Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Aplicações orientadas a hipermídia como solução para a Web moderna

Fernando Henrique Junqueira Muniz Barbi Silva

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Paulo Roberto Miranda Meirelles

São Paulo
2024

“Si mundus vos odit, scitote quia
me priorem vobis odio habuit.”
Ioann. 15, 18

Agradecimentos

"Manete in me, et ego in vobis. Sicut palmes non potest ferre fructum a semetipso, nisi manserit in vite, sic nec vos, nisi in me manseritis."
— Evangelium secundum Ioannem 15, 4

Agradeço a Deus, pois nada bom pode vir de mim, somente d'Ele.

Agradeço à minha noiva, Julia, a mulher com quem decidi passar o resto da minha vida, que durante todo este trabalho confiou em mim mais que eu mesmo.

Agradeço também ao meu orientador, o Prof. Paulo Meirelles, que aceitou me supervisionar durante o desenvolvimento deste projeto e me deu liberdade para tratar do tema com o mesmo entusiasmo com o qual o apresentei.

Por fim, agradeço aos meu familiares por todo o apoio diante das mais variadas circunstâncias.

Resumo

Fernando Henrique Junqueira Muniz Barbi Silva. **Aplicações orientadas a hipermídia como solução para a Web moderna**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

O cenário atual do desenvolvimento de software para a Web é caracterizado pela ampla adoção dos arcabouços e bibliotecas de *Single-Page Application* (SPA) para a construção da interface do usuário. Embora este modelo não tenha sido concebido como uma solução universal, ele pode ser percebido como tal por parte dos desenvolvedores, que frequentemente o adotam para seus projetos, incluindo aqueles que não recebem nenhum benefício deste estilo arquitetônico, mas apenas sua complexidade inerente. Este trabalho apresenta uma análise crítica desse panorama, fundamentando-se nas restrições REST descritas por Roy Fielding em sua tese de doutorado *"Architectural Styles and the Design of Network-based Software Architectures"* (Fielding, 2000). Além disso, propõe a arquitetura de uma aplicação orientada a hipermídia (HDA) como uma solução que respeita as restrições REST e que pode implementar padrões modernos de interface de usuário por meio de estratégias de transclusão de documentos. O desenvolvimento do protótipo de um painel de administração para o projeto BikeSP, implementado em duas versões — a primeira utilizando SPA com React e a segunda utilizando HDA com htmx — serve como prova de conceito para demonstrar a viabilidade e as implicações da adoção da arquitetura HDA, com ênfase nos aspectos relacionados à experiência de desenvolvimento.

Palavras-chave: hipermídia. aplicação orientada a hipermídia. sistema de hipermídia. *representational state transfer*. *single-page application*. desenvolvimento web. BikeSP. *RESTful API*.

Abstract

Fernando Henrique Junqueira Muniz Barbi Silva. **Hypermedia-driven applications as an approach for the modern Web**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

The current scenario of software development for the Web is characterized by the widespread adoption of Single-Page Application (SPA) frameworks and libraries for building the user interface. Although this model was not conceived as a universal solution, it can be perceived as such by developers, who often adopt it for their projects, including those that do not benefit from this architectural style, but are only affected by its inherent complexity. This paper presents a critical analysis of this landscape, based on the REST constraints outlined by Roy Fielding in his thesis *"Architectural Styles and the Design of Network-based Software Architectures"* (Fielding, 2000). Furthermore, it proposes the architecture of a hypermedia-driven application (HDA) as a solution that respects the REST constraints and can implement modern user interface patterns through document transclusion strategies. The development of a prototype for an administration panel for the BikeSP project, implemented in two versions — one using SPA with React and the the other using HDA with htmx — serves as a proof of concept to demonstrate the feasibility and implications of adopting the HDA architecture, with a focus on development experience.

Keywords: hypermedia. hypermedia-driven application. hypermedia system. representational state transfer. single-page application. web development. BikeSP. RESTful API.

Lista de abreviaturas

AJAX	JavaScript e XML Assíncronos (*Asynchronous JavaScript and XML*)
API	Interface de Programação de Aplicações (*Application Programming Interface*)
CDN	Rede de Distribuição de Conteúdo (*Content Delivery Network*)
CSS	Folhas em Estilo Cascata (*Cascading Style Sheets*)
DOM	Modelo de Documento por Objetos (*Document Object Model*)
HATEOAS	Hipermídia como motor do estado da aplicação (*Hypermedia As The Engine Of Application State*)
HDA	Aplicação Orientada a Hipermídia (*Hypermedia-Driven Application*)
HTML	Linguagem de Marcação de Hipertexto (*HyperText Markup Language*)
HTTP	Protocolo de Transferência de Hipertexto (*HyperText Transfer Protocol*)
JS	JavaScript (*ECMAScript*)
JSON	Notação de Objetos JavaScript (*JavaScript Object Notation*)
JSX	Extensão de Sintaxe de JavaScript (*JavaScript Syntax Extension*)
MPA	Aplicação de Múltiplas Páginas (*Multi-Page Application*)
REST	Transferência de Estado Representacional (*Representational State Transfer*)
RPC	Chamada Remota de Procedimento (*Remote Procedural Call*)
SoC	Separação de Conceitos (*Separation of Concerns*)
SPA	Aplicação de Página Única (*Single-Page Application*)
SSR	Renderização no Lado do Servidor (*Server-Side Rendering*)
URI	Identificador Uniforme de Recursos (*Uniform Resource Identifier*)
URL	Localizador Uniforme de Recursos (*Uniform Resource Locator*)
XML	Linguagem de Marcação Extensível (*Extensible Markup Language*)
WWW	Rede Mundial de Computadores (*World Wide Web*)

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

[1] Do inglês: *stateless.*

[2] Do inglês: *layered system.*

Introdução

O cenário atual do desenvolvimento de software para a Web é marcado pela grande presença dos arcabouços e bibliotecas de JavaScript para a construção da interface do usuário. Estes arcabouços são utilizados principalmente para desenvolver software do tipo *Single-Page Application*, que, de modo genérico, consiste em utilizar diretamente o código em JavaScript no cliente para construir e atualizar a interface do usuário, em vez de contar com as funcionalidades embutidas nos navegadores para essa funcionalidade. Essas tecnologias tornaram-se populares rapidamente, pois surgiram num contexto em que as ferramentas básicas (os blocos de construção da web: HTML, CSS e as Web APIs) apresentavam soluções precárias para o desenvolvimento de aplicações que exigiam uma interface de usuário mais interativa, dinâmica e modular.

Esta crescente popularidade levou à adoção desses arcabouços em uma ampla gama de projetos, **incluindo aqueles que não requerem interações complexas.** Embora não tenham sido concebidos como uma solução universal, essa é a percepção de uma parte da comunidade de desenvolvimento para a Web, de modo que é possível encontrar na formação de novos desenvolvedores aqueles que iniciam seus estudos com o uso dessas bibliotecas, sem a compreensão prévia dos seus fundamentos e sem o entendimento do que ocorre concretamente debaixo de sua abstração.**A consequência prática deste fenômeno é que muitos desenvolvedores têm se afastado dos princípios fundamentais da Web.**

A maior parte da Web é estática e não demanda interatividade intensa, logo não necessita do alto nível de abstração e das implicações arquiteturais inerentes a essas bibliotecas. **Projetos simples, que deveriam ser triviais, não se beneficiam deste estilo arquitetônico, mas recebem apenas sua complexidade**, resultando numa arquitetura complicada, abarrotada de dependências externas, de conceitos abstratos e de uma série de processos que, em última análise, não contribuem para o sucesso da aplicação, mas apenas sobrecarregam o time de desenvolvimento com obstáculos para a eficácia da manutenção do código.

Objetivo

O presente trabalho busca apresentar uma análise crítica das implicações desta arquitetura, fundamentando-se nas restrições REST descritas por Roy Fielding em sua tese de doutorado *"Architectural Styles and the Design of Network-based Software Architectures"* (Fielding, 2000). Além disso, propõe a arquitetura de uma aplicação orientada a hipermídia (HDA) como uma solução que respeita as restrições REST e que pode implementar padrões modernos de interface de usuário por meio de estratégias de transclusão de documentos.

Propomos demonstrar que é possível reduzir significativamente a complexidade das aplicações e que é viável construir aplicações orientadas a hipermídia, utilizando técnicas tradicionais, para criar interfaces de usuário que atendem a diversos padrões modernos. O objetivo deste trabalho é, portanto, **oferecer uma alternativa ao paradigma *Single-Page Application* através da retomada da compreensão da Web enquanto um sistema de hipermídia.**

Organização do texto

O texto está organizado em quatro capítulos. No primeiro capítulo, define-se o que é a arquitetura REST como foi descrito na tese de doutorado de Roy Fielding (Fielding, 2000), estilo arquitetônico que rege os princípios estruturais da Web moderna. Estes conceitos serão fundamentais para oferecer uma análise coerente e uma solução eficaz nos capítulos seguintes.

O diagnóstico do estado atual da Web será realizado no segundo capítulo, explicando em que consiste a arquitetura *Single-Page Application*, como ela rompeu com os princípios de arquitetura REST e como este paradigma implica em mais complexidade no processo de desenvolvimento para a Web.

No terceiro capítulo, será apresentada uma possível solução: uma aplicação orientada a hipermídia. Neste capítulo, será definido o que é hipermídia, o que carateriza um sistema de hipermídia e o processo de transclusão no cliente, que permitirão a adoção de padrões modernos de interface gráfica a partir de técnicas tradicionais.

O último capítulo contém uma análise da aplicação dos conceitos discutidos até então no desenvolvimento do protótipo de um painel de administração para o projeto BikeSP,[3] implementado em duas versões — a primeira utilizando SPA com React e a segunda utilizando HDA com htmx — que servirão como prova de conceito para demonstrar a viabilidade e as implicações da adoção da arquitetura HDA, com ênfase nos aspectos relacionados à experiência de desenvolvimento.

[3] Um projeto desenvolvido para implementar um programa voltado a dar créditos no Bilhete Único para quem se desloca utilizando bicicletas na cidade de São Paulo. Mais detalhes podem ser encontrados em https://interscity.org/bikesp.

Capítulo 1

O estilo arquitetônico REST

Não há bala de prata.[1] Roy Fielding argumenta em sua tese de doutorado que "uma boa arquitetura não é criada no vácuo". Isto significa que as decisões de *design* em nível de arquitetura **devem** ser tomadas levando em conta o contexto dos requisitos funcionais, comportamentais e sociais do sistema a ser projetado (Fielding, 2000).

Um problema que sempre esteve presente agravou-se ao longo dos últimos anos no meio do desenvolvimento de software para a Web: muitos projetos de software são iniciados com a decisão de adotar as tecnologias mais populares. O fato de uma biblioteca de código ter se popularizado leva alguns a crerem, erroneamente, que não apenas esta biblioteca será uma solução para qualquer problema particular, mas será a melhor solução, ou até mesmo a única aceitável. **Não é o projeto que deve servir à tecnologia, mas a tecnologia ao projeto.** Ainda mais grave: a adoção de determinadas linguagens e bibliotecas trazem consigo decisões arquiteturais. Não faz sentido adotar uma tecnologia *a priori*, por conseguinte se submeter às suas implicações arquiteturais, para apenas mais adiante descobrir se os requisitos do sistema exigiam tal arquitetura.

Os desenvolvedores de projetos para a Web se esqueceram do princípio de que "a forma segue a função"[2] e se tornaram partidários do método de "*design* pelo que está na moda". Fielding atribui o abandono da "forma segue a função" à falta de entendimento do motivo pelo qual um determinado conjunto de restrições arquiteturais são úteis, ou seja: o juízo por trás de uma boa arquitetura de software não é aparente aos projetistas quando esta arquitetura é selecionada para ser aplicada novamente (Fielding, 2000).

Esta é exatamente a situação em que o desenvolvimento para a Web se encontra hoje em dia: linguagens, tecnologias e bibliotecas são utilizadas sem pleno entendimento por parte do desenvolvedor dos efeitos colaterais em potencial, especialmente na arquitetura. O grande diferencial de um bom desenvolvedor de software é ser capaz de avaliar objetivamente o impacto das suas várias decisões de *design* no comportamento do sistema. Por isso, não convém iniciar este estudo a partir das bibliotecas que serão utilizadas, mas é necessário

[1] Referência ao termo *"silver bullet"* utilizado no famoso artigo *"No Silver Bullet: Essence and Accidents of Software Engineering"* de Fred Brooks (Brooks, 1987).

[2] Do inglês: *form follows function.*

estabelecer restrições arquiteturais para uma aplicação Web que estejam de acordo com a arquitetura própria da WWW (*World Wide Web*). Estas restrições serão fundamentais para o juízo na seleção das tecnologias do projeto e para a análise da eficácia das diferentes implementações. Felizmente, o levantamento de requisitos da WWW já foi realizado por Roy Fielding, bem como a sugestão de um estilo arquitetônico adequado para suas aplicações, o qual foi cunhado de REST (*Representational State Transfer*) (Fielding, 2000).

1.1 *Representational State Transfer*

Representational State Transfer (REST) é o estilo arquitetônico para sistemas de hipermídia distribuídos descrito por Roy Fielding[3] em sua tese de doutorado. Fielding derivou o estilo REST a partir de outros estilos arquitetônicos baseados em rede, incorporando algumas restrições adicionais para atender a necessidades específicas.[4]

Fielding, 2000 afirma que *"a Rede Mundial de Computadores obteve sucesso em grande parte porque sua arquitetura de software foi projetada para satisfazer as necessidades de um sistema de hipermídia distribuído na escala da Internet"*. Ele deu ênfase particular na estruturação da arquitetura de sistemas distribuídos de forma a tirar o máximo proveito da infraestrutura já existente da *World Wide Web*. Ao fim de sua tese, Fielding formaliza os conceitos e define o REST como um modelo de arquitetura para sistemas baseados na Web. A ideia central do REST é de que a comunicação entre sistemas deve ser **simples, escalável e baseada em um conjunto claro de princípios**.

Desde então, o REST tem sido usado para guiar o *design* e o desenvolvimento da arquitetura da Web moderna. O REST será utilizado como **modelo de estilo arquitetônico para um projeto de software para a Web**, pois enfatiza a escalabilidade das interações entre componentes, a generalidade das interfaces, a implantação independente de componentes, a redução da latência da interação através de componentes intermediários, reforça a segurança e encapsula sistemas legados (Fielding, 2000).

1.2 As restrições do estilo REST

Os princípios de engenharia de software que orientam o estilo REST são conservados através de um conjunto de restrições de interação. Portanto, para um sistema ser considerado *RESTful*[5] é preciso que ele respeite as restrições do estilo REST assim como foram definidas por Fielding, 2000.

