ANÁLISE COMPARATIVA ENTRE LINGUAGENS DE PROGRAMAÇÃO WEB AO CONSUMIR APIS

Daniel Junkes Mafioleti[1], André Faria Ruaro[2]

Resumo: Consumir APIs é uma atividade comum no desenvolvimento *frontend*. Para facilitar esse processo, surgiu a biblioteca HTMX, que permite fazer requisições diretamente por meio de atributos HTML, eliminando a necessidade de escrever código JavaScript. Este trabalho desenvolveu dois sistemas que consomem uma API feita em Django: um utilizando React no frontend e outro usando apenas Django com HTMX. Ambos implementam um CRUD simples, com as funcionalidades de adicionar, editar e remover itens de uma lista. O objetivo foi avaliar a funcionalidade e a viabilidade do HTMX no consumo de APIs. Os resultados mostraram que o HTMX facilita o consumo da API no frontend, tornando o processo mais direto e com menos código. No entanto, isso desloca a responsabilidade da lógica para o *backend*, exigindo que a API retorne componentes HTML renderizados em vez de apenas dados em JSON. Com isso, o uso do HTMX pode tornar o *backend* mais complexo e exigir uma organização mais detalhada no retorno das requisições. Ainda assim, a biblioteca se mostrou eficiente quando se deseja um desenvolvimento mais rápido e simples no *frontend*, sem depender de *frameworks* mais robustos como o React.

Palavras-chave: Python; Django; Javascript; *web*; API; Comparação, htmx.

[1]Curso de Ciência da Computação, UniversIDade do Extremo Sul Catarinense (Unesc), danielmafioleti2@unesc.net

[2]Curso de Ciência da Computação, UniversIDade do Extremo Sul Catarinense (Unesc), andre.ruaro@unesc.net

ABSTRACT: Consuming APIs is a common activity in frontend development. To facilitate this process, the HTMX library was created, which allows requests to be made directly through HTML attributes, eliminating the need to write JavaScript code. This work developed two systems that consume an API made in Django: one using React on the frontend and the other using only Django with HTMX. Both implement a simple CRUD, with the functionalities of adding, editing and removing items from a list. The objective was to evaluate the functionality and feasibility of HTMX in consuming APIs. The results showed that HTMX facilitates the consumption of the API on the frontend, making the process more direct and with less code. However, this shifts the responsibility of the logic to the *backend*, requiring the API to return rendered HTML components instead of just JSON data. Therefore, the use of HTMX can make the backend more complex and require a more detailed organization in the return of requests. Still, the library has proven to be efficient when you want faster and simpler development on the frontend, without depending on more robust frameworks like React.

Keywords: Python; Django; Javascript; *web*; API; Comparação, htmx.

1 INTRODUÇÃO

No presente cenário, com a tecnologia da informação sendo usada em diferentes contextos, surgiu a necessidade de ter serviços online e informatizados, com isso nasceram muitas linguagens de programação e elas foram sendo usadas nos mais diversos contextos. Mais especificamente no contexto da Internet surgiram *Application Programming Interfaces* (API), elas são serviços que recebem ou enviam informações para um cliente (Maior, 2023).

Para realizar a criação de APIs é necessário o uso de *frameworks*. Segundo Farias (2022), *frameworks* são estruturas usadas para a criação de aplicações que tem consigo soluções completas para os desenvolvedores, tendo funções pré-definidas que ajudam e aceleram o desenvolvimento de novos projetos. A maioria das linguagens conhecidas possuem *frameworks* para a criação de APIs, por exemplo, em Javascript temos o Express, em Java temos o Spring Boot e em Python temos o Django.

Javascript é uma linguagem de programação interpretada sendo popularmente usada em navegadores. Segundo Stefanov (2011) o Javascript começou sendo uma forma de manipular alguns elementos do HTML (*HyperText Markup Language*), porém cresceu rapidamente e atualmente além do *client-side*, é usada no *server-side* com Node.js ou .NET.

Uma das possíveis formas de consumir APIs é a criação de requisições AJAX (*Asynchronous JavaScript And XML*) que pode ser feita com o uso da interface fetch já disponível no Javascript ou com o uso de bibliotecas externas, como por exemplo, a biblioteca htmx.

Hmtx funciona como uma extensão do HTML, criado em 2020 por Carlos Gross. Ele permite realizar requisições AJAX sem a necessidade de JavaScript mas para isso é necessário armazenar mais HTML no servidor do que as estruturas convencionais. A resposta pode ser inserida em qualquer lugar do DOM (*Document Object Model*), sendo possível substituir outro elemento já presente na página (Paakkanen, 2023). Além disso, segundo o site htmx (2020b), com ele é possível reduzir em até 67% o tamanho do código comparado com o *framework* React além de deixar a implementação com APIs mais simples.

