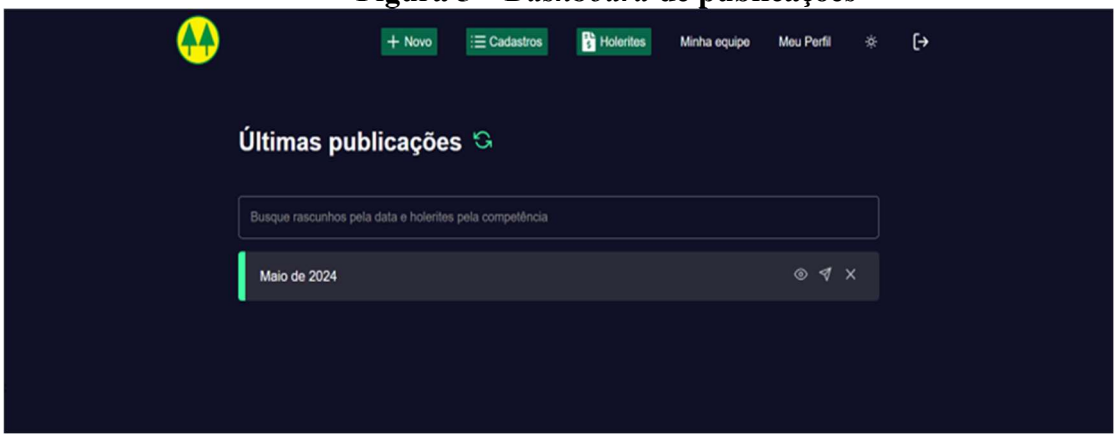
• *Utils*: contém utilitários de controle de renderização das telas. A partir deles, é possível bloquear, no nível da camada de aplicação, quais telas um usuário pode acessar a depender da sua permissão;

• Services: contém os arquivos que configuram utilitários para chamar a API do *back-end*.

A lógica fundamental do front-end neste projeto se baseia em páginas que são componentes *React*, cada uma com sua própria estrutura e estilo CSS. Esses componentes encapsulam sua própria lógica, que pode envolver chamadas a funções internas ou interações com funções presentes nos diferentes contextos do NextJS via Context API, uma funcionalidade do *React*.

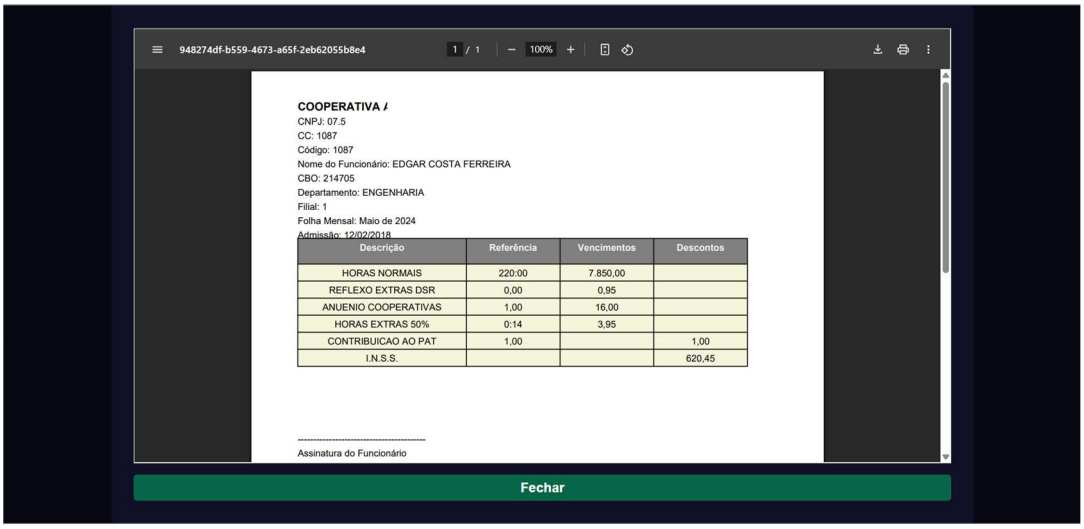
Dessa forma, é possível ter um programa cujas telas são reativas e fazem diferentes chamadas a funções de acordo com os eventos disparados pelo usuário. As Figuras 5 e 6 ilustram as páginas de dashboard de publicações cadastradas pelo analista de recursos humanos e a tela de visualização de um holerite que é mostrada ao usuário, respectivamente. O arquivo exposto é apenas um PDF falso produzido com o mesmo padrão de texto extraído dos holerites que seguem o *layout* emitido pelo sistema contábil.