---

[3] Roy Fielding foi um dos principais arquitetos do protocolo HTTP (*HyperText Transfer Protocol*) e também contribuiu significativamente para a criação do padrão URI (*Uniform Resource Identifier*) e do sistema de navegação baseado em URLs (*Uniform Resource Locator*).

[4] A restrição de **interface uniforme**, por exemplo, se destaca no estilo REST por diferenciá-lo de outros estilos arquitetônicos baseados em rede.

[5] Isto é, estar em conformidade com todos os princípios e convenções da arquitetura REST.

1.2.1 Modelo Cliente-Servidor

Fielding, 2000 argumenta que o princípio por detrás desta restrição é a "separação de conceitos" (SoC),[6] pois, ao separar as preocupações da interface do usuário das preocupações de armazenamento de dados, melhora-se a portabilidade da interface do usuário entre várias plataformas e aumenta-se a escalabilidade ao simplificarmos os componentes do servidor.

Restrição 1: Cliente-Servidor
Os requerentes dos serviços (clientes) são separados do fornecedor de um recurso ou serviço (servidor) através de uma interface bem definida.

O servidor compartilha recursos com os clientes, já um cliente não compartilha qualquer de seus recursos, mas solicita um conteúdo ou serviço ao servidor. Portanto, os clientes enviam *requisições* e o servidor devolve *respostas*.

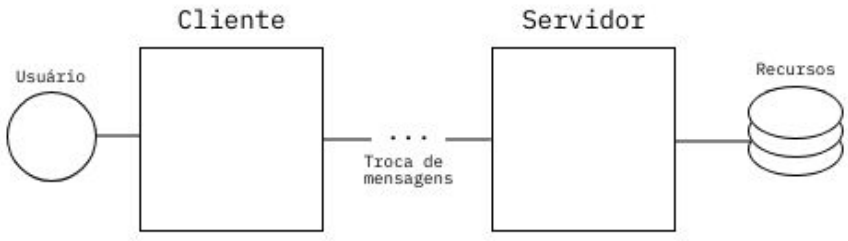


Figura 1.1: *Restrição de cliente-servidor.*

Esta separação é o que apoia a escala mundial da WWW, dada em múltiplos domínios organizacionais, pois permite que os componentes evoluam de maneira independente.

1.2.2 Ausência de estado[7]

O serviço (ou servidor) não deve manter informações sobre o estado de uma interação no intervalo entre duas ou mais requisições. Cada requisição vinda do cliente deve conter toda a informação necessária para sua interpretação, sem depender de nenhum contexto armazenado no lado do servidor. A comunicação deve ser sem estado por natureza. O estado da sessão, portanto, é mantido inteiramente no cliente (Fielding, 2000).

Restrição 2: Ausência de estado
Cada requisição feita a um servidor por um cliente específico deve ser independente e conter todas as informações necessárias para ser inteiramente processada.

Portanto, em um sistema *RESTful*, cada vez que um cliente faz uma solicitação, o servidor não tem conhecimento sobre as requisições anteriores. Cada requisição é tratada como se fosse a primeira, sem que o servidor precise recuperar o contexto das interações anteriores.

---

[6] Do inglês: *separation of concerns.*

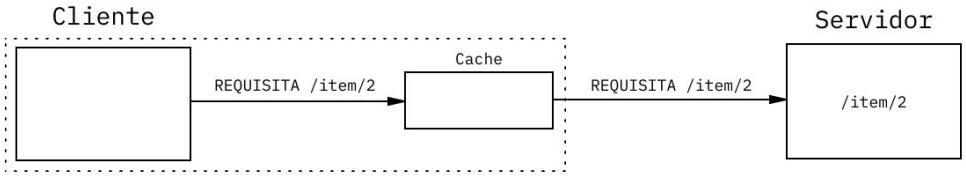
[7] Do inglês: *stateless.*

1.2.3 Suscetível ao uso de *cache*

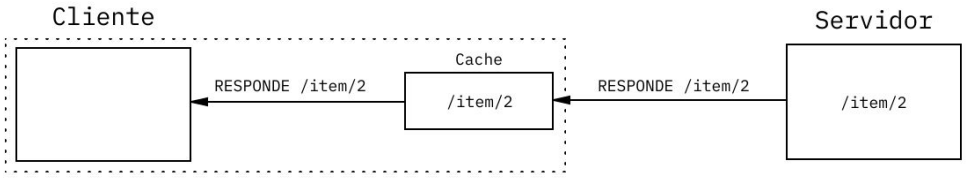
Embora o estilo REST não permita que o servidor mantenha estado entre requisições, ele permite que o cliente (ou algum intermediário, como *proxies* e *gateways*) armazene cópias em *cache* das respostas recebidas do servidor, Esta restrição foi inserida por Fielding para aprimorar a eficiência da rede, ao reduzir o número de requisições enviadas ao servidor e a velocidade de resposta. No entanto, cabe ao servidor rotular as respostas indicando se são passíveis de serem armazenadas em *cache* ou não.

Restrição 3: Cache
Dados dentro de uma resposta do servidor a uma requisição devem ser rotulados implícita ou explicitamente como passíveis de serem armazenados em cache ou não. Se for passível, o receptor recebe o direito de reutilizar esses dados para requisições posteriores equivalentes (Fielding, 2000).

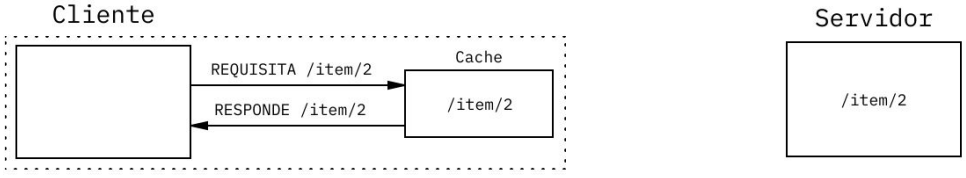
O receptor pode ser tanto o cliente quanto um servidor intermediário, como um *proxy*. Um esquema simples do funcionamento desta restrição está descrito na Figura 1.2.



(a) *Etapa 1: O cliente verifica se o recurso se encontra em cache. Se não houver, a requisição é encaminhada ao servidor.*



(b) *Etapa 2: O servidor responde com o recurso solicitado. Neste caso, o recurso é passível de ser armazenado em cache, portanto uma cópia é salva no cache do cliente, e a resposta é encaminhada para a função que a solicitou.*



(c) *Etapa 3: Caso o cliente faça a requisição do recurso novamente, poderá recuperá-lo diretamente do cache, sem a necessidade de encaminhar a requisição para o servidor através da rede.*

Figura 1.2: *Esquema de funcionamento da restrição de cache.*

1.2.4 Interface uniforme

A restrição da interface uniforme é fundamental para o *design RESTful* do sistema, e é a responsável por distinguí-lo dos demais estilos arquitetônicos. É o que torna a arquitetura REST simples e escalável, pois, ao desacoplar a arquitetura, permite que cada parte evolua independentemente, e reduz o nível de abstração e complexidade na comunicação entre componentes.

Restrição 4: Interface uniforme
É necessário uma interface padronizada e consistente entre o cliente e o servidor que respeite as sub-restrições: identificação de recursos, manipulação de recursos por meio de representações, mensagens autodescritivas e hipermídia como motor do estado da aplicação.

Para obter uma interface uniforme no REST, são necessárias mais quatro sub-restrições arquiteturais para orientar o comportamento desses componentes, descritas na subseções abaixo.

Identificação de recursos

Cada recurso no sistema deve ser identificado de maneira unívoca utilizando um identificador de recurso, como URI (*Uniform Resource Identifier*), URL (*Uniform Resource Locator*), URN (*Uniform Resource Name*) etc.

Manipulação de recursos por meio de representações

Ao interagir com um sistema *RESTful*, o cliente manipula os recursos através do envio e recebimento de representações desses recursos.

O servidor pode devolver HTML (*HyperText Markup Language*) ao cliente mesmo se esta não for a representação interna dos recursos no banco de dados, pois os recursos são conceitualmente separados das representações devolvidas.

A representação do recurso pode conter tanto dados quanto metadados sobre a requisição. Em uma requisição do cliente ao servidor em HTTP (*HyperText Transfer Protocol*), por exemplo, os dados estão contidos no corpo da resposta e os metadados costumeiramente são inseridos no cabeçalho. E estas informações devem ser suficientes para manipular o recurso no servidor.

Como exemplo, ao solicitar a primeira página de uma lista de usuários ao servidor, pode-se devolver ao cliente uma resposta em HTTP, contendo em seu *payload* o código em HTML contido no Programa 1.1. Por mais que, no servidor, os recursos de usuários não sejam representados em HTML, mas sim com a representação interna do banco de dados, é possível manipular estes recursos através da representação em HTML enviada ao cliente, seja enviando uma requisição do método `POST` a partir do formulário ou clicando em um *link* para solicitar outro recurso, no caso do exemplo seria a próxima página de usuários.

Programa 1.1 Exemplo de manipulação de recursos por meio de representações em HTML.

```
<!DOCTYPE html>
<html lang="pt-BR">

<body>
  <h1>Lista de Usuários (Página 1)</h1>

  <!-- Tabela de usuários -->
  <table>
    <thead>
      <tr>
        <th>ID</th>
        <th>Nome</th>
        <th>Email</th>
        <th>Ações</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>1</td>
        <td>Ana Silva</td>
        <td>ana.silva@email.com</td>
        <td><button>Editar</button> <button>Excluir</button></td>
      </tr>
      <tr>
        <td>2</td>
        <td>Carlos Souza</td>
        <td>carlos.souza@email.com</td>
        <td><button>Editar</button> <button>Excluir</button></td>
      </tr>
      <tr>
        <td>3</td>
        <td>Maria Oliveira</td>
        <td>maria.oliveira@email.com</td>
        <td><button>Editar</button> <button>Excluir</button></td>
      </tr>
    </tbody>
  </table>

  <!-- Formulário para adicionar novo usuário -->
  <h2>Cadastrar Novo Usuário</h2>
  <form action="/usuarios" method="POST">
    <label for="nome">Nome:</label><br>
    <input type="text" id="nome" name="nome" required>

    <input type="submit" value="Cadastrar">
  </form>

  <!-- Link para a próxima página -->
  <a href="/usuarios?pagina=2">Ir para a próxima página</a>
</body>

</html>
```

Mensagens autodescritivas

Toda informação necessária para exibir e também realizar operações em cima dos recursos sendo representados devem estar presentes na resposta.

Cada mensagem enviada pelo cliente ou pelo servidor deve incluir informações suficientes para descrever como processá-la. Estas informações podem estar contidas nos metadados, como por exemplo a descrição do tipo de mídia no atributo `Content-Type` do HTTP, ou pode estar contida diretamente nos dados, como os controles de hipermídia no HTML da Figura 1.1 indicando os recursos para os quais o cliente poderá navegar a partir deste.

Hipermídia como motor do estado da aplicação (HATEOAS)

A última sub-restrição da interface uniforme do REST é o uso de hipermídia como motor do estado da aplicação (HATEOAS).[8] O cliente não deve necessitar de saber previamente a estrutura do servidor ou quais recursos ele pode acessar. Em vez disso, ao acessar um recurso inicial, deve receber *links* dinâmicos que apontam para outros recursos disponíveis (Gross *et al.*, 2023).

Isso torna o sistema mais flexível e desacoplado, uma vez que o cliente, através dos *links* fornecidos pelo servidor de forma dinâmica, torna-se capaz de descobrir os recursos disponíveis conforme interage com a aplicação, evitando que seja necessário possuir informações codificadas sobre o servidor.

Na representação dos recursos em HTML (Figura 1.1) há um exemplo claro, pois o HTML, como hipermídia, codifica o estado da aplicação, e em seu corpo diz o que pode e o que não pode ser feito com esta representação em particular (através dos controles de formulário e de *link*).

No exemplo, a hipermídia do HTML junto do protocolo de hipermídia HTTP serviram de interface uniforme na comunicação entre cliente e servidor, e permitem o uso de um mesmo navegador como cliente para todas as páginas web, pois o navegador não precisa conhecer detalhes de implementação ou as estruturas dos diversos servidores com os quais troca mensagens.

1.2.5 Sistema em camadas[9]

A última restrição obrigatória de um sistema *RESTful* é a arquitetura "em camadas".

Restrição 5: Sistema em camadas
A arquitetura do sistema deve ser restringida em camadas hierárquicas, de modo que um componente não possa ver além da camada imediata com a qual está interagindo.

Ao restringirmos o comportamento do sistema a uma única camada, limitamos a complexidade geral e promovemos a independência da estrutura. Isto permite que múltiplos

[8] Abreviatura derivada do inglês: *Hypermedia As The Engine Of Application State.*

[9] Do inglês: *layered system.*

servidores ajam como um intermediário entre o cliente e o eventual servidor da única fonte de verdade.[10]

O sistema em camadas permite a escala em nível mundial, pois viabiliza a existência de CDNs (*Content Delivery Networks*), resultando no envio mais rápido do recurso ao usuário final e na redução da carga no servidor de origem.

1.2.6 Código sob demanda (*scripting*)

A introdução da restrição de código sob demanda simplifica a implementação do cliente, pois reduz o número de recursos que precisam ser implementados previamente. No entanto, diminui a visibilidade do sistema, então é apenas uma restrição opcional dentro do estilo REST (Fielding, 2000).

Restrição 6: Código sob demanda (Opcional)
Um sistema REST pode permitir que a funcionalidade do cliente seja estendida através do download e execução de código sob demanda, na forma de applets ou scripts.

[10] Do inglês: *source of truth server*. Indica o servidor que atua como o detentor real dos recursos, e não um intermediário que armazena cópias ou se comunica com outras fontes para buscar os recursos.

Capítulo 2

O estado atual do desenvolvimento para a Web

Um dos tipos de software mais sofisticado desenvolvido nos tempos atuais é o navegador web. Um navegador web é capaz de compreender HTML e CSS (*Cascading Style Sheets*), que são os blocos de construção fundamentais para as páginas da web. Além disso, é capaz de interpretar uma série de outros formatos de arquivo, como imagens, áudios, vídeos etc. Como se não bastasse tamanha versatilidade, há mais uma qualidade que amplia consideravelmente sua sofisticação: esse tipo de software fornece ao usuário um ambiente de programação e execução de JavaScript.[1]

O motor de execução de JavaScript[2] embutido nos navegadores é tão poderoso que permite aos desenvolvedores a criação de aplicações inteiras quase tão sofisticadas quanto aplicações nativas, mas que são executadas dentro do navegador. Foi devido ao avanço rápido e à capacidade impressionante de execução de script nesse ambiente que alguns desenvolvedores (especialmente aqueles que desenvolvem bibliotecas de código para a Web) passaram a deixar de lado as funcionalidades de hipermídia embutidas nos navegadores.