O levantamento bibliográfico sobre o uso do htmx trouxe poucos resultados. Kaurinović (2022) desenvolvou um sistema que permite a criação e gerenciamento de perfil e a criação e interação com posts utilizando Django e htmx, Kaurinovic conclui que combinar o *framework* Django com a biblioteca htmx permite a criação de aplicações *web* interativas sem precisar utilizar *frameworks* de *Single Page Application* (SPA) como por exemplo o React, Angular ou Vue e reduz o tempo de desenvolvimento ao permitir que de forma eficaz as aplicações permitem o usuário salvar, editar e excluir dados.

(Vieira, 2024) desenvolveu um sistema utilizando htmx com o objetivo digitalizar processos manuais, melhorar a eficiência da empresa e a comunicação interna em uma microempresa do setor da moda. Vieira conclui que o uso da biblioteca htmx possibilitou uma interface mais interativa contribuindo para a eficiência e usabilidade do sistema.

Ferreira e Zuchi (2018) realizaram uma analise sobre os *frameworks frontend* para aplicações *web* baseado em Javascript, foram analisados os *frameworks* Angular, Vue e React. Foram feitas pequenas aplicações semelhantes em cada *framework* na versão *Command Line Interface* (CLI).

Cada aplicação possui um formulário com 2 campos, um para nome e outro para *email*, e uma lista com 100 registros sendo carregados a partir de um arquivo JSON, nessa lista é possível editar e remover resgistros. Foi concluido que o React possuí o menor tamanho, o Vue possui a melhor velocidade de renderização e o Angular foi o que menos se destacou.

Com isso, esta pesquisa se propõe a realizar uma comparação entre o uso de htmx em conjunto com Django e as formas tradicionais de consumo de APIs. Será analisado a facilidade de implementação e viabilidade do uso do htmx.

Como a biblioteca htmx é relativamente nova, com sua primeira versão sendo lançada em 2020 conforme o site oficial htmx (2020b), não existem muitos artigos científicos comentando sobre ela, possuindo mais informações em vídeos e *web* sites, porém não possuem validade científica dos fatos. Com a pouca disponibilidade de informações cientificamente comprovadas, se faz necessário um estudo científico sobre o uso da biblioteca htmx. Com um estudo científico é possível verificar as diferenças existentes entre o uso dos métodos tradicionais e o uso do htmx ao consumir APIs.

A realização de uma comparação traz benefícios pois a partir dela podemos verificar qual a tecnologia que atende melhor nossas necessidades além de verificar a eficiência das tecnologias envolvidas em diferentes áreas. Segundo Coelho (2022), realizar comparações é bastante necessário para a criação de novas ideias e conceitos.

De acordo com a pesquisa Stack-Overflow-Survey (2023) a linguagem mais usada pelos desenvolvedores é o Javascript seguido pelo Python, o que nos mostra a força do Javascript no desenvolvimento de aplicações. Os objetivos específicos deste trabalho consistem em: aperfeiçoar o conhecimento nas linguagens de programação Javascript e Python como também nos *frameworks* React e Django, compreender a biblioteca htmx e utiliza-la na criação de APIs, desenvolver duas aplicações, realizar uma comparação entre as aplicações desenvolvidas e analisar os resultados obtidos.

2 MATERIAIS E MÉTODOS

O desenvolvimento de uma análise comparativa entre linguagens de programação *web* no consumo de APIs tem como objetivo avaliar duas abordagens distintas: o uso da biblioteca htmx em comparação com o método tradicional de consumo de APIs. A proposta é identificar as vantagens, limitações e impactos de cada abordagem na construção de aplicações *web*.

Para a realização da análise, foi desenvolvido um sistema que permite ao usuário adicionar, visualizar, editar e remover itens de uma lista. Esse sistema foi implementado em duas versões distintas: uma utilizando

React e outra com Django e a biblioteca htmx. A API, desenvolvida em Django, também possui duas versões: uma tradicional, consumida pelo *frontend* em React, e outra integrada diretamente com o Django por meio do htmx.

Foi necessário modificar a estrutura da API para que ela possa ser consumida pela biblioteca htmx, uma vez que o htmx requer que as respostas das requisições contenham diretamente trechos de HTML. Normalmente, APIs retornam dados no formato JSON, o que não é compatível com a forma como o htmx funciona. Portanto, a API original foi ajustada para retornar HTML, garantindo sua compatibilidade com o funcionamento do htmx.