Figura 5 – D*ashboard* de publicações



Fonte: Autor, 2025.

Figura 6 – Modal de visualização do holerite selecionado



Fonte: Autor, 2025.

3.5 Testes

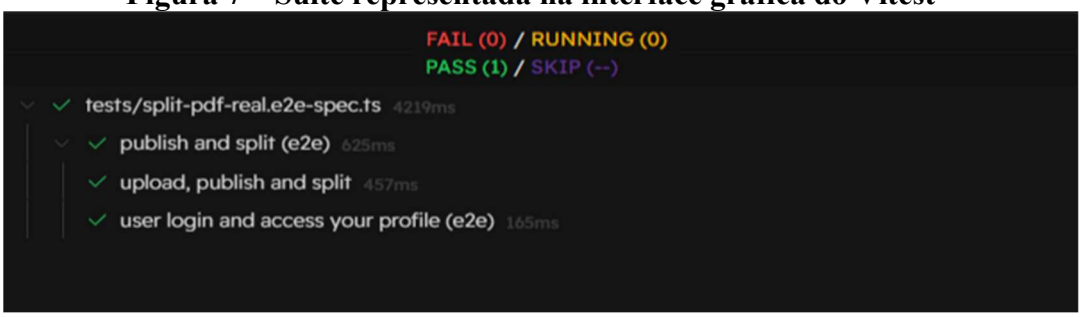
Os testes foram desenvolvidos com foco em uma abordagem denominada "ponta a ponta" (E2E), utilizando a ferramenta Vitest em conjunto com a biblioteca *Supertest* para simular requisições HTTP contra a aplicação. Os testes são executados em um ambiente separado a partir da construção de um *schema* de dados dinâmico a cada execução, o que garante isolamento e confiabilidade de resultados.

O programa Flask de extração de texto foi empacotado com *Docker* de modo que seja executado em paralelo e monitorado com a biblioteca *wait-on*, de maneira a garantir que esteja disponível antes da execução da suíte. A suíte de testes abrange dois focos principais: a funcionalidade crucial da aplicação (receber um arquivo, executar a separação deste e armazenar holerites) e o acesso autenticado do usuário à plataforma.

A partir do primeiro objetivo, construiu-se um teste automatizado que reproduz a criação da publicação, a execução do serviço de separação e criptografia, o armazenamento desses dados no formado desejado e a listagem das informações. Acerca do fluxo de teste da autenticação, a suíte testa o login de usuário e o acesso deste às suas informações de perfil.

Com isso, foi possível criar uma esteira de testes que cobre a principal função de interesse do sistema de forma confiável e realista, validando as etapas críticas do fluxo de publicação e acesso aos documentos. A suíte de testes pode ser visualizada na Figura 7.

Figura 7 – Suíte representada na interface gráfica do Vitest



Fonte: Autor, 2025.

4 RESULTADOS E DISCUSSÕES

A execução da suíte de testes apresentou resultados satisfatórios. Os cenários previstos foram validados com sucesso, desde o upload e criação da publicação até a validação do armazenamento dos arquivos em formato cifrado e acesso autenticado do usuário. A esteira de testes demonstrou robustez ao isolar o ambiente a cada execução por meio do uso de *schemas* dinâmicos do banco de dados.