Com o passar do tempo, o navegador passou a ser encarado como um interpretador de código, que seria o responsável por executar e gerenciar os recursos da aplicação desenvolvida inteiramente em JavaScript. As aplicações construídas deste modo[3] passaram a ser chamadas de SPAs (*Single-Page Applications*) e se tornaram a norma emergente no desenvolvimento para a Web.

---

[1] Em inglês, é referido como *JavaScript Runtime*.

[2] Este motor de execução é referido em inglês como *JavaScript Engine*. O motor de JavaScript é a parte do ambiente de execução responsável pela interpretação e execução do script, pela compilação JIT (*Just-In-Time*) e pelo gerenciamento de memória. Além do motor, o ambiente de execução de JavaScript dos navegadores oferece uma série de outras funcionalidades, como os módulos de sistema, as APIs de Web (DOM, Timers, Fetch etc) e o laço de eventos (*event loop*).

[3] Isto é, ignorando as funcionalidades de hipermídia já embutidas no navegador em detrimento de construir uma aplicação inteiramente em JavaScript.

2.1 *Single-Page Application*

A forma tradicional de navegação entre páginas nos navegadores ocorre através de requisições HTTP disparadas pelo próprio navegador conforme o usuário interage com algum controle de hipermídia do HTML (como elementos de âncora `<a>` ou de formulário `<form>`). O servidor responderá a essa requisição enviando outro documento HTML que então substituirá completamente a página exibida pelo navegador. A navegação, portanto, se dá através do carregamento e da substituição de páginas inteiras, e esse processo é gerenciado pelo próprio navegador.[4] Este modelo existe desde os primórdios da Web, e sua funcionalidade se encontra embutida na implementação dos navegadores.

O termo *Single-Page Application* (SPA) é popularmente utilizado para se referir a uma aplicação que, em vez de utilizar o modelo de navegação tradicional descrito anteriormente, irá **dinamicamente atualizar a página vigente com os dados fornececidos pela resposta do servidor**. Em um SPA, uma atualização de página nunca ocorre propriamente[5] Em vez disso, utiliza-se JavaScript para construir e atualizar a interface gráfica do usuário diretamente.

Em vez de deixar que o comportamento padrão do navegador se responsabilize tanto pelo disparo das requisições como pela atualização da página, um SPA tipicamente dispara requisições ao servidor através de AJAX (*Asynchronous JavaScript and XML*), comunicando-se através de uma API JSON (*JavaScript Object Notation*). Ao receber a resposta JSON do servidor, o código em JavaScript do cliente irá atualizar a página vigente com esses novos dados, geralmente através de alguma biblioteca reativa de *front-end*[6] , também escrita em JavaScript.[7] Uma comparação dos dois mecanismos de atualização de página pode ser vista na Figura 2.1.

Portanto, uma aplicação SPA não está mais transitando entre as páginas utilizando controles de hipermídia, mas trocando "dados puros" (*plain data*) com o servidor para em seguida atualizar o conteúdo dentro de uma única página. Como não é a página que é alterada, mas sim seu conteúdo, chama-se "aplicação de página única" (*Single-Page Application*).

2.1.1 As restrições implícitas de um SPA

À primeira vista, pode parecer que o termo *Single-Page Application* refere-se apenas a uma descrição da estratégia de atualização da interface gráfica. Porém, basta analisar com um pouco mais de cautela para perceber que esta decisão de *design* implica alguns efeitos colaterais na arquitetura.

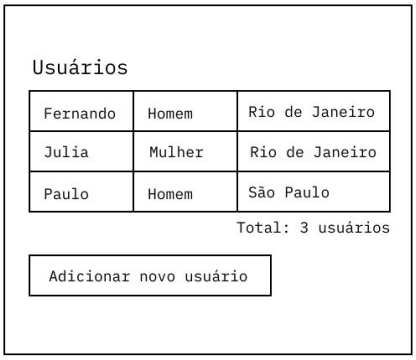
---

[4] O tempo MPA (*Multi-Page Application*) está se tornando popular para descrever aplicações com esse tipo de comportamento.

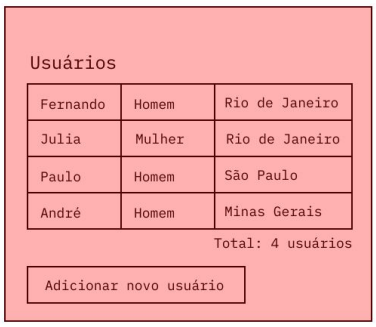
[5] Isto é, do jeito tradicional: recebendo uma página inteira como resposta do servidor e sobrescrevendo a atual com a nova.

[6] *Front-end* refere-se à parte de uma aplicação ou site com a qual o usuário interage diretamente. Pode se referir diretamente à interface do usuário.

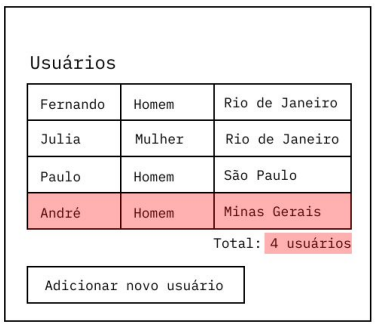
[7] Exemplos populares dessas bibliotecas e arcabouços são: React, Vue, Angular, Svelte, Solid etc.



(a) *Estágio inicial da página antes de o cliente clicar no botão para adicionar um novo usuário.*



(b) *Atualização da página no modo padrão: toda a página é substituída por um novo documento enviado pelo servidor. Os elementos atualizados estão marcados em vermelho.*



(c) *Atualização da página em um SPA: os elementos da interface são atualizados dinamicamente. Os elementos atualizados estão marcados em vermelho.*

Figura 2.1: *Comparação do mecanismo de atualização de página embutido nos navegadores com a implementação de um Single-Page Application.*

O cliente robusto

Em primeiro lugar, torna-se **necessário**[8] o uso de *scripting* no lado do cliente, uma vez que é preciso substituir o comportamento padrão do navegador pelo próprio gerenciamento de requisições e atualizações de página. Portanto, as interações com o servidor deverão ser definidas em JavaScript a ser executado pelo seu ambiente de execução.[9]

O código no lado do cliente será responsável por:

1. Disparar as requisições HTTP para o servidor;
2. Receber a resposta do servidor e decodificar o *payload* com base no estado do cliente;
3. Implementar o tratamento de possíveis erros;
4. Atualizar a interface gráfica com as informações obtidas no Item 2.

[8] Não é mais opcional, como visto na Seção 1.2.6.

[9] Geralmente através da API de `fetch`, disponível nos navegadores modernos.

Por exigir operações sensíveis, as aplicações do tipo SPA geralmente são escritas utilizando bibliotecas e arcabouços de JavaScript, para facilitar a manipulação desses eventos.[10] Em geral, essas bibliotecas utilizam o paradigma de **programação reativa** para a construção da interface gráfica, tendo em vista (pelo Item 4) que cabe também ao código do cliente atualizar a interface gráfica de forma dinâmica, e muitas vezes assíncrona. Logo, percebe-se que a parcela de código do sistema contida no cliente será mais complexa e robusta que o usual.

A interação com o servidor via API JSON

Apesar de a sigla AJAX fazer referência à XML (*Extensible Markup Language*), a prática quase universal atualmente é que o corpo da resposta HTTP à requisição seja codificado em formato JSON.[11]

É importante notar que a interação baseada em JSON com o servidor **não usa hipermídia**. A API JSON não responde com hipermídia, mas com JSON, que não possui hiperligações ou outros controles de hipermídia (Gross *et al.*, 2023). **A API JSON em um SPA é, na verdade, uma API de dados, somente** (Gross *et al.*, 2023).

2.2 Benefícios gerados pelo modelo SPA

O modelo *Single-Page Application* popularizou-se pois trazia consigo a possibilidade de resolver alguns problemas no desenvolvimento web.

2.2.1 Componentização e reutilização de código

O uso das bibliotecas de JavaScript, como *React*, por exemplo, trouxe consigo a prática da "componentização" do código.[12]

A arquitetura fundamental da Web baseia-se no conceito de separação de preocupações (ou separação de conceitos).[13] Pode-se constatar este fato não apenas na separação das camadas de comunicação na Internet, mas principalmente na divisão da construção das páginas em HTML, CSS e JavaScript: o HTML é principalmente utilizado para a organização do conteúdo da página, o CSS para a definição do estilo em que o conteúdo será apresentado, e o JavaScript para a definição de como o conteúdo se comportará e como o usuário poderá interagir com ele.

Ao definirmos um novo botão em uma página, seria necessário criar sua representação no documento HTML, indicando qual seria seu conteúdo interno (por exemplo, a *string* "Clique aqui"). Em seguida, faz-se referência ao elemento HTML dentro da folha

[10] O JavaScript é a linguagem de *scripting* própria da web. Existem outras alternativas para executar código no navegador, como *applets* ou utilizando *Web Assemly*, mas não se comparam à popularidade do uso de JavaScript, que é utilizado em 99% dos sites.

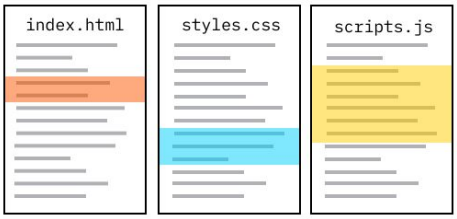
[11] Refere-se ao *payload* do HTTP, evidentemente.

[12] É importante notar que, na época em que o modelo SPA surgiu e começou a se tornar popular, ainda não estavam vastamente implementadas as funcionalidades de *Web Components* nos navegadores.

[13] Do inglês: *Separation of Concerns*.

de estilos CSS e então se descreve a aparência do botão (cores, fontes, bordas etc). Por fim, no arquivo de *scripts*, selecionamos o botão através do DOM e atribuímos a ele um comportamento específico ao ser clicado (como mudar de cor ou encaminhar o usuário para outra página). No modelo tradicional, portanto, todos os elementos da página possuem as definições de conteúdo interno, estilos e comportamentos nos respectivos arquivos. Para definir um único botão seria necessário manipular três arquivos diferentes, um para cada "preocupação" do SoC.

A partir do momento em que a aplicação passa a ser escrita inteiramente em JavaScript no SPA, torna-se possível criar "componentes de interface do usuário". Ao criar um botão, não seria mais necessário introduzir referências a ele em três documentos diferentes que também contêm informações de outros elementos da página, mas criar-se-ia apenas um arquivo, em que se descreveria o conteúdo, o estilo e o comportamento deste botão apenas.



(a) *Organização tradicional baseada na separação de conceitos. O código do componente estava espalhado entre os arquivos de acordo com a preocupação. Nestes arquivos também há código dos demais componentes.*

(b) *Arquivo de um componente SPA, contendo as definições de conteúdo, estilo e comportamento de um único componente no mesmo arquivo.*

Figura 2.2: *Efeito da componentização do SPA na disposição de código entre os arquivos.*

A consequência imediata deste tipo de técnica foi a maior facilidade na **reutilização de código**. Encapsulando o botão num único componente, facilitou-se a criação de novas instâncias deste componente, ao reduzir o número de arquivos a atualizar em caso de qualquer alteração.

2.2.2 Interatividade

O modelo SPA também ofereceu uma solução para a demanda da época de uma experiência de usuário mais fluida e interativa. Ao evitar o carregamento completo da página a cada atualização, o tempo de espera foi reduzido e a responsividade do sistema ao usuário foi aprimorada, dando a impressão de uma experiência mais ágil e contínua. Comportamentos que antes eram muito difíceis ou impossíveis de se obter no modelo tradicional puderam ser adotados com bastante agilidade, como atualizações da interface em tempo real, animações complexas e sistemas com longas e complicadas interações.

2.2.3 Especialização da equipe

Por fim, o modelo SPA favoreceu o movimento de especialização das equipes de desenvolvimento para a Web, dividindo claramente as responsabilidades entre *front-end* e *back-end*. Uma parcela da equipe se concentra na construção da interface e na interação do usuário, passa a ser responsável pela maior parte lógica de apresentação e manipulação de dados, utilizando arcabouços e bibliotecas reativos. A outra, por sua vez, trata da gestão de dados, das regras de negócio, da autenticação, do fornecimento de APIs JSON aos clientes etc. Esse paradigma permitiu que equipes especializadas em cada área pudessem atuar de maneira mais independente, se concentrando em aspectos específicos do sistema, ao mesmo tempo que a comunicação entre as camadas é realizada via APIs bem definidas.

2.3 As calamidades causadas pelo modelo SPA

Porém, o uso dessas bibliotecas e arcabouços sofisticados de JavaScript para a construção de aplicações do tipo SPA se tornou tão popular que estão sendo adotadas até mesmo para construir aplicações extremamente simples. A generalização do SPA causou o aumento desnecessário da complexidade dos sistemas, especialmente no código que será executado no lado do cliente. Idealmente, um SPA deveria ser utilizado somente nas arquiteturas em que os benefícios descritos anteriormente (componentização, interação robusta e especialização da equipe) são requisitos de alta prioridade no sistema (Gross *et al.*, 2023). Em relação às perdas resultantes da adoção precipitada da arquitetura *Single-Page Application*, pode-se citar alguns exemplos.

2.3.1 Não é *RESTful* pois viola a restrição da interface uniforme

Apesar de uma API baseada em JSON com interface HTTP ser popularmente chamada de API REST, ela não pode ser considerada *RESTful*. Trata-se de um engano grave. O motivo pelo qual uma API JSON não é *RESTful* é o fato de não respeitar algumas restrições REST, principalmente a sub-restrição de HATEOAS (Seção 1.2.4) inclusa na restrição de interface uniforme (Seção 1.2.4). Para exemplificar este fato, basta analisar dois exemplos de respostas do servidor, uma em HTML no Programa 2.1 e outra em JSON no Programa 2.2.

Programa 2.1 Exemplo de resposta do servidor em HTML.

```
<html>

<body>
  <p>ID do usuário: 12</p>
  <p>Nome: Fernando</p>
  <p>Nacionalidade: Brasileiro</p>
  <nav>
    <p>Links:</p>
    <a href="/usuarios/12/amigos">Lista de amigos</a>
    <a href="/usuarios/12/favoritos">Favoritos</a>
    <a href="/usuarios/12/fotos">Fotos publicadas</a>
  </nav>
</body>

</html>
```

Programa 2.2 Exemplo de resposta do servidor em JSON.

```
{
  "id_usuario": 12,
  "usuario": {
    "pais": "BR",
    "nome": "Fernando"
  },
  "status": "ativo"
}
```

A diferença entre as duas respostas do servidor é que a primeira, em HTML, é *RESTful* e a outra, em JSON, não.