A metodologia foi estruturada em quatro etapas: aprofundamento do conhecimento sobre Django e React; desenvolvimento dos sistemas e APIs; realização da comparação entre as abordagens; e análise dos resultados, com o objetivo de avaliar a utilização da biblioteca htmx.

Para aprofundar o conhecimento em Django e React, foi realizada uma pesquisa bibliográfica que resultou na leituras de artigos, *blogs* e vídeos com o intuito de melhorar a compreensão sobre os respectivos *frameworks*.

2.1 HTMX

Htmx é uma biblioteca escrita em JavaScript criada por Carlos Gross e lançada em 2020, ela permite realizar requisições HTTP diretamente no HTML sem a necessidade de escrever códigos JavaScripts complexos (Kaurinović, 2022).

Como o htmx é uma biblioteca livre de dependências, para utilizá-la é possível apenas adicionar uma *TAG* `<Script>` no ínicio da página HTML fazendo sua importação *online* ou simplesmente a baixando e copiando seu caminho, outra forma de utilizar o htmx é fazendo sua instalação pelo NPM do Node.js (htmx, 2020a).

Ao utilizar a biblioteca htmx é necessária uma mudança na funcionalidade da API pois a sua resposta deve ser o HTML, fazendo com que o servidor tenha que armazenar mais HTML. Essa mudança simplifica o desenvolvimento *front-end* e reduz a quantidade de JavaScript no lado do cliente, melhorando o desempenho principalmente em dispositivos com recursos limitados (Paakkanen, 2023).

Realizar requisições AJAX diretamente no HTML é o principal da biblioteca htmx e para isso ele possui atributos que são colocados dentro

de uma *TAG* HTML. Para realizar uma solicitação GET, POST, PUT ou DELETE com o htmx é necessário apenas colocar os atributos `hx-get='URL'`, `hx-post='URL'`, `hx-put='URL'` e `hx-delete='URL'` respectivamente, sendo "URL"o *endpoint* da API (htmx, 2020a).

Por padrão, as respostas das requisições são colocadas dentro do elemento que a chamou, mas é possível mudar esse comportamento através do atributo `hx-target="<CSS selector>"`, esse atributo recebe um seletor de CSS. Além disso também é possível definir como a resposta será colocada com o atributo `hx-swap=""` (htmx, 2020a).

Com essas funcionalidades, o htmx permite criar páginas dinamicas e interativas sem a necessidade de utilizar um *framework* específico para isso. A pesquisa Stack-Overflow-Survey (2024) mostra que 3,3% dos programadores que responderam a pesquisa utilizam o htmx em seus projetos, um grande aumento comparado com a pesquisa do ano de 2024 em que o htmx nem aparecia entre os *framework* e tecnologias *web* mais utilizadas pelos programadores.

2.2 SISTEMA REACT

Para o desenvolvimento do sistema que consome a API de forma convencional utilizando JavaScript, o *backend* foi implementado com Django, enquanto o *frontend* foi desenvolvido utilizando o *framework* React.

2.2.1 Criação da API

Para a criação da API, foi utilizado o *framework* Django. O primeiro passo consistiu na instalação do Django e após instalá-lo, foi iniciado o projeto com o nome 'core'. Essa inicialização gerou uma pasta com a estrutura básica do projeto, contendo os arquivos de configuração e os *endpoints* iniciais da API.

Para iniciar a criação de um novo endpoint, foi criado um aplicativo com o nome 'api'. Esse aplicativo gerou uma nova pasta com o nome do aplicativo, onde foram desenvolvidos os *endpoints* necessários.

Para que o Django reconheça e utilize o aplicativo criado, foi necessário editar o arquivo 'settings.py', localizado na pasta principal do projeto. Dentro dele, foi adicionada a referência ao novo aplicativo, permitindo sua integração com o restante do sistema.

Foi necessário criar um arquivo chamado 'urls.py' dentro da pasta do aplicativo. Esse arquivo é responsável por definir os *endpoints* específicos daquele módulo. Em seguida, foi realizado o ajuste no arquivo 'urls.py'

da pasta principal do projeto, onde foi adicionada uma referência ao arquivo criado. Dessa forma, os *endpoints* definidos no aplicativo passaram a ser acessíveis a partir das rotas configuradas no projeto.

No arquivo 'models.py', foi criado o modelo que representa os dados utilizados pela API. Para isso, foi definida uma classe que herda de `models.Model`, permitindo o mapeamento do modelo para o banco de dados por meio do ORM do Django. Essa classe contém dois campos: item, definido como um `CharField` para armazenar textos curtos, e status, definido como um `BooleanField` para representar um valor booleano (verdadeiro ou falso). Esse modelo serve como base para as operações de criação, leitura, atualização e exclusão (CRUD) realizadas pela API.