A integração com o serviço externo, empacotado em container *Docker* e monitorado durante os testes, mostrou-se eficaz. O uso de *mocks* permitiu controlar a autenticação e validar os fluxos esperados de autorização e segurança da aplicação.

Apesar da cobertura eficiente dos fluxos principais, identificou-se que há a limitação representada pela ausência de testes unitários que cubram módulos menores da aplicação. A abordagem atual foca em validar o comportamento global da aplicação, carecendo de testes específicos para funções da camada de negócio. Dessa forma, conclui-se que a introdução de testes unitários pode contribuir para maior granularidade na detecção de falhas e reforço da confiabilidade a longo prazo, ficando no domínio de trabalhos futuros.

5 CONCLUSÕES

O trabalho desenvolvido atingiu o objetivo principal pelo qual foi concebido: construir um sistema capaz de automatizar o processo de separação, criptografia e disponibilização de holerites de forma segura e com acesso controlado. A arquitetura projetada, com integração de um *back-end* em NodeJS, banco de dados PostgreSQL e serviço externo em Flask para extração de dados se demonstrou eficiente para resolver o problema apresentado.

A implementação de uma suíte de testes ponta a ponta contribuiu para validar os fluxos principais do sistema de forma isolada e confiável. O uso de conteinerização e schemas dinâmicos reforçou a robustez do ambiente de testes.

Como limitação, destaca-se a ausência de testes unitários. Apesar disso, a solução entregue atende ao escopo proposto em quesitos técnicos e oferece uma base estruturada para futuras extensões, como dashboards, relatórios e auditoria de acessos.

REFERÊNCIAS

DIMEP. **Controle de Ponto**. Disponível em: https://www.dimep.com.br/solucoes/controle-de-ponto. Acesso em: 30 mar. 2025.

DOCKER. **Docker – platform designed to help developers build, share, and run container applications**. Disponível em: https://www.docker.com/. Acesso em: 28 mai. 2025.

EXPRESSJS. **Express – Fast, unopinionated, minimalist web framework for Node.js**. Disponível em: https://expressjs.com/. Acesso em: 30 mar. 2025.

FLASK. **Flask Documentation**. Disponível em: https://flask.palletsprojects.com/en/stable/. Acesso em: 30 mar. 2025.

JWT. **Introduction to JSON Web Tokens**. Disponível em: https://jwt.io/introduction. Acesso em: 30 mar. 2025.

MORAIS, Izabelly S.; ZANIN, Aline. **Engenharia de software**. 1. ed. Porto Alegre: SAGAH, 2020.

NEXT.JS. **Next.js Documentation**. Disponível em: https://nextjs.org/docs. Acesso em: 30 mar. 2025.

OLIVEIRA, Luana Y. M.; OLIVEIRA, Pablo R. B.; SAWITZKI, Roberta; et al. **Gestão de pessoas**. 1. ed. Porto Alegre: SAGAH, 2018.

POSTGRESQL. **PostgreSQL: The World's Most Advanced Open Source Relational Database**. Disponível em: https://www.postgresql.org/ . Acesso em: 2 set. 2024.

PRISMA. **Prisma**. Disponível em: https://www.prisma.io/. Acesso em: 30 mar. 2025.

TECSMART. EPM – **Solução completa de gestão de RH**. Disponível em: https://www.tecsmart.com.br/epm/ . Acesso em: 30 mar. 2025.

TYPESCRIPT. **TypeScript: TypeScript for the New Programmer**. Disponível em: https://www.typescriptlang.org/pt/docs/handbook/typescript-from-scratch.html. Acesso em: 30 mar. 2025.

VALENTE, M. T. **Engenharia de software moderna:** princípios e práticas para Desenvolvimento de Software com Produtividade. [S.l.]: Independente, 2020.

VITEST. **Vitest – next generation testing framework powered by Vite**. Disponível em: https://vitest.dev/. Acesso em: 28 mai. 2025.