Análise da resposta em HTML

O navegador, ao receber a resposta, não tem conhecimento prévio do que é um usuário, um nome, uma nacionalidade etc, mas apenas sabe processar e exibir HTML. Sendo o navegador o cliente de hipermídia, não precisa saber nada sobre esse tipo de dado e as rotas associadas a ele na API, a não ser as informações que podem ser descobertas dentro da própria resposta, como as URLs e os controles de hipermídia.[14]

Se o estado do recurso for alterado no servidor de maneira que sejam alteradas as ações e controles disponíveis ao usuário no momento em que acessa o recurso, então a resposta em HTML devolvida pelo servidor também será alterada de modo a conter o novo conjunto de ações disponíveis. O navegador exibirá o novo HTML sem nenhum conhecimento prévio do significado das regras e entidades de negócio. Por exemplo, trocar o status do usuário de "ativo" para "inativo" pode acarretar na perda da função de visualização da lista de amigos. Portanto, o hipertexto é o motor do estado da aplicação, pois o próprio

[14] Este comportamento está em total conformidade com as sub-restrições de mensagens autodescritivas e de HATEOAS do estilo REST.

hipertexto é responsável por carregar dentro de si as informações necessárias vindas do servidor para que o cliente continue interagindo com o sistema.

Análise da resposta em JSON

Em contrapartida, na resposta em JSON, a mensagem não é autodescritiva e o cliente precisa saber como interpretar corretamente os campos para exibir a interface de usuário apropriada. Além disso, é necessário que o cliente saiba o conjunto de ações disponíveis com base em informações derivadas de outra fonte de informação externa à resposta, como uma documentação da API ou uma implementação interna (Gross *et al.*, 2023).

No caso concreto do Exemplo 2.2, o cliente precisa saber como interpretar o campo `status` para exibir os elementos corretos na interface do usuário, uma vez que esse tipo de informação não consta na resposta. Portanto, a resposta em JSON não é autodescritiva e não codifica o estado do recurso dentro de uma hipermídia, falhando assim na restrição de interface uniforme (Seção 1.2.4).

O hipertexto é necessário

O papel da hipermídia no HATEOAS é tão importante que o próprio Fielding, em sua tese, nomeia a subrestrição como "a restrição de hipermídia"[15] (Fielding, 2000). **O hipertexto é uma restrição**[16] , ou seja, se o motor do estado da aplicação não for orientado a hipertexto, então não pode ser considerado *RESTful*, muito menos uma API REST (Fielding, 2008).

Uma API baseada em uma interface HTTP não é necessariamente uma API REST. Se o motor do estado da aplicação não é orientado à hipertexto, então o sistema não pode ser considerado *RESTful*, portanto não há API REST. Segundo Fielding, "uma API REST deve ser acessada sem conhecimento prévio além da URI inicial (...), a partir desse ponto, todas as transições de estado da aplicação devem ser conduzidas pela seleção do cliente das escolhas fornecidas pelo servidor, que estão presentes nas representações recebidas ou implícitas pela manipulação dessas representações pelo usuário" (Fielding, 2008).

Como todas as respostas de uma API REST são autodescritivas e codificam o conjunto de ações disponíveis, não há necessidade de versionar a API, nem mesmo documentá-la. A qualquer mudança nos recursos ou nas representações, as respostas mudam, e isto é suficiente, pois o cliente não precisa de conhecimento prévio.

Remote Procedure Call

Uma API JSON estruturada da maneira descrita até então é, na verdade, uma API semelhante ao estilo RPC (*Remote Procedure Call*). O cliente e o servidor estão acoplados, pois o cliente precisa de conhecimento adicional sobre o recurso com o qual ele está

---

[15] A expressão original em inglês é: *the hypermedia constraint.*

[16] Quando Roy Fielding menciona hipertexto, quer dizer "a apresentação simultânea de informações e controles de forma que a informação se torne a possibilidade através da qual o usuário obtém escolhas e seleciona ações. Hipertexto não precisa ser necessariamente HTML em um navegador. Máquinas podem seguir links quando entendem o formato dos dados e os tipos de relacionamento."(Fielding, 2008)

trabalhando, e este conhecimento precisa ser obtido de outra fonte externa à resposta em JSON. O oposto daquilo proposto como REST originalmente por Roy Fielding. Afinal, a API JSON é somente uma API de dados (Seção 2.1.1), não utiliza hipermídia e não contém nas suas respostas o estado da aplicação, mas apenas a representação de um determinado recurso em formato JSON.

2.3.2 Duplicação do estado do servidor no cliente

Tendo em vista que a resposta da API em JSON não é uma hipermídia,[17] cabe ao código do lado do cliente decodificar a representação do recurso e convertê-la em HTML que será exibido dinamicamente na interface do usuário. Para formar o HTML correto, é necessário conhecer o estado da aplicação, isto é, o contexto atual do recurso indicado pela própria informação e pelas transições disponíveis ao cliente, baseadas na resposta do servidor. Porém, a API JSON não codifica o estado da aplicação. Apesar de os dados estarem contidos na resposta, as operações cabíveis em cima destes dados não estão. Portanto, o cliente deve gravar o estado do servidor para saber como atualizar corretamente a interface gráfica do usuário. Logo, **o código de atualização da interface do usuário necessita de ter conhecimento a respeito da estrutura interna e do significado dos dados recebidos do servidor.**

O gerenciamento de estados no cliente é algo difícil e complexo, existem diferentes formas de realizá-lo, usualmente utilizando alguma biblioteca de terceiros.[18] O estado da aplicação, portanto, é **duplicado** no lado do cliente, adicionando mais uma camada de complexidade que deve ser gerenciada por essas bibliotecas: **a sincronia do estado do cliente com o servidor.**

2.3.3 Acoplamento do cliente com o servidor

A principal vantagem da restrição de interface uniforme é o desacoplamento dos serviços que ela oferece ao cliente. Os navegadores web são capazes de interpretar a interface uniforme do HTML, seus elementos e seus controles. Uma vez que a própria representação descreve o estado da aplicação, o navegador é capaz de atualizar a interface do usuário sem nenhum conhecimento da estrutura interna do servidor.

Uma API JSON não respeita a restrição de interface uniforme do estilo REST e necessita de ter conhecimento a respeito da estrutura interna e do significado dos dados recebidos. Como resultado, o código em JavaScript do cliente e a API JSON oferecida pelo servidor têm um **acoplamento forte**,[19] pois, se o formato da resposta JSON mudar, o código do cliente quase certamente também precisará ser alterado, pois esta resposta só pode ser interpretada a partir do conhecimento privilegiado da estrutura interna do servidor.

Torna-se necessário que o cliente tenha conhecimento dos seguintes itens (Gross *et al.*, 2023):

---

[17] Portanto, ferindo a restrição REST de interface uniforme.

[18] Como *Redux*, *RxJS*, *Signals* etc.

[19] Os componentes estão altamente interligados e dependem fortemente uns dos outros para o bom funcionamento do sistema. A alteração em um módulo frequentemente exigirá a mudança em outro.

- Como os campos do objeto devolvido são estruturados e nomeados;
- Como eles se relacionam uns com os outros;
- Como atualizar os dados locais com os novos dados correspondentes;
- Como desenhar esses dados na interface gráfica do navegador;
- Quais são as ações adicionais que podem ser tomadas com esses dados;

Este conhecimento a respeito dos dados e do *endpoint* da API não é provido pela resposta HTTP, mas é fornecido por meio de algum canal paralelo além da resposta em si (Gross *et al.*, 2023). Naturalmente, o acoplamento dessas duas componentes dificulta a manutenção e evolução do sistema, e o torna menos flexível e mais propenso a erros, pois qualquer modificação que não for ajustada nas duas pontas pode causar efeitos colaterais indesejados.

O uso de uma interface como uma API deveria ser uma ferramenta para favorecer o desacoplamento, mas pode causar o efeito oposto se as mensagens não puderem ser interpretadas de forma independente. Colocar o estado do lado do cliente também reduz o controle do servidor sobre a consistência da aplicação, e se torna dependente da implementação correta da semântica entre várias versões do cliente.

2.3.4 Processo de inicialização inchado

Em um SPA, a interface gráfica é construída e atualizada pelo código em JavaScript no lado do cliente. Isto significa que o processo de renderização inicial da página é diferente do método tradicional.

1. O usuário digita a URL e o navegador dispara uma requisição HTTP de método `GET` para o servidor, solicitando o recurso[20];
2. O servidor devolve o arquivo `index.html`: um documento HTML quase vazio, mas que contém as *tags* de `<script>` com as referências aos arquivos que carregarão o JavaScript do cliente;
3. Ao receber o `index.html`, o navegador faz a requisição ao servidor do recurso `main.js`, referenciado na *tag* `<script>`;
4. O servidor responde à requisição com o arquivo `main.js`;
5. O cliente carrega os scripts[21] , e através dele dispara as demais requisições AJAX à API JSON solicitando os dados para preencher o conteúdo da página. Neste momento, costuma-se iniciar a exibição de algum indicador de carregamento (como um "esqueleto" da página ou *spinners*) até que o conteúdo esteja pronto;

[20] Para o esquema descrito, optou-se pela simplicidade no servidor, uma vez que o interesse é analisar a complexidade do cliente. Tipicamente, as requisições das etapas 1 e 2 seriam feitas para um CDN (*Content Delivery Network*) e as requisições da etapa 5 para a API JSON propriamente.

[21] Este carregamento consiste em executar o JavaScript no lado do cliente, após o HTML inicial ter sido carregado, para adicionar interatividade ao conteúdo estático, ligando os componentes interativos e aplicando os manipuladores de eventos. Este processo é chamado de **hidratação**.

6. O servidor recebe as requisições e solicita os recursos ao banco de dados;
7. O banco de dados devolve os recursos ao servidor;
8. O servidor processa os dados recebidos do banco, e gera sua representação em JSON;
9. O servidor devolve ao cliente os dados solicitados em formato JSON através da API;
10. O cliente processa os dados recebidos, cria a representação em HTML correspondente e atualiza a interface do usuário. Neste momento os indicadores de carregamento são substituídos pelo HTML do conteúdo restante;

Este processo está representado no diagrama da Figura 2.3.

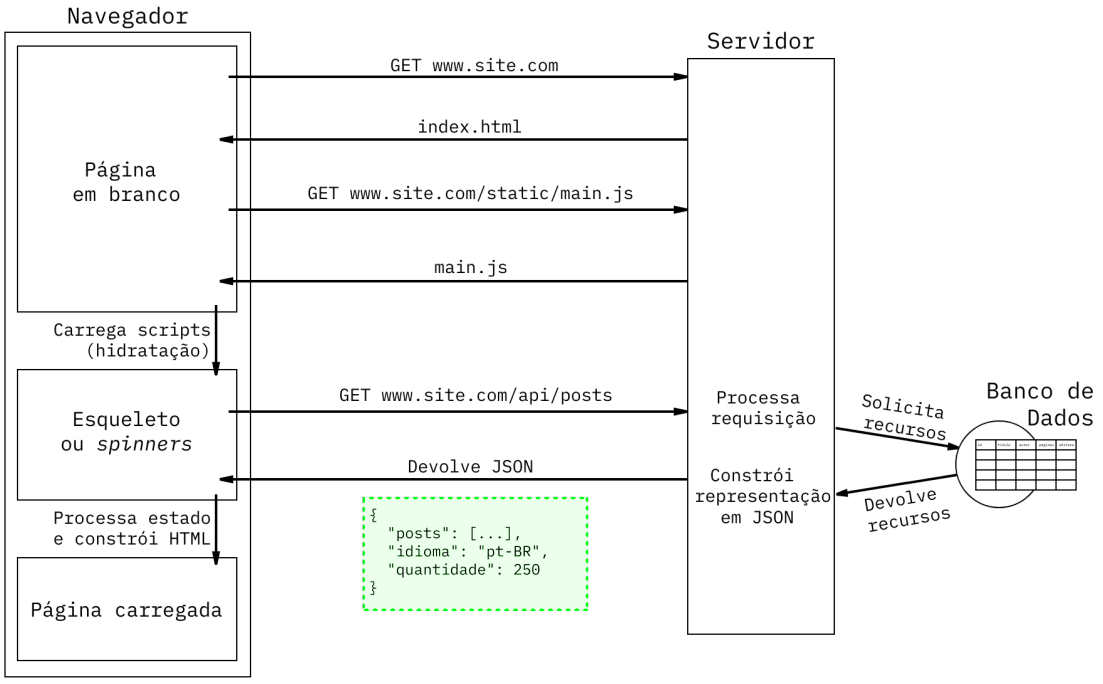


Figura 2.3: *Diagrama do processo de inicialização de uma aplicação SPA no cliente.*

Deste momento em diante, o cliente em JavaScript está totalmente carregado no navegador,[22] e a comunicação com o servidor se dará unicamente através da API JSON, para troca de dados somente. Isto significa que **toda a aplicação é carregada no lado do cliente no momento da inicialização**.[23]

Por um lado, a aplicação pode ser mais rápida ao navegar entre as páginas, uma vez que boa parte já foi carregada previamente. Por outro lado, **o tempo de carregamento inicial**

[22] Existem técnicas modernas de carregamento sob demanda (*lazy loading*) de scripts e componentes para aplicações do tipo SPA que não serão abordadas neste trabalho pois exigem técnicas complexas e arcabouços de código bastante específicos. Não há esta funcionalidade na maioria das aplicações do tipo SPA atualmente.

[23] Processo referido em inglês como: *initial full load* ou *eager loading*.

é muito maior em virtude do número de etapas e do volume de dados a ser transferido pela rede. Este processo é bastante ineficiente para a rede, e o tempo de carregamento e volume de dados aumenta conforme aplicação é estendida, independentemente de se o usuário for acessar uma ou todas as páginas disponíveis na aplicação, pois é necessário carregar todas as funcionalidades de antemão. O efeito deste fenômeno foi um aumento notável na quantidade de *scripts* transferidos somente para exibir a página inicial de um site, até mesmo aqueles com funcionalidades simples, em virtude do carregamento de muitos arquivos de JavaScript (bibliotecas, suas dependências e o código personalizado).

Como foi atestado por Nikita Prokopov, em seu artigo "O Inchaço do JavaScript em 2024" (*The JavaScript Bloat in 2024*) em que fez as métricas em sites populares, até mesmo nos sites cujo conteúdo é majoritariamente estático, o volume de scripts transferidos pode chegar a 16 megabytes apenas para o carregamento da página inicial (Prokopov, 2024). Ou seja, além de ser um processo complexo, o carregamento inicial de um SPA é desfavorável na perspectiva da eficiência de rede, impactando negativamente o desempenho percebido pelo usuário.