Após a criação do modelo, foi necessário aplicar as alterações ao banco de dados. Para isso, utilizaram-se os comandos `python.exe .\manage.py makemigrations` e, em seguida, `python.exe .\manage.py migrate`. O primeiro comando gera os arquivos de migração com base na estrutura definida na classe do modelo, enquanto o segundo executa essas migrações, atualizando o banco de dados conforme as novas definições.

No arquivo 'views.py', foram criadas as funções responsáveis pelo processamento das requisições da API. Para utilizar o modelo definido anteriormente, foi necessário importar a classe correspondente. Cada função recebe um parâmetro request, que contém os dados enviados pelo *frontend*. No caso das funções que recebem um corpo de requisição (*payload*), foi necessário converter o conteúdo de `request.body` para um objeto JSON, a fim de extrair os dados necessários para o processamento.

A função create-item recebe os dados vindo da requisição POST, converte em JSON, cria um item, adiciona no banco de dados e retorna o item com o ID conforme a Figura 1.

Figura 1- Função: create-item.

```python
from django.http import JsonResponse
from API.models import Item
import json

def create_item(request):
    dados = json.loads(request.body)
    item = dados.get('item')
    status = dados.get('status')

    item = Item(item=item, status=status)
    item.save()

    return JsonResponse({'message':
    'Item criado com sucesso!',
    'item': {'id': item.id, 'item': item.item,
    'status': item.status}})
```

Fonte: Elaborado pelo autor.

A função `get_itens` recebe uma requisição do método GET e retorna todos os itens cadastrados no banco de dados a Figura 2.

Figura 2- Função: get_itens.

```python
def get_itens(request):

    itens = Item.objects.all().values()

    return JsonResponse({'itens': list(itens)})
```

Fonte: Elaborado pelo autor.

A função `update_item` recebe uma requisição do método PUT, busca no banco de dados o item filtrando por ID e atualiza os valores com os dados vindo da requisição conforme a Figura 3.

Figura 3- Função: update_item.

```
def update_item(request, id):
    try:
        item = Item.objects.get(id = id)
        dados = json.loads(request.body)

        item.item = dados.get('item')
        item.status = dados.get('status')

        item.save()
        return JsonResponse({'message':
            'Item atualizado com sucesso!'})
    except Item.DoesNotExist:
        return JsonResponse({'error':
            'Item não encontrado'}, status=404)
```

Fonte: Elaborado pelo autor.

A função `delete_item` recebe uma requisição do método DELETE e remove o item conforme o ID recebido na requisição conforme a Figura 4.

Figura 4- Função: delete_item.

```
def delete_item(request, id):
    try:
        item = Item.objects.get(id = id)
        item.delete()
        return JsonResponse({'message':
            'Item deletado com sucesso!'})
    except Item.DoesNotExist:
        return JsonResponse({'error':
            'Item não encontrado'}, status=404)
```

Fonte: Elaborado pelo autor.

No arquivo ’urls.py’ do aplicativo, foi criada uma lista, na qual foram definidos os *paths* responsáveis por criar os *endpoints* do aplica-

tivo. Em cada *path*, foi referenciada a função correspondente do arquivo 'views.py', permitindo que o sistema identifique qual função deve ser executada ao receber uma requisição naquele *endpoint* conforme a Figura 5.

Figura 5- *endpoints* do aplicativo.

```python
from django.urls import path
from . import views

urlpatterns = [
    path('get_itens/', views.get_itens, name='get_itens'),
    path('create_item/', views.create_item, name='create_item'),
    path('update_item/<str:id>/', views.update_item, name='update_item'),
    path('delete_item/<str:id>/', views.delete_item, name='delete_item'),
]
```

Fonte: Elaborado pelo autor.

Para permitir a comunicação entre o *frontend* e o *backend*, foi necessário configurar o CORS (Cross-Origin Resource Sharing), uma função que controla o acesso a recursos de um site por páginas web hospedadas em domínios diferentes, permitindo o compartilhamento de recursos entre diferentes domínios na web de forma segura.

Primeiramente, a biblioteca 'django-cors-headers' foi instalada. Após a instalação, o arquivo 'settings.py' foi acessado para realizar as configurações necessárias. Na lista INSTALLED_APPS, foi adicionado o aplicativo 'corsheaders'. Ainda no mesmo arquivo, foram inseridas as seguintes configurações: CORS_ALLOW_ALL_ORIGINS = True, CORS_ALLOW_CREDENTIALS = True, CORS_ALLOWED_ORIGINS = [], CSRF_TRUESTED_ORIGINS = [], além disso, dentro das listas CORS_ALLOWED_ORIGINS e CSRF_TRUESTED_ORIGINS foram incluídos os endereços IP autorizados a realizar requisições ao servidor, garantindo a segurança das informações. Dessa forma a API foi finalizada e ficou pronta para ser utilizada.

2.2.2 Criação do *frontend*

Para o desenvolvimento do *frontend*, foi utilizado o React. O projeto React foi criado por meio do comando: `npx create-react-app meu-aplicativo` onde meu-aplicativo é o nome do projeto. Embora o create-react-app esteja considerado obsoleto, ele atendeu às necessidades deste projeto. Para executar a aplicação, foi utilizado o comando: `npm start` dentro da pasta do projeto React, iniciando o servidor de desenvolvimento.

O arquivo 'App.js' foi modificado para consumir os *endpoints* de

busca e criação da API. Para a criação de um novo item, foi implementado um campo de entrada (*input*) do tipo texto, acompanhado de um botão que dispara a função responsável por enviar a requisição para a API.

Para consumir a API, foi utilizada a função `fetch`, que é padrão do JavaScript para realizar requisições HTTP. A função responsável por criar um item realiza uma requisição do tipo POST para o *endpoint* responsável pela criação do item, enviando no corpo da requisição o valor digitado no *input* no campo item e definindo o campo status como *false*. O servidor responde com o item criado, incluindo um ID, que é então adicionado à lista de itens exibida na interface.

Já a função que busca os itens é executada assim que a página carrega. Ela faz uma requisição GET para o endpoint de busca de itens e a resposta da API, que contém a lista de itens, é adicionada à lista exibida no *frontend*.

Foi criado um arquivo chamado 'item.js', responsável por interagir com os *endpoints* de atualização e exclusão de itens da API. O componente representa um item que exibe o nome recebido da API, uma *checkbox* que reflete o *status* do item, além de incluir um botão para editar e outro para deletar o item.

A função responsável por atualizar o item é acionada quando o *status* da *checkbox* é alterado ou quando o botão de salvar, durante a edição do nome, é clicado. Essa função realiza uma requisição do tipo PUT para o *endpoint* responsavel por atualizar o item, enviando o ID do item na URL da API para identificar qual registro deve ser atualizado.

Já a função para excluir o item é chamada ao pressionar o botão de deletar, realizando uma requisição DELETE para o *endpoint* responsavel por excluir o item, também com o ID do item informado na URL para especificar qual item deve ser removido.

2.3 SISTEMA DJANGO + HTMX

Para a criação de um novo aplicativo em Django integrado com htmx, foi reutilizada a estrutura da API anterior. Foi gerado um novo aplicativo e configurado conforme mencionado anteriormente.

No arquivo 'urls.py', foi criada uma nova lista, contendo os *endpoints* que serão utilizados especificamente pelo htmx. Após a definição dessa lista, ela foi adicioanada na lista padrão de *endpoints* para que o Django os utilize.

Além disso, foi criado um arquivo chamado 'views_htmx.py' para

separar as *views* específicas do htmx das *views* convencionais do Django. Nesse arquivo, foram implementadas as funções que correspondem aos *endpoints* definidos na lista anterior.

Na estrutura do projeto, fora da pasta do aplicativo, foi adicionada uma pasta chamada '*templates*'. Dentro dela, foi criada uma subpasta chamada '*partials*', que por sua vez contém a pasta '*htmx_components*'.

Além disso, dentro da pasta do aplicativo, também foi criada uma pasta '*templates*' para que o Django possa localizar os *templates* específicos do aplicativo. Para garantir que o Django reconheça a pasta principal de *templates*, foi necessário editar o arquivo 'settings.py'. Na configuração TEMPLATES, dentro do campo DIRS, foi adicionada a seguinte linha: `[os.path.join(BASE_DIR, 'templates')]`

Na pasta '*templates*' do aplicativo, foi criado um arquivo HTML para ser o HTML principal da aplicação. Nesse arquivo foi importado o htmx conforme orientado no site oficial do htmx. Foi desenvolvido um formulário com um campo obrigatório e um botão do tipo *submit*. Dentro da TAG `<forms>`, foram utilizados os atributos do htmx. No atributo `hx-post` foi informado o *endpoint* da API que se deseja acessar, no atributo `hx-trigger` foi informado qual será o evento que irá desencadear a requisição e no atributo `hx-target` foi informado em qual local será colocado a resposta da requisição conforme a Figura 6. Além disso, foi criada uma `<div>` com um ID específico para receber e renderizar a resposta da requisição..