2.3.5 Processo complexo de compilação e empacotamento

Como o carregamento inicial da aplicação pode ser muito prejudicado pelo volume de arquivos a ser carregado no cliente, torna-se necessário um processo de compilação para otimizá-los. O processo de compilação[24] transforma o código escrito pelos desenvolvedores em algo que pode ser eficientemente executado por um navegador, com desempenho e confiabilidade, sem perder a capacidade de manutenção. Esse processo costuma ocorrer ao longo das seguintes etapas ao utilizar as bibliotecas mais populares de SPA:

1. **Transpilação:** converte a sintaxe utilizada para código em JavaScript compatível com a maioria dos navegadores. Geralmente se utiliza JSX (*JavaScript Syntax Extension*) e JavaScript moderno para escrever o código a ser transpilado.
2. **Minificação:** reduz o tamanho dos arquivos de HTML, CSS e JS (*JavaScript*) removendo espaços em branco e comentários, reduzindo os nomes de variáveis e de funções. Gera uma versão dos arquivos que não é legível para humanos, mas ocupa bem menos espaço, reduzindo a quantidade de dados transferidos na rede.
3. **Empacotamento**[25]**:** combina o código dos arquivos (código próprio e dependências de terceiros) em um número menor de arquivos otimizados. Este processo reduz o número de arquivos, portanto reduz o número de requisições HTTP necessárias para o carregamento inicial da aplicação.
4. **Separação de código:** divide os arquivos gerados na etapa anterior em fatias.[26] Em vez de carregar toda a aplicação de uma vez, o navegador pode carregar somente o código necessário para a página ou funcionalidade atual.[27]

[24] Por compilação, entende-se o processo referido em inglês como *build*, presente nas aplicações modernas de JavaScript.

[25] Do inglês: *bundling*.

[26] Do inglês: *chunks*.

[27] Não está presente no SPA puro, mas em alternativas mistas com arcabouços de SSR (*Server-Side Rendering*).

5. **Otimização de recursos:** comprime arquivos de imagens, otimiza os arquivos de fontes e insere pequenos recursos diretamente no JavaScript, como imagens vetoriais e estilos.
6. **Variáveis de ambiente:** injeta as variáveis de ambiente diretamente no código de acordo com o respectivo cenário: desenvolvimento, homologação, produção etc.
7. **Remoção de código inutilizado**[28]**:** remove código inutilizado[29] do pacote. Esta etapa é especialmente importante ao utilizar dependências de terceiros, que podem importar uma longa lista de funções ou módulos que não serão utilizados na aplicação final.
8. **Geração de arquivos estáticos:** o arquivo `index.html` é gerado com referência aos pacotes de JS e CSS manipulados nas etapas anteriores, e os arquivos compilados são movidos para um diretório separado, prontos para o despacho.[30]

Esse é mais um processo complexo inserido no modelo SPA. Logo é possível perceber que, uma vez inserido o processo de compilação, não é mais possível realizar atualizações pontuais na aplicação. Em uma aplicação tradicional em HTML, CSS e JS, caso seja necessário atualizar alguma parte do site, bastaria atualizar os arquivos no servidor relativos à atualização. Com o processo de compilação de um SPA, torna-se necessário reproduzir o processo de compilação e despacho de **toda** a aplicação, independentemente do escopo ou do tamanho das modificações realizadas. Ou seja, a cada simples mudança em uma função de JavaScript, ou na cor de um botão, torna-se necessário substituir todos os arquivos do código do cliente por novas versões compiladas, passando novamente por todas as etapas acima, como pode ser visto na Figura 2.4.

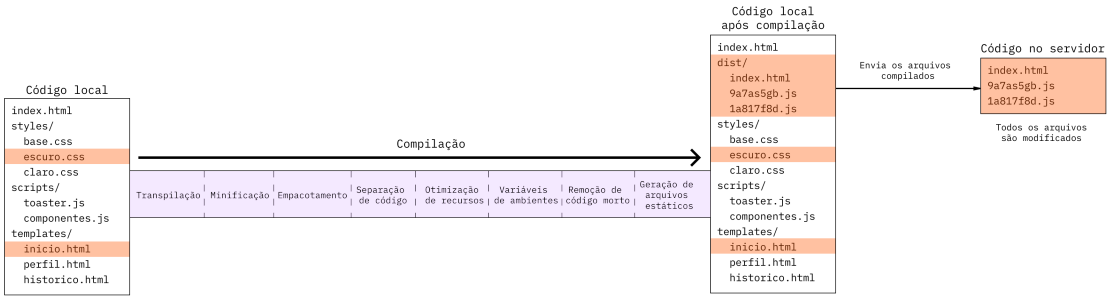


Figura 2.4: *Diagrama do processo de compilação e despacho de uma aplicação SPA no cliente.*

2.3.6 Alto nível de abstração para ocultar a complexidade

Pôde-se perceber que uma aplicação em SPA exige o controle de operações complexas, muitas vezes assíncronas, e que exigem uma atenção redobrada a certos conceitos. Para tornar a manutenção viável e aumentar a produtividade de desenvolvimento, a grande maioria dos desenvolvedores implementa este modelo utilizando uma biblioteca ou arcabouço

[28] Conhecido em inglês como: *tree shaking*.

[29] Do inglês: *dead code*.

[30] Do inglês: *deployment*.

desenvolvida especificamente para gerenciar esses processos. Todavia, essas bibliotecas são caracterizadas por seu alto nível de abstração.

Porém, este nível de abstração não é necessário para realizar tarefas simples e triviais da Web, como listar artigos de um blog, adicionar linhas a uma tabela ou submeter formulários de cadastro. Além disso, o aumento do nível de abstração e a criação de novos objetos leva a um distanciamento cada vez maior dos fundamentos para os quais aquele código será convertido: HTML, CSS e JS compatíveis com o navegador. A alta abstração dos *Single-Page Applications* acaba por ocultar a sua enorme complexidade. Este ocultamento da complexidade por interfaces simplificadas pode dificultar a depuração e compreensão do comportamento interno da aplicação, especialmente em aplicativos grandes e complexos.

A subseções abaixo descrevem exemplos mais populares destas abstrações.

Manipulação abstrata do DOM

Em um SPA, a manipulação do DOM (*Document Object Model*) é feita de forma mais abstrata. O DOM não é manipulado diretamente ao interagir com os elementos HTML gerados pelo código em JavaScript, mas o desenvolvedor interage com os componentes e estados de forma declarativa. Isto é feito através de um DOM virtual[31],[32] que consiste em uma representação em memória da estrutura real do DOM. O DOM virtual permite que o desenvolvedor escreva a interface como ela deve se parecer de acordo com o estado, e a biblioteca cuidará de forma automática e eficiente da atualização do DOM real, **abstraindo a lógica de como e quando atualizar a interface com base nas mudanças de estado.**

Controle de navegação abstraído

Uma vez que a navegação não ocorre através do comportamento implementado no navegador (como descrito na Seção 2.1), este controle também precisa ser abstraído para permitir a navegação sem o recarregamento completo da página. O "roteamento" é controlado por mudanças de estado e URL sem a necessidade de recarregar a página, manipulando APIs dos navegadores e atualizando o DOM virtual. Na perspectiva do desenvolvedor, é como se estivesse de fato lidando com uma página separada, mas, concretamente, ocorre uma série de operações gerenciadas pela biblioteca para simular a navegação entre diferentes páginas.

Gerenciamento de estado abstrato

A abstração do gerenciamento de estados é um dos pontos principais para a adoção de uma biblioteca reativa ao desenvolver um SPA. Os estados são tratados como objetos reativos, que podem ser declarados como variáveis comuns que, sob determinado contexto, atuam como variáveis reativas. Estas biliotecas abstraem a necessidade de manipulação manual do estado na aplicação, e mitigam os erros ao omitir a implementação concreta e oferecer ao usuário uma sintaxe simplificada.

---

[31] Do inglês: *virtual DOM*.

[32] Duas bibliotecas muito populares que utilizam a estratégia de DOM virtual são React e Vue.

Componentização

Através da abstração da interface em componentes reutilizáveis, os desenvolvedores podem se concentrar mais na lógica de negócios e na interface do usuário, sem se preocupar com a estrutura interna do DOM. Cada componente gerencia seu próprio estado, encapsula estilo, lógica e estrutura em uma única unidade (2.2) e se comunica com outros componentes de maneira modular.

2.3.7 Enormes cadeias de dependências

Em virtude do aumento da dificuldade de compreensão do comportamento da biblioteca causado pelo seu alto nível de abstração, é comum que software do tipo SPA possua um grande número de dependências. Operações que no modelo tradicional são simples, como submeter formulários e navegar entre páginas através de *links* tornaram-se complexas a partir do momento em que não há hipermídia como motor do estado da aplicação, mas o estado precisa ser duplicado no cliente e constantemente sincronizado com o servidor (Seção 2.3.2).

Portanto, é comum adotar bibliotecas complexas e abstratas, implementadas para o arcabouço específico (portanto, fortemente acopladas) para gerenciar aspectos específicos da aplicação. Exemplos comuns de biblioteca servem para roteamento (rotas e navegação sem troca de página), para compilação, para gerenciamento global de estados (modelo centralizado e reativo para simplificar o fluxo de dados), para gerenciar as requisições da API, para controlar o estado de formulários etc. Ou seja, não basta ter de aprender a biblioteca de JavaScript, é preciso instruir-se nas bibliotecas auxiliares que a utilizam como base para conseguir implementar uma aplicação sofisticada.

Um grande número de dependências em um projeto aumenta significativamente a complexidade e os riscos associados à manutenção. Cada dependência introduz um ponto potencial de falha, seja por bugs, incompatibilidades com atualizações futuras ou vunerabilidades de segurança. Além disso, tornam o projeto ainda mais difícil de configurar e executar. Quando muitas funcionalidades essenciais dependem de bibliotecas de terceiros, eventuais mudanças ou descontinuações em uma delas podem exigir reescritas complexas, causando a perda de controle sobre o código do projeto.

2.3.8 A curva de aprendizado é muito maior

Por mais que as abstrações se proponham a tornar a experiência de desenvolvimento mais agradável e enxuta, isto só ocorre a longo prazo. O desenvolvedor que deseja construir um projeto em SPA não apenas precisa saber todos os fundamentos da Web, o funcionamento do navegador e os blocos fundamentais (HTML, CSS e JS), como ainda é preciso aprender uma biblioteca ou arcabouço destinados ao desenvolvimento deste tipo de aplicação, que abstraem os itens anteriores (Seção 2.3.6). E, uma vez tendo aprendido essas bibliotecas, será preciso aprender uma série de dependências e outras bibliotecas para a implementação de funcionalidades específicas. A simplicidade só vem a longo prazo, com o costume e o domínio dessas bibliotecas, suas abstrações e suas dependências em conjunto. Para os novos desenvolvedores, a curva de aprendizado é muito maior, e isto pode sobrecarregá-lo.

2.3.9 A adoção de JavaScript no código do servidor

O desenvolvedor terminará com uma grande base de código escrita em JavaScript para executar no lado do cliente caso escolha por adotar o modelo SPA para sua aplicação. Naturalmente, diante da falta de uma interface uniforme e da grande ruptura na localidade do código ao dispôr de duas bases distintas (de *back-end* e *front-end*), a única forma de mitigar a duplicação de lógica causada pelo acoplamento forte (Seção 2.3.3) é através do uso da mesma linguagem de programação nas duas pontas, para permitir a reutilização das estruturas.

Porém, o JavaScript é a única linguagem que é executada diretamente no navegador do usuário. Os navegadores não são capazes de executar Python, Java, Ruby etc, mas apenas JavaScript. Como não é possível que o desenvolvedor individualmente altere esta realidade, a única opção restante é que adote JavaScript como linguagem de programação da lógica do servidor também. **Há uma pressão para adotar JavaScript como linguagem de programação do código do servidor.** Por conta disso, as outras linguagens de programação vêm perdendo espaço para o "monopólio" JavaScript, pois a competição é desigual. Não há outra opção para evitar a duplicação de código.

Utilizar JS no servidor permite compartilhar lógica e estrutura de dados entre as duas bases de código, a curva de aprendizado (Seção 2.3.8) será levemente reduzida, uma vez bastará dominar uma única linguagem para trabalhar nas duas pontas da aplicação, e o complexo sistema de compilação (Seção 2.3.5) e gerenciamento de dependências (Seção 2.3.7) poderá ser reutilizado para as duas bases. Porém, paga-se o preço de perder a flexibilidade de optar por linguagens mais adequadas para certos procedimentos no servidor, ou até mesmo mais performáticas. Ao aderir ao SPA, ou abdica-se de uma linguagem de programação diferente no servidor ou rende-se ao retrabalho na duplicação de lógica entre as pontas. Vale notar que, uma vez utilizando JavaScript nas duas pontas, a adoção de JSON como notação intermediária da comunicação entre elas também é pressionada, em virtude da alta correlação entre ambos.

2.3.10 JavaScript é tratado como linguagem de propósito geral

A linguagem JavaScript foi desenvolvida por Brendan Eich em 1995, durante seu trabalho na empresa Netscape. Eich criou a linguagem em um período de dez dias, com o objetivo de possibilitar que desenvolvedores da web implementassem pequenas interações nas páginas sem a necessidade de plugins ou softwares adicionais, sendo o próprio navegador responsável pela interpretação do script. Dessa forma, o JavaScript foi projetado como uma linguagem de *scripting*, para adicionar interações dinâmicas de forma pontual nas páginas web. Com o tempo, no entanto, passou a ser utilizado também no lado do servidor, expandindo seu papel para o de uma linguagem de propósito geral.

Ocorre que o uso de JavaScript como uma linguagem de propósito geral traz desafios uma vez que a linguagem foi originalmente concebida para tarefas rápidas e pequenas. É uma linguagem interpretada, de alto nível, com tipagem dinâmica e fraca, e que adota uma abordagem multiparadigma, o que é ótimo para *scripting*, mas nem sempre desejável para um servidor robusto.

2.3.11 HTML não é tratado como hipermídia

Em um SPA, não apenas o JavaScript deixa de ser encarado como uma linguagem de *scripting* para ser visto como uma linguagem de propósito geral, mas o HTML também deixa de ser usado como uma hipermídia e se torna uma mera linguagem de descrição visual. O HTML não é mais o motor do estado da aplicação e seus controles de hipermídia são inutilizados, isto é, nenhum elemento interage com o servidor através de seu mecanismo nativo de hipermídia, mas apenas incita algumas interações locais com o script na memória do navegador, que então é sincronizado com o servidor utilizando APIs JSON de dados. O HTML foi reduzido a ser uma linguagem de definição de interface do usuário (Gross *et al.*, 2023).