Figura 6- Forms para criação de itens.

```
<form
  hx-post="{% url 'create_item' %}"
  hx-trigger="submit"
  hx-target="#list-itens">
  <input type="text" name="item" class="form-control" required>
  <button clas="btn btn-sucess" type="submit">Salvar</button>
</form>
```

Fonte: Elaborado pelo autor.

No arquivo 'views.py', foi criada uma função que realiza uma consulta no banco de dados para buscar todos os itens, armazenando-os em uma variável. Em seguida, a função retorna o resultado da chamada com a função `render`, passando como argumentos o objeto *request*, o arquivo HTML localizado na pasta templates e um contexto contendo a variável que representa os itens recuperados.

Dentro da pasta 'htmx_components' foi criado um arquivo HTML

responsável por exibir os itens na tela principal. Nesse arquivo, foi utilizado um laço `FOR` para iterar sobre cada item presente na lista, renderizando o componente correspondente para cada um. Dentro do `FOR` foi criado uma TAG `<span>` responsável por mostrar o texto do item, uma *checkbox* responsável por mostrar o status, um botão de edição e um botão para excluir o item.

Na TAG da *checkbox* foram adicionados os atributos do htmx. No momento em que o valor da *checkbox* é alterado, é realizada uma requisição para o *endpoint* responsavel por alterar o *status* do item retornando a resposta da API na TAG com o ID 'list-itens'.

O botão de edição segue o mesmo padrão, porém utiliza o atributo do htmx para realizar uma requisição do tipo GET ao *endpoint* responsavel por retornar o componente de edição do item. O alvo da requisição é a própria TAG do item, permitindo que o formulário de edição seja carregado naquele local.

Já o botão de deletar realiza uma requisição para o *endpoint* responsável por excluir o item, passando o ID do item na URL, e tem como alvo a lista de itens, a `<div>` com ID list-itens, para que a lista seja atualizada após a exclusão.

Na pasta 'htmx_components' foi criado um componente HTML que contém uma TAG `<form>` para possibilitar a edição do nome do item, esse item é retornado na função de editar o item.

Esse `<form>` possui os atributos do htmx utilizados anteriormente e também o atributo `hx-swap="outerHTML"`, garantindo que a resposta substitua completamente o HTML do item editado.

Dentro do formulário, há um campo `<input>` do tipo texto que recebe o nome atual do item, e um botão do tipo *submit* para enviar a requisição. O alvo (`hx-target`) da requisição é a `<div>` com o ID list-itens, que contém a lista dos itens.

Além disso, foi adicionado um botão para cancelar a edição, que utiliza o atributo `hx-get` para realizar uma requisição ao *endpoint* responsável por buscar apenas um item, passando o ID do item na URL. O `hx-target` deste botão é a TAG com o ID dinâmico item-ID que varia conforme cada item vindo do banco de dados para restaurar a exibição original do item.

Para isso foi criado um arquivo HTML separado contendo o código para renderizar apenas um único item. Esse arquivo foi criado copiando o conteúdo interno do laço `FOR` presente no arquivo 'itens.html'.

No arquivo ’views_htmx.py’, foram criadas as funções responsáveis pelo processamento dos dados via htmx. A função de criar item é responsável por criar um novo item no banco de dados e, em seguida, retorna a lista atualizada de itens. A função de deletar o item segue a mesma lógica, mas remove o item especificado do banco de dados antes de retornar a lista atualizada de itens. A função de alterar o *status* altera o *status* do item (por exemplo, de ativo para inativo ou vice-versa). Todas essas funções retornam o mesmo componente, ’itens.html’.

Foi criada uma função responsavel por retornar o componente de edição de item onde ela realiza a busca do item pelo ID e renderiza o *template* criado para edição, passando o item que será editado como contexto. A função de atualizar o item recebe os dados da requisição para atualizar o item correspondente no banco de dados e retorna o componente ’itens.html’ com a lista atualizada de itens. Já a função para buscar apenas um item realiza a busca do item pelo ID no banco de dados e retorna o template ’item.html’, juntamente com os dados do item solicitado.

No arquivo ’urls.py’, na lista criada para os *endpoint* do htmx foi adicionado os *endpoints* juntamente com as suas respectivas funções criadas no arquivo ’htmx_views.py’ conforme a Figura 7.

Figura 7- Arquivo urls.py.

```python
from django.urls import path
from api_htmx import htmx_views
from . import views

urlpatterns = [
    path('', views.get_itens, name='list_itens'),
]

htmx_patterns = [
    path('create_item', htmx_views.create_item, name='create_item'),
    path('delete_item/<int:item_id>/', htmx_views.delete_item, name='delete_item'),
    path('toggle-status/<int:item_id>/', htmx_views.toggle_status, name='toggle_status'),
    path('edit-item-form/<int:item_id>/', htmx_views.edit_item_form, name='edit_item_form'),
    path('update-item/<int:item_id>/', htmx_views.update_item, name='update_item'),
    path('get-item/<int:item_id>/', htmx_views.get_item, name='get_item'),
]

urlpatterns += htmx_patterns
```

Fonte: Elaborado pelo autor.

3 DISCUSSÃO E RESULTADOS

Os resultados obtidos nesta pesquisa vieram de analises comparativas entre os dois modelos desde a sua criação até suas funcionalidades observando a facilidade de implementação e conhecimento necessário para sua execução.

O htmx facilitou as requisições à API por meio de seus atributos, mas tornou a criação da API mais complexa, exigindo maior compreensão da estrutura e lógica para replicar as funcionalidades já presentes no sistema React.

A curva de aprendizado do htmx é rápida para consumir a API, pois basta adicionar atributos nas TAGs HTML. Porém, criar a API para htmx é mais lento, pois exige programar partes separadas e integrá-las corretamente. Já no React, o aprendizado é mais linear, facilitado pela API que atua apenas no envio dos dados conforme os resultados apresentados na Tabela 1.

Tabela 1 - Resultados das analises

	Django + htmx	Django + React
Complexidade da API	Mais complexa.	Mais simples.
Formato da resposta	HTML	JSON
Consumo da API	Simples via atributos do htmx.	Requer fetch/axios, mais flexível.
Curva de aprendizado	Suave caso conheca o Django.	Linear. Necessário aprender React.
Renderização	Server-side.	Client-side.
Renderização parciais do *frontend*	Facil com o uso do htmx.	Necessário utilizar funções do React
Separação *frontend* *backend*	Não.	Sim.
Reutilização da API	Baixa.	Alta.

Fonte: Elaborado pelo autor.

No React a criação da API ficou mais simples, apenas realizando a busca no banco de dados e retornando o JSON sem se preocupar em renderizar componentes em HTML, porém para consumir a API foi mais trabalhoso ja que foi necessário escrever as funções para realizar o consumo da API e o tratamentos dos dados pelo *frontend*.

No React, um único arquivo consegue gerenciar a exibição, edição, salvamento e exclusão de um item, utilizando apenas dois *endpoints* da API para essas funcionalidades. Já no htmx, foi necessário criar três arquivos diferentes e três *endpoints* distintos para implementar as mesmas funcionalidades. Isso ocorre porque o htmx depende de componentes HTML separados para cada ação específica, enquanto no React é possível utilizar o mesmo componente para todas essas funções.

Em relação a tamanho de códigos, apesar do htmx necessitar mais arquivos e funções, em geral, é necessário escrever menos códigos comparado com o React. No React é necessário escrever uma função que busca os dados e faz a requisição para a API e no htmx é só adicionar os atributos na TAG dentro do HTML, não sendo necessário programar nada nessa parte, isso diminui o tamanho de código no *frontend*. No *backend* pelo fato do htmx precisar de componentes e *endpoints* exclusivos, a API fica maior em relação com a API do React.

Observando a quantidade de requisições, no React o número de chamadas à API é mais controlado, pois o desenvolvedor define exatamente quando e como cada requisição será realizada, podendo agrupar múltiplas ações em uma única chamada, o que reduz o volume total de requisições. Já no htmx, as requisições são disparadas automaticamente a partir dos eventos configurados nas TAGs HTML, como cliques ou carregamento de página. Com isso, pode haver um aumento no número de chamadas, especialmente quando as ações são divididas em componentes e endpoints distintos. Enquanto no React é possível realizar várias operações em um único endpoint, no htmx frequentemente cada operação exige um endpoint próprio, o que pode gerar mais requisições ao servidor durante a interação do usuário com a aplicação.

Como consequência de um maior número de requisições, pode haver um aumento na carga do servidor, consumo de banda e latência na resposta ao usuário. No caso do htmx, como as requisições são mais fragmentadas e frequentes, o servidor precisa processar múltiplas chamadas menores, o que pode impactar a escalabilidade da aplicação em cenários com grande volume de acessos simultâneos. Além disso, o maior número de conexões pode aumentar o tempo total de carregamento em redes com alta latência ou em dispositivos com limitações de desempenho. Já no React, com a possibilidade de consolidar múltiplas operações em uma única requisição, o tráfego é mais otimizado, reduzindo o tempo de resposta geral e o uso de recursos no backend.