Ainda é de se notar que a sintaxe de HTML utilizada por algumas bibliotecas é ilegal e vai contra as especificações, como o uso de `className` para definir as classes do elemento em React, ou a possibilidade de criar elementos "auto-fechantes"[33] que não existem no HTML, como `<br />` ou `<img />`.[34] (WHATWG, 2024). Portanto, a arquitetura Single-Page Application renuncia à arquitetura de hipermídia. Abandona as vantagens existentes da arquitetura RESTful da web e as funcionalidades incorporadas no HTML e nos seus controles de hipermídia, a favor de comportamentos conduzidos pelo JavaScript.

2.3.12 Navegador não é tratado como cliente de hipermídia

Por fim, com o passar do tempo o navegador também deixou de ser encarado com seu propósito original: de ser um cliente de hipermídia;[35]; e passou a ser visto como um mero interpretador de JavaScript, que é o responsável por executar e gerenciar os recursos da aplicação web que passa a ser desenvolvida inteiramente em JavaScript. Ser um cliente de hipermídia é visto apenas como um aspecto legado de seu comportamento, para justificar o fato de se utilizar HTML como uma linguagem de descrição visual (e não uma solução mais adequada para esta função).

[33] Do inglês: *self-closing tags*.

[34] Elementos auto-fechantes não existem no HTML, são uma sintaxe ilegal muito presentes nos SPAs. O que há na especificação do HTML são elementos vazios (*void elements*), que são elementos que não podem possuir nenhum elemento filho, como `area`, `base`, `br`, `col`, `embed`, `hr`, `img`, `input`, `link`, `meta`, `source`, `track` e `wbr`.

[35] O navegador web é o exemplo canônico de um cliente de hipermídia.

Capítulo 3

A estrutura hipermídia da Web

Dado que a abordagem SPA apresenta as calamidades descritas no capítulo anterior, torna-se conveniente a busca por alternativas viáveis. Se os problemas descritos não eram enfrentados pelos desenvolvedores no período anterior ao surgimento do SPA, é pertinente resgatar as técnicas tradicionais a fim de garantir o pleno cumprimento das restrições do estilo arquitetural REST.

Este capítulo propõe o uso de sistemas de hipermídia como solução, juntamente com a técnica de transclusão de documentos executada no cliente, em vista de resgatar as características da Web em sua fase inicial (em que as especificações eram rigorosamente respeitadas) acompanhadas de técnicas modernas que não estavam disponíveis 20 anos atrás.

3.1 Sistema de hipermídia

Segundo Carson Gross, Adam Stepinski, e Deniz Akşimşek em seu livro *"Hypermedia Systems"* ("Sistemas de Hipermídia") (Gross *et al.*, 2023), um sistema de hipermídia é composto por quatro componentes:

1. Hipermídia;
2. Protocolo de hipermídia;
3. Servidor de hipermídia (que serve uma API de hipermídia ao cliente);
4. Cliente de hipermídia;

3.1.1 Hipermídia

O prefixo *hiper* deriva do grego, que significa "além" ou "sobre", indicando que a hipermídia transcende as mídias comuns, como revistas e jornais, que são consumidas de forma passiva.

Definição 1: Hipermídia
Hipermídia é um tipo de mídia que pode ser consumida de forma não linear, incluindo

ramificações de uma localização na mídia para outra, por meio de controles de hipermídia incorporados na mídia (Gross et al., 2023).

O exemplo canônico[1] de hipermídia é o **hipertexto**. Uma hipermídia do tipo texto é chamada de hipertexto. Portanto, pode-se dizer que o hipertexto é uma subcategoria de hipermídia. É "uma peça de escrita não sequencial" (Nelson, 1977).[2]

Definição 2: Controle de hipermídia
Um controle de hipermídia é um elemento dentro de uma hipermídia que descreve (ou controla) algum tipo de interação, geralmente com um servidor remoto, codificando informações sobre essa interação de forma direta e completa dentro de si mesmo (Gross et al., 2023).

As **hiperligações**[3] são o exemplo canônico de controles de hipermídia. Hiperligações fazem referência a outro documento ou a um elemento específico dentro do mesmo documento.

Em HTML, as hiperligações são representadas pelo elemento de âncora `<a>`. Ao clicarmos no texto de um elemento de âncora definido em HTML, o cliente será direcionado para um novo conteúdo, que pode ser outro documento HTML ou um outro elemento localizado na mesma página.

A hipermídia da Web: o HTML

Atualmente, o HTML *(HyperText Markup Language)* é a hipermídia mais utilizada de todos os tempos. Sendo o HTML uma hipermídia, espera-se encontrar em sua estrutura a presença de controles de hipermídia.

O HTML possui dois controles de hipermídia:

- O elemento de âncora: `<a>`[4]
- O elemento de formulário: `<form>`[5][6]

O elemento de âncora provê o principal mecanismo para navegar pela Web, através do atributo `href` que faz referência a *hypertext reference* (referência de hipertexto). O atributo `href` especifica uma referência a outro documento ou o fragmento de outro documento. É este atributo, portanto, que torna o elemento de âncora um controle de hipermídia, que permite navegar de documento a documento ou de recurso a recurso.

Enquanto os elementos de âncora provisionam a navegação entre documentos ou recursos, os formulários permitirão operações de modificação desses recursos. O elemento de formulário possui o atributo `method` que especifica qual deve ser o método da requisição

[1] Por exemplo canônico, entende-se um dos exemplos mais representativos e clássicos de determinado conceito, que expressa de maneira clara e eficaz como o conceito funciona.

[2] Este conceito foi criado pelo filósofo e sociólogo estadunidense Ted Nelson nos anos 1960. Na verdade, foi o termo "hipertexto" que unido à palavra "multimídia" deu origem ao termo "hipermídia".

[3] Do inglês: *hyperlinks*.

[4] Do inglês: *anchor tag*.

[5] Do inglês: *form tag*.

[6] Introduzido a partir do HTML 2.0 (Berners-Lee e Connoly, 1995)

em HTTP a ser disparada pelo navegador após a submissão do formulário.[7] Também possui o atributo `action`, que indica para onde esta requisição deve ser enviada.

Dentro do elemento do formulário serão inclusos outros elementos de entrada do usuário (como `input`, `select`, `button` entre outros), que podem ou não afetar o conteúdo enviado na requisição HTTP enviada pelo navegador ao servidor. O formulário do HTML, além de ser um controle de hipermídia, foi o que permitiu que a Web deixasse de ser apenas um sistema orientado a documentos para se tornar uma arquitetura de aplicações. O formulário permite que o cliente atualize o estado dos recursos no servidor, e, portanto, torna as aplicações orientadas a hipermídia possíveis (Gross *et al.*, 2023). Esse elementos são as únicas duas maneiras nativas de um usuário interagir com um servidor em HTML puro.

3.1.2 Protocolo de hipermídia

Um controle de hipermídia é responsável por controlar algum tipo de interação com o servidor. Esta interação é realizada por meio de uma comunicação cuja sintaxe e semântica é convencionada através de um protocolo, para servir de linguagem comum entre as duas pontas.

Definição 3: Protocolo de hipermídia
Um protocolo de hipermídia consiste nas regras de sintaxe, de semântica e de sincronização que governam a comunicação entre dois sistemas que trocam hipermídia entre si.

O protocolo de hipermídia da Web: o HTTP

O exemplo canônico de um protocolo de hipermídia é o HTTP, o protocolo utilizado para transferir a hipermídia do HTML entre servidores de hipermídia e clientes de hipermídia, sendo, portanto, o elemento que conecta os outros componentes do sistema. O HTTP é um protocolo simples por *design*, estruturado na troca de requisições por respostas.

Para a requisição enviada pelo cliente, o protocolo provê diferentes **métodos** indicando a natureza da requisição, **cabeçalhos** que contêm metadados que podem ser utilizados pelo servidor para determinar como responder corretamente à requisição e, opcionalmente, um conjunto de **dados** (*payload*). A resposta do protocolo HTTP devolvida pelo servidor contém um **código de retorno** entre 100 e 599, **cabeçalhos** contendo metadados para orientar o cliente a exibir a representação dos recursos e, por fim, a **representação dos recursos** solicitados em formato HTML.

3.1.3 Servidor de hipermídia

Qualquer servidor que pode responder a uma requisição HTTP com uma resposta HTTP é considerado um servidor de hipermídia. Estas interações podem ser definidas numa estrutura de **API de hipermídia**, que será responsável por aproveitar os métodos, os cabeçalhos e os códigos de retorno HTTP para criar uma comunicação robusta e coerente, que utiliza esses elementos da maneira para a qual foram projetados.

[7] A especificação atual do HTML prevê apenas `GET` ou `POST` como valores aceitos pelo atributo `method`, mesmo o HTTP possuindo uma série de outros métodos (WHATWG, 2024).

3.1.4 Cliente de hipermídia

Um cliente de hipermídia é um software capaz de interpretar corretamente uma hipermídia e seus controles de hipermídia, para exibí-los corretamente na interface gráfica ao encontrá-los em uma resposta do servidor. A exibição correta destes elementos é fundamental para que todo o sistema de hipermídia funcione com uma união harmônica. Sem um cliente capaz de realizar a exibição correta, os controles de hipermídia e a API de hipermídia perdem sua utilidade e todo o sistema colapsa.[8]

Portanto, a implementação do cliente de hipermídia é fundamental para tirar proveito da interface uniforme do REST, que deve esperar por uma hipermídia que generaliza a exibição do recurso e não um formato fixo que exige um processamento especial. O mesmo cliente pode servir para os mais diversos tipos de servidores e aplicações, desde que utilizem a mesma interface uniforme.

Os clientes de hipermídia da Web: os navegadores

O exemplo canônico de um cliente de hipermídia é o navegador web, que é capaz de interpretar HTML e exibí-lo ao usuário de maneira adequada para a interação com o servidor.

3.2 A Web é um sistema de hipermídia

A Rede Mundial de Computadores (WWW) pode ser compreendida, a partir da análise de seus componentes, como um **sistema de hipermídia distribuído** amplamente consolidado(Gross *et al.*, 2023).

Na Web, o HTML serve de base para a representação de hipermídia e o protocolo HTTP regula a comunicação entre os clientes e os servidores através desta hipermídia. Os navegadores modernos desempenham o papel de clientes de hipermídia sofisticados e eficientes, e os servidores de hipermídia se encontram distribuídos globalmente, abrangendo múltiplos domínios organizacionais.

Esta estrutura já consolidada da Web como sistema de hipermídia oferece uma base sólida sobre a qual se pode construir aplicações. Se esta arquitetura for considerada, respeitando a finalidade dos componentes e suas especificações, **pode-se tirar proveito destas estruturas para maior eficiência no desenvolvimento de aplicações em seu ecossistema.**

A Web não é um ambiente de arquitetura arbitrária, mas sim um ecossistema que impõe diretrizes claras para garantir sua coerência. A arquitetura própria da Web simplifica o processo de integração. Se essas diretrizes não forem respeitadas, pode se tornar necessário desenvolver soluções redundantes para problemas já resolvidos, capazes de gerar uma série de consequências semelhantes às descritas na Seção 2.3 ao ignorar o sistema de hipermídia inerente.

[8] Este é um dos motivos pelos quais os navegadores web possuem implementações robustas.

Dessa forma, ao desenvolver uma aplicação Web destinada à execução em navegadores, não é necessário construir os componentes do sistema de hipermídia a partir do nada, mas concentrar os esforços em construir um servidor robusto e compatível com a arquitetura de uma aplicação orientada a hipermídia, seguindo com rigor as especificações dos componentes da WWW para tirar proveito de suas características estruturais intrínsecas, especialmente a arquitetura REST.

3.3 Transclusão

Todavia, a estrutura descrita até então já estava presente nos primórdios da Web, antes do surgimento do modelo SPA. Apenas a retomada destes princípios não é capaz de substituir o comportamento dinâmico dos elementos da página em uma aplicação de página única. Torna-se necessário a adoção de uma técnica adicional, a qual se dá o nome de transclusão.

Definição 4: Transclusão
A técnica de transclusão consiste na inclusão de parte ou de todo um documento eletrônico em um ou mais outros documentos, resultando em um único documento composto por partes reunidas dinamicamente a partir de fontes separadas.

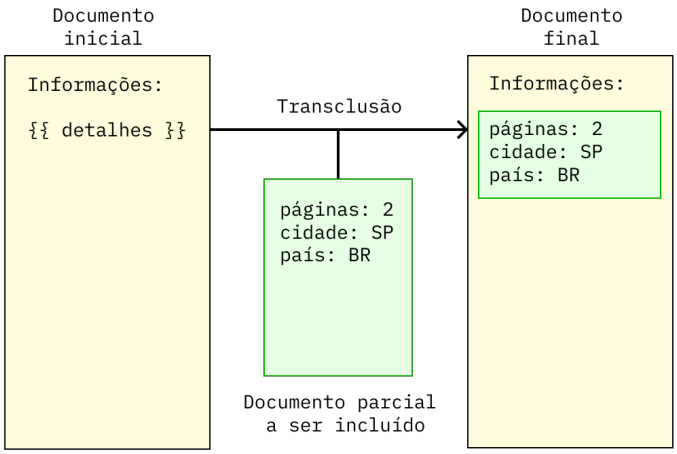


Figura 3.1: *Diagrama demonstrando o funcionamento do processo de transclusão.*

A composição de documentos HTML através do método de transclusão também é muito antiga no desenvolvimento Web, e sempre esteve presente nos motores de *templates*[9] para gerar os documentos devolvidos pelo servidor. Um motor de *templates* processa os arquivos com documentos parciais de HTML para gerar um documento único, que, ao ser recebido pelo navegador, substituiria completamente a página exibida pelo cliente (navegação embutida, descrita no início da Seção 2.1).

Para realizar a substituição dinâmica de parte da página de forma semelhante ao SPA, é necessário manipular o DOM (*Document Object Model*). Portanto, **a transclusão deve ocorrer do lado do cliente**, e não apenas no servidor, pois não basta a mera atualização

[9] Do inglês: *template engines*.

de um documento de texto em HTML, mas a efetiva substituição na representação deste documento na memória no navegador.

Porém, a funcionalidade de transclusão não está implementada por padrão nos navegadores web, não há suporte para outro tipo de atualização a não ser o escrito no início da Seção 2.1. A única forma de obter este comportamento no cliente de hipermídia, portanto, é através da extensão da funcionalidade do cliente por meio do *download* e da execução de *scripts*.

3.3.1 A diferença entre transclusão e o modelo SPA

A princípio, pode parecer que a sugestão de adotar a transclusão no lado do cliente não seja tão diferente do uso de uma biblioteca SPA. Se as duas abordagens fossem equivalentes, seria mais adequado adotar uma biblioteca robusta e confiável de *Single-Page Application* em vez de substituir seu comportamento por *scripts* escritos a partir do zero, pois além de terminar com todas as calamidades mencionadas na Seção 2.3, mas sem a necessidade de dar manutenção à lógica de gerenciamento de estados (que seria gerenciada por uma biblioteca de terceiros).

Porém, apesar de ambas envolverem a execução de JavaScript no navegador para sobrescrever o comportamento de atualização da interface gráfica do cliente, há uma diferença notável entre as duas abordagens. A técnica de transclusão ocorre através do envio de HTML do servidor para que a substituição ocorra no cliente e para que a tranclusão inclua este documento (ou parte dele) no HTML sendo exibido ao usuário naquele momento (representado na memória através do DOM).

Uma vez que é enviado HTML, e não JSON, pelo servidor, a restrição de HATEOAS (Seção 1.2.4) é respeitada, e a hipermídia atua como motor do estado da aplicação. Como o estado está codificado na representação, não é mais necessário que o cliente processe, armazene e sincronize o estado duplicado do servidor, portanto toda a lógica necessária a essas etapas é descartada.

Não sendo necessário realizar a gerência de estado no cliente, não é necessário que o cliente de hipermídia tenha conhecimento da estrutura interna do servidor, portanto é possível eliminar o acoplamento forte (Seção 2.3.3), uma vez que as mensagens também serão autodescritivas. Em outras palavras, a estratégia de transclusão no cliente a partir do HTML enviado pelo servidor consiste em uma comunicação de interface uniforme (Seção 1.2.4), e que, portanto, **é *RESTful,* diferentemente de um SPA**.

Na presença da interface uniforme, temos a garantia de que as mensagens são autodescritivas, e, sendo em formato HTML, o próprio navegador é capaz de interpretar e converter na representação esperada em memória. Isto permite que, diferentemente do SPA, os scripts relacionados à atualização da interface gráfica sejam generalizados. Portanto, com uma biblioteca que implemente a transclusão no lado do cliente, **não é necessário implementar lógica específica ao domínio para a atualização da interface**, mas apenas diretivas relacionadas aos elementos do HTML em si.

3.3.2 Exemplo de transclusão: adição de item na tabela com notificação flutuante

Na Figura 3.2 é possível visualizar um exemplo do funcionamento da transclusão ocorrendo no cliente de hipermídia para adicionar elementos dinâmicos na interface. Os programas e o diagrama abaixo assumem uma biblioteca de tranclusão fictícia, para simplicidade.

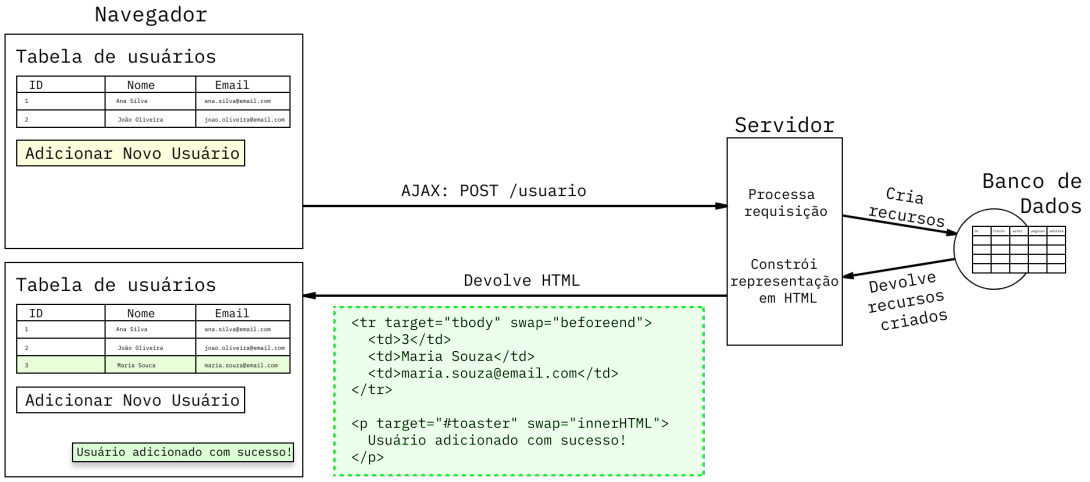


Figura 3.2: *Exemplo de transclusão: adição de item na tabela com notificação flutuante.*

1. A interface gráfica do usuário se encontra no estado inicial descrito no Programa 3.1;
2. Após preencher o formulário, o usuário clica no botão e envia uma requisição HTTP de método `POST` para a rota `/usuario`, indicando o pedido de inserção de um novo usuário;
3. O servidor recebe a requisição HTTP e cria o registro do novo usuário no banco de dados;
4. O servidor gera o HTML e responde à requisição com uma resposta HTTP contendo o código HTML do Programa 3.2;
5. O cliente recebe a resposta da requisição, que é interceptada pela biblioteca `transclusion.js`;
6. A biblioteca de transclusão analisa o HTML da resposta em busca dos atributos `swap` e `target`[10];
7. A biblioteca de transclusão realiza as substituições necessárias, inserindo os elementos HTML da resposta nos respectivos elementos da página descritos em `target` com a estratégia definida em `swap`;
8. A interface gráfica do usuário é atualizada para o estado final descrito no Programa 3.3.

[10] Para maior simplicidade nesta representação fictícia, a sintaxe utilizada para o atributo `swap` é a mesma da interface `insertAdjacentHTML` do JavaScript, e para o atributo `target` utilizou-se a mesma sintaxe de seletores de CSS.

Programa 3.1 Exemplo de transclusão: HTML inicial

```
<!DOCTYPE html>
<html lang="pt-br">

<head>
  <title>Tabela de Usuários</title>
  <script src="transclusao.js"></script>
</head>

<body>
  <h1>Tabela de Usuários</h1>
  <table>
    <thead>
      <tr>
        <th>ID</th>
        <th>Nome</th>
        <th>Email</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>1</td>
        <td>Ana Silva</td>
        <td>ana.silva@email.com</td>
      </tr>
      <tr>
        <td>2</td>
        <td>João Oliveira</td>
        <td>joao.oliveira@email.com</td>
      </tr>
    </tbody>
  </table>
  <form action="/usuario" method="post">
    <input type="text" name="nome" placeholder="Nome">
    <input type="email" name="email" placeholder="Email">
    <button type="submit">Adicionar Novo Usuário</button>
  </form>

  <!-- Notificações flutuantes -->
  <div id="toaster"></div>
</body>

</html>
```

Programa 3.2 Exemplo de transclusão: Resposta do servidor

```
<tr target="tbody" swap="beforeend">
  <td>3</td>
  <td>Maria Souza</td>
  <td>maria.souza@email.com</td>
</tr>
```

cont ⟶

⟶ *cont*

```
6
7   <p target="#toaster" swap="innerHTML">
8     Usuário adicionado com sucesso!
9   </p>
```

Programa 3.3 Exemplo de transclusão: HTML final

```
1   <!DOCTYPE html>
2   <html lang="pt-br">
3
4   <head>
5     <title>Tabela de Usuários</title>
6     <script src="transclusao.js"></script>
7   </head>
8
9   <body>
10    <h1>Tabela de Usuários</h1>
11    <table>
12      <thead>
13        <tr>
14          <th>ID</th>
15          <th>Nome</th>
16          <th>Email</th>
17        </tr>
18      </thead>
19      <tbody>
20        <tr>
21          <td>1</td>
22          <td>Ana Silva</td>
23          <td>ana.silva@email.com</td>
24        </tr>
25        <tr>
26          <td>2</td>
27          <td>João Oliveira</td>
28          <td>joao.oliveira@email.com</td>
29        </tr>
30        <tr>
31          <td>3</td>
32          <td>Maria Souza</td>
33          <td>maria.souza@email.com</td>
34        </tr>
35      </tbody>
36    </table>
37    <form action="/usuario" method="post">
38      <input type="text" name="nome" placeholder="Nome">
39      <input type="email" name="email" placeholder="Email">
40      <button type="submit">Adicionar Novo Usuário</button>
41    </form>
42
43    <!-- Notificações flutuantes -->
44    <div id="toaster">
```

cont ⟶

```
    ⟶ cont
45      <p>Usuário adicionado com sucesso!</p>
46    </div>
47  </body>
48
49  </html>
```

3.4 Limitações do HTML como hipertexto

Além de suas capacidade atuais, é possível estender o HTML através de generalizações de seus comportamentos, de modo que ele continue inteiramente coerente com a proposta do modelo de hipermídia da Web. A ideia por trás desta iniciativa é generalizar o HTML como hipermídia e torná-lo mais completo como um hipertexto ao remover algumas limitações. Como não rompem com o modelo de hipermídia, poderão ser implementadas futuramente pelos principais navegadores e pelos padrões da Web, mas, por enquanto, a única forma de obter essas generalizações é através de código sob demanda (*scripts*).

3.4.1 Substituição total do documento durante a navegação

Uma das principais limitações do comportamento atual dos navegadores (Seção 2.1) é a necessidade de substituição completa do documento HTML durante a navegação. Essa limitação, causa da introdução do modelo SPA, impacta diretamente a experiência do usuário. O processo de recarregamento completo da página, intrínseco à interação com controles de hipermídia, como links de navegação ou submissões de formulários, exige que um novo documento HTML seja totalmente processado a cada interação. Tal abordagem resulta em um comportamento perceptível ao usuário e, muitas vezes, inconveniente, particularmente em aplicações que demandam alta interatividade.

Essa característica representa uma oportunidade significativa para a evolução do HTML. A generalização de sua funcionalidade, de modo a permitir que as respostas às requisições substituam apenas partes específicas do documento existente, ao invés de exigir sua substituição total, poderia conferir o dinamismo desejado em arquiteturas SPA, sem comprometer as vantagens da hipermídia. Não há nenhuma restrição inerente ao HTML ou à sua arquitetura que imponha a necessidade de substituição integral do documento durante a troca de hipermídia no cliente. No entanto, como os navegadores não oferecem suporte nativo a essa funcionalidade, sua implementação deve ser realizada por meio de transclusão no lado do cliente, utilizando *scripts*, conforme discutido na Seção 3.3.

3.4.2 Limitação aos elementos âncoras e formulários para requisições HTTP

Conforme discutido na Seção 3.1.1, o HTML define apenas dois elementos como controles de hipermídia: a âncora (`<a>`) e o formulário (`<form>`). Esses são os únicos mecanismos nativos capazes de disparar requisições HTTP, possibilitando a interação do cliente com os recursos disponíveis no servidor. Essa restrição constitui uma limitação importante na

utilização do HTML como ferramenta de hipermídia, dado que outros elementos não podem, por padrão, assumir o papel de controle de hipermídia. A ampliação dessa funcionalidade permitiria que **qualquer elemento do HTML atuasse como um controle capaz de enviar requisições ao servidor.** Por exemplo, um botão (`<button>`) poderia ser configurado para disparar uma requisição solicitando a remoção de um recurso específico no servidor, em vez de ser necessário encapsulá-lo em um formulário para atingir este objetivo.

3.4.3 Limitação aos eventos de clique e submissão para disparo de requisições HTTP

Além da restrição imposta pela limitação a dois controles de hipermídia (âncora e formulário), o HTML também limita os eventos capazes de disparar essas requisições HTTP a dois tipos específicos: o clique sobre uma âncora e a submissão de um formulário. Embora o Modelo de Objeto de Documento (DOM) ofereça suporte a uma ampla gama de eventos que podem ser disparados e monitorados, como digitação de teclas, posicionamento do cursor sobre elementos ou alteração de estados como foco, o HTML nativo não permite que tais eventos sejam utilizados para acionar requisições HTTP de maneira direta (mas apenas por comportamento definido em *scripts*). A possibilidade de expandir o HTML para permitir que mais eventos (ou qualquer evento) dispare requisições ao servidor abriria novas possibilidades de construção da interface. Por exemplo, a detecção de um evento de foco em um campo de entrada (`<input>`) ou a interação do cursor com um elemento (*mouseover*) poderiam ser configuradas para enviar informações ao servidor ou solicitar recursos.

3.4.4 Limitação aos métodos GET e POST no HTML

Em um sistema de hipermídia, é importante seguir as especificações para o uso correto dos métodos e dos códigos de retorno em HTTP. Porém, um fato estranho sobre o HTML é que seus controles de hipermídia nativos são capazes apenas de enviar requisições HTTP de métodos `GET` e `POST`. Essa limitação implica que todas as operações de obtenção de dados são realizadas com o método GET, enquanto todas as atualizações ou mutações são realizadas com o método `POST`.

Apesar disso, o protocolo HTTP oferece uma maior variedade de métodos, como `PUT`, `PATCH` e `DELETE`, que foram projetados para representar diferentes ações lógicas em conformidade com as especificações do protocolo. O uso desses métodos seria mais adequado para diversas operações e promoveriam maior clareza semântica. Se um botão da interface gráfica é responsável por apagar determinado recurso, seria adequado que ele disparasse uma requisição de método `DELETE`, e não `POST`. Considerando que o protocolo HTTP foi concebido para ser utilizado em conjunto com o HTML, e por isso ambos foram projetados em paralelo, é de se estranhar que o HTML puro não ofereça suporte a todos os métodos do protocolo HTTP. Atualmente, a única maneira de acessar métodos como `PUT`, `PATCH` e `DELETE` é por meio de requisições AJAX realizadas com *scripts*.

Capítulo 4

Implementação de uma prova de conceito no projeto BikeSP

Uma prova de conceito foi implementada para o projeto BikeSP com o objetivo de validar os conceitos desenvolvidos nos capítulos anteriores. Através dessa implementação, foi possível avaliar o impacto real da adoção dos estilos arquitetônicos propostos, além de realizar métricas sobre o código desenvolvido.

4.1 Projeto BikeSP

O projeto BikeSP é uma iniciativa desenvolvida com o objetivo de implementar políticas públicas na cidade de São Paulo visando promover a migração modal para o uso da bicicleta como meio de transporte. Essa estratégia é baseada no incentivo monetário aos ciclistas, por meio da criação de um programa que concede créditos de mobilidade àqueles que utilizam bicicletas para seus deslocamentos urbanos. Trata-se de um projeto piloto, focado nos moradores da cidade de São Paulo, que visa avaliar a viabilidade de um programa de remuneração para ciclistas.[1]

4.2 Projeto de um painel de gerenciamento para administradores

O código do projeto BikeSP já inclui os protótipos de um aplicativo móvel desenvolvido em React Native[2]e de um servidor escrito em Python com o auxílio da biblioteca Flask,[3] que oferece uma API no formato JSON ao cliente. No entanto, com a proposta da disponibilização do projeto ao público, tornou-se necessário implementar uma série de ações de gerenciamento no banco de dados para oferecer suporte adequado aos usuários.

[1] Mais detalhes podem ser encontrados em https://interscity.org/bikesp.

[2] https://reactnative.dev/

[3] https://flask.palletsprojects.com/en/stable/

Até o momento, essas tarefas administrativas eram realizadas por meio de *scripts* escritos em Python. A proposta de desenvolver uma interface gráfica para o gerenciamento visa, além de melhorar a visualização dessas operações, possibilitar que usuários sem conhecimento em programação e na estrutura interna do banco de dados possam administrar os recursos do sistema, dispensando a necessidade de executar *scripts* na linha de comando.

A prova de conceito foi desenvolvida tendo em vista suprir a necessidade desta interface gráfica, na qual o mesmo sistema foi desenvolvido em duas versões, mantendo as mesmas funcionalidades e aparência. A primeira implementação foi realizada utilizando um ecossistema em React, configurando uma aplicação de página única (SPA). A segunda versão foi construída com a utilização da biblioteca htmx, visando a criação de uma aplicação orientada a hipermídia (HDA).

Na primeira versão deste sistema é possível autenticar-se como administrador e, em seguida, realizar operações de criação, atribuição e consulta de bônus no Bilhete Único dos usuários.

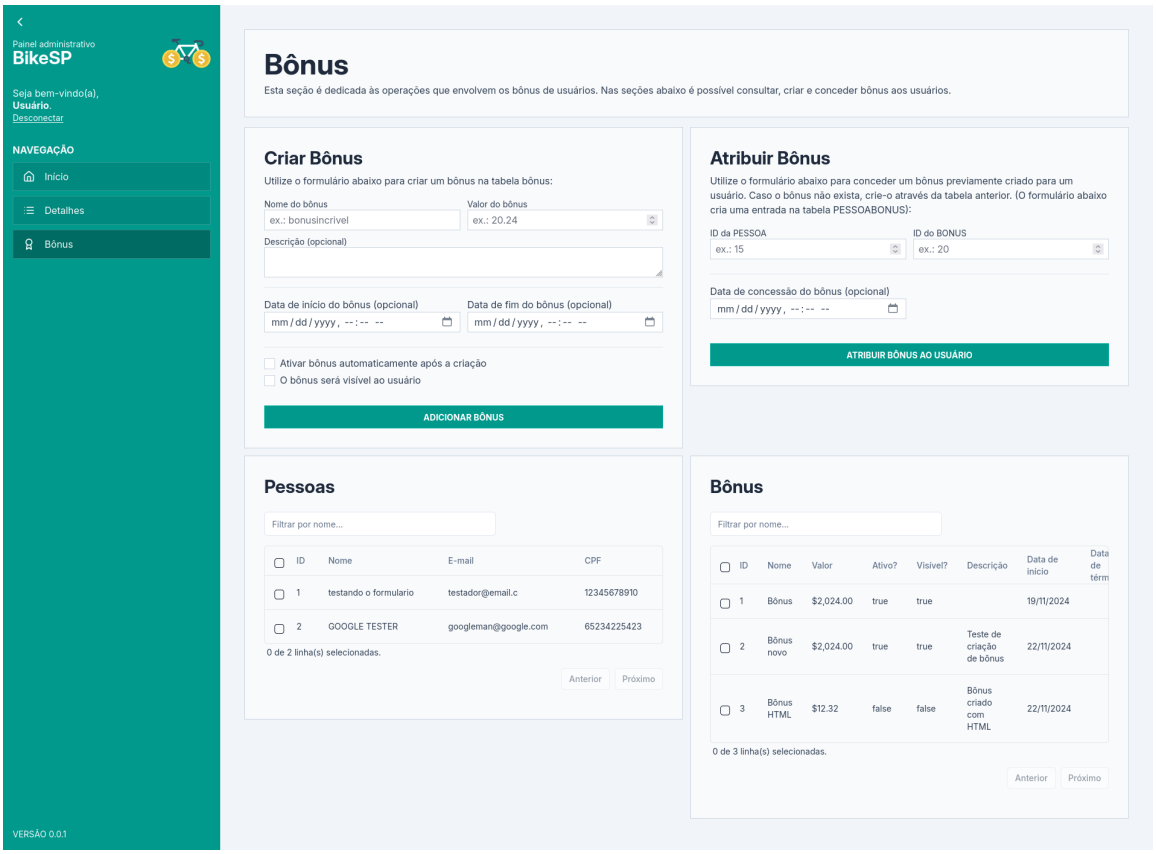


Figura 4.1: *Captura de tela da aplicação desenvolvida em React.*

4.3 Primeira implementação: SPA

A escolha das tecnologias utilizadas para o desenvolvimento de software no modelo *Single-Page Application* baseou-se no critério de popularidade, uma vez que o objetivo seria reproduzir as técnicas mais comuns utilizadas neste cenário. O primeiro passo foi

decidir qual seria a biblioteca ou arcabouço responsável pelo gerenciamento do estado da aplicação e da atualização dinâmica da interface gráfica. Foram consultadas duas pesquisas recentes desenvolvimento para a Web: a *StackOverflow Developer Survey* de 2024[4] e a *State of JavaScript* de 2023.[5]

Na pesquisa realizada pelo StackOverflow, ao serem perguntados sobre os arcabouços e tecnologias web que foram utilizadas de maneira extensiva no último ano, o React se destaca dentre os demais arcabouços de *front-end*, selecionado por 39,5% dos inquiridos, seguido pelo arcabouço Angular, selecionado por apenas 17,1% dos inquiridos (Tabela 4.1).

StackOverflow Survey 2024: Arcabouços e tecnologias para a Web[a]	
Pergunta: *"Quais arcabouços e tecnologias web você utilizou de forma extensiva no desenvolvimento no último ano e quais você deseja trabalhar no próximo ano? (Se você trabalhou com o framework e deseja continuar a utilizá-lo, marque ambas as opções na mesma linha.)"*	
React	39,5%
Angular	17,1%
Vue	15,4%
Svelte	5,5%
Solid	1,2%
Número total de inquiridos	48,503

Tabela 4.1: *Resultados selecionados da pesquisa do StackOverflow em 2024 relativos ao uso de arcabouços para o desenvolvimento da interface gráfica do usuário (front-end).*

[a] A versão completa pode ser consultada em https://survey.stackoverflow.co/2024/technology/\#1-web-frameworks-and-technologies.

Sustentando a preferência do público, a última pesquisa realizada pelo *State of JavaScript* também indica que o React possui maior popularidade entre os desenvolvedores, sendo mais comum encontrar inquiridos que tiveram alguma experiência com essa biblioteca que as demais (Tabela 4.2).

Todavia, há de se notar que, dentre os arcabouços listados na Tabela 4.2, o React fica em segundo lugar na ordem de experiências negativas coletadas proporcionalmente pela mesma pesquisa (Tabela 4.3), indicando que sua grande popularidade não necessariamente indica uma grande satisfação por parte dos usuários.

Foi escolhida a tecnologia React para desenvolver a aplicação em SPA em virtude de sua larga popularidade em cima dos outros arcabouços, como demonstrado pelas Tabelas 4.1 e 4.2. Em seguida, foi necessário definir quais seriam as bibliotecas e ferramentas do ecossistema de React que seriam utilizadas para o desenvolvimento.

Novamente, optou-se pelo critério de popularidade, desta vez consultando a pesquisa *State of React*[6] de 2023 e boas práticas de desenvolvimento nesta plataforma. Optou-se

[4] Pode ser encontrada em: https://survey.stackoverflow.co/2024/.

[5] Pode ser encontrada em https://2023.stateofjs.com/en-US/.

[6] Pode ser encontrada em: https://2023.stateofreact.com/en-US/

State of JavaScript Survey 2023: Experiência com arcabouços de *front-end*[a]				
	Usou	Ouvir falar	Nunca usou	Total de inquiridos
React	84,4%	15,5%	0,1%	21,619
Vue	58,1%	49,5%	0,4%	21,589
Angular	45,8%	53,9%	0,3%	21,578
Svelte	25,8%	70,5%	3,7%	21,507
Solid	9%	67,6%	23,4%	21,432

- **Usou:** inquiridos que utilizaram um item.
- **Ouviu falar:** inquiridos que ouviram falar sobre um item, mas não o utilizaram.
- **Nunca ouviu falar:** inquiridos que nunca ouviram falar sobre um item.

Tabela 4.2: *Resultados selecionados da pesquisa do State of JavaScript em 2023 relativos à experiência com arcabouços para o desenvolvimento da interface gráfica do usuário (front-end).*

[a] A versão completa pode ser consultada em https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/.

pela seleção das seguintes tecnologias:

- TanStack Query: gerenciamento de requisições, consultas e *cache*;
- React Router: navegação e gerenciamento de rotas;
- TanStack Table: tabelas altamente dinâmicas e personalizáveis;
- Sonner's Toaster: exibe notificações temporárias e não invasivas;
- Iconify (Lucide): pacote de ícones;
- shadcn/ui: biblioteca de componentes de interface de usuário;
- TailwindCSS: utilitário de CSS que permite estilizar diretamente no JSX através de classes pré-definidas;
- Vite: ferramenta de compilação para aplicações Web.

4.4 Segunda implementação: HDA

A seleção das tecnologias utilizadas no sistema orientado a hipermídia foi guiada pelo princípio da simplicidade. Buscou-se priorizar dependências com uma curva de aprendizado reduzida, que tirassem proveito da sintaxe já presente nas tecnologias do projeto BikeSP e nas APIs nativas dos navegadores.

Nesse contexto, foram analisadas três bibliotecas com abordagem orientada a hipermí-

State of JavaScript Survey 2023: Sentimento em relação a arcabouços de *front-end*.[a]			
	Positivo	Negativo	Total de inquiridos
Solid	88,9%	11,1%	1,929
Svelte	84,6%	15,4%	5,548
Vue	76%	24%	10,816
React	72,2%	27,8%	18,246
Angular	43,5%	56,5%	9,883

As respostas foram coletadas somente a partir dos inquiridos que responderam terem utilizado a tecnologia.

- **Positivo:** inquiridos que têm interesse em aprender mais sobre uma tecnologia ou estão dispostos a usá-la novamente.
- **Negativo:** inquiridos que não têm interesse em aprender mais sobre uma tecnologia ou que a utilizaram e tiveram uma experiência negativa.

Tabela 4.3: *Resultados selecionados da pesquisa do State of JavaScript em 2023 relativos ao sentimento dos desenvolvedores com arcabouços para o desenvolvimento da interface gráfica do usuário (front-end).*

[a] A versão completa pode ser consultada em https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/.

dia: Unpoly,[7] Hotwire,[8] e htmx.[9] Essas bibliotecas compartilham a característica de adotar o HTML como elemento central da aplicação, utilizando a hipermídia como motor de estado (Seção 1.2.4), buscando minimizar, ou até mesmo eliminar completamente, a necessidade de código personalizado em JavaScript.

A análise comparativa revelou que a biblioteca Unpoly oferece uma ampla gama de funcionalidades, muitas das quais não se alinham diretamente aos requisitos específicos deste projeto. Por exemplo, as funcionalidades de divisão em camadas e de pré-carregamento a partir de previsões de interação são parte do núcleo da biblioteca, mas não são necessárias na aplicação BikeSP, cuja simplicidade tornaria esse recurso redundante e potencialmente complexo de gerenciar.

De forma semelhante, o Hotwire consiste em uma coleção de bibliotecas, dentre as quais se destaca o Turbo.[10] Sua abordagem abstrai grande parte dos detalhes técnicos, de modo que o alto nível de abstração pode dificultar a compreensão dos mecanismos internos das estruturas empregadas, que pode ser desvantajoso em cenários que exigem maior controle do comportamento do sistema ou a implementação de uma estrutura simples que não necessita de abstrações.

[7] https://unpoly.com/

[8] https://hotwired.dev/

[9] https://htmx.org/

[10] https://turbo.hotwired.dev/

Por outro lado, a biblioteca htmx apresenta um nível de abstração reduzido em comparação ao Turbo e à Unpoly, priorizando flexibilidade em detrimento de uma abordagem mais implícita. Embora o htmx não ofereça suporte ao desenvolvimento de aplicativos móveis, isso não constitui uma limitação, dado que esse aspecto não faz parte do escopo do projeto do painel de administração do BikeSP. Além disso, a biblioteca é menos diversificada em suas estruturas e funcionalidades que a Unpoly, o que contribui para uma baixa curva de aprendizado e facilidade de uso.

Com base na análise comparativa, optou-se pela utilização do htmx neste projeto, por sua maior proximidade com o uso de HTML puro e pela menor abstração oferecida em relação às outras opções avaliadas. Enquanto a Unpoly fornece uma gama de ferramentas mais abrangente e o Hotwire adota uma abordagem mais automática, o htmx mostrou-se como a solução mais adequada aos objetivos específicos do sistema em HDA que pretendia-se desenvolver para o BikeSP. O resumo pode ser visto na Tabela 4.4.

	Unpoly	Hotwire	htmx
HTML é o elemento central?	Sim	Sim	Sim
Cria novos elementos no HTML?	Não	Sim	Não
Cria novos atributos no HTML?	Sim	Sim	Sim
Nível de abstração	Baixo	Alto	Baixo
Funcionalidades prontas sem intervenção do desenvolvedor[a]	Muitas	Moderado	Poucas
Suporte para aplicações móveis?	Não	Sim	Não

Tabela 4.4: *Comparação entre as bibliotecas de HDA: Unpoly, Hotwire e htmx.*

[a] Do inglês: *out of the box functionalities.*

4.4.1 Breve descrição da biblioteca htmx

A biblioteca htmx foi escolhida por constituir, dentre as opções avaliadas, a experiência mais próxima do uso de HTML tradicional. Esta biblioteca se propõe a expandir as funcionalidades do HTML de modo a permitir maior interatividade enquanto hipermídia (Seção 3.4), através da adição de 36 novos atributos que descrevem padrões de comportamento dos elementos mediante interação do usuário.

Estes novos atributos adicionados pelo htmx tornam o HTML mais expressivo. A biblioteca é responsável por processá-los e por executar o comportamento descrito por eles. Os atributos configuram um disparos de requisições HTTP às rotas definidas no servidor, que devolvem HTML que será inserido no DOM pela biblioteca htmx através de técnicas de transclusão (Seção 3.3).

Sem o uso do htmx, seria necessário escrever código personalizado de JavaScript para realizar essas operação, tendo em vista que este comportamento de transclusão não está implementado nos navegadores por padrão. Qualquer evento acionado por qualquer

elemento HTML é capaz de disparar uma requisição HTTP de qualquer método, e o HTML contido na