Em comparação com os trabalhos relacionados foi observado que o htmx traz beneficios significativos no *frontend* para o consumo das APIs como mencionado por eles mas como visto nesse projeto, com o htmx, o foco do desenvolvimento muda do *frontend* para o *backend* tornando o desenvolvimento mais complexo levando em conta a quantidade de *end-points* e componentes necessários para o seu funcionamento.

De modo geral, considerando todo o processo desde a criação até o consumo da API, a utilização do htmx trouxe benefícios e melhorias em relação ao React, especialmente por simplificar o consumo da API.

4 CONCLUSÃO

Este trabalho trouxe uma analise referente ao uso do htmx em comparação com o uso de funções padrões do Javascript para o consumo de APIs sendo observado desde a criação da API até o seu consumo com dois sistemas distintos onde foi possível analisar as principais diferenças entre os dois sistemas.

Os resultados demonstraram que o HTMX oferece uma melhoria significativa no consumo da API no frontend, graças ao uso simplificado de seus atributos embutidos nas TAGs HTML, o que facilita a comunicação com o *backend* em comparação com a função padrão `fetch` do JavaScript. Essa abordagem reduz a complexidade do código *frontend*, tornando-o mais limpo e direto, além de acelerar a implementação das interações com a API.

O uso do htmx pode resultar em um número maior de requisições à API, uma vez que cada interação geralmente demanda uma chamada específica ao servidor para buscar e renderizar fragmentos HTML. Essa fragmentação torna o frontend mais simples, mas aumenta o volume de comunicação com o backend.

Já com o uso do fetch no React, foi possível consolidar diversas operações em menos chamadas, reduzindo a quantidade total de requisições realizadas. Dessa forma, o htmx pode gerar maior tráfego de rede e sobrecarga no servidor, enquanto o fetch permite um controle mais centralizado e otimizado do fluxo de dados.

Utilizando o htmx o foco do desenvolvimento muda para o *backend*, pois é necessário criar uma API que retorne diretamente fragmentos HTML para cada interação, ao invés do formato JSON tradicional. Isso pode tornar o desenvolvimento da API mais complexo e exigir um maior cuidado na estruturação das respostas, no gerenciamento das *views* e na

renderização dos componentes HTML. Dessa forma, apesar de simplificar o *frontend*, o HTMX pode aumentar a complexidade e o esforço necessário para manter e desenvolver o *backend*.

Como trabalho futuro indico realizar uma comparação em um projeto de maior escala utilizando outros atributos que o htmx oferece com o objetivo de avaliar a usabilidade do htmx em projetos mais complexos.

REFERÊNCIAS

COELHO, B. **Método comparativo: o que é, quais os tipos e como funciona**. 2022. Acessado em: 26 maio 2024. Disponível em: <https://blog.mettzer.com/metodo-comparativo/#:~:text=De%20todo%20modo%2C%20as%20comparaÃğÃµes,novas%20ideias%20e%20novos%20conceitos.>

FARIAS, L. H. C. R. **Estudo comparativo da utilização de design patterns no desenvolvimento de aplicação web utilizando frameworks front-end**. 2022.

FERREIRA, H. K.; ZUCHI, J. D. **Análise comparativa entre frameworks frontend baseados em javascript para aplicações web**. 2018. 111–123 p.

HTMX. **</> htmx - Documentation**. 2020. Acessado em: 06 nov. 2024. Disponível em: <https://htmx.org/docs/>.

HTMX. **</> htmx - high power tools for html**. 2020. Acessado em: 20 maio 2024. Disponível em: <https://htmx.org/>.

KAURINOVIĆ, T. **Razvoj web aplikacije korištenjem okvira Django i biblioteke HTMX**. Tese (Doutorado) — University of Rijeka. Faculty of Informatics and Digital Technologies, 2022.

MAIOR, M. J. V. S. **Análise comparativa de performance de frameworks para APIs Rest**. Dissertação (B.S. thesis) — UNIVERSIDADE FEDERAL DE PERNAMBUCO, 2023.

PAAKKANEN, J. **Upcoming JavaScript web frameworks and their techniques**. 2023.

STACK-OVERFLOW-SURVEY. **Stack Overflow Developer Survey 2023**. 2023. Acessado em: 20 maio 2024. Disponível em: <https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages>.

STACK-OVERFLOW-SURVEY. **Technology | 2024 Stack Overflow Developer Survey**. 2024. Acessado em: 11 nov. 2024. Disponível em: <https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language>.

STEFANOV, S. **Padrões JavaScript**. 2011.

VIEIRA, P. D. d. S. **Desenvolvimento de um sistema SAAS de baixo custo para controle da cadeia produtiva em uma pequena manufatura**. 2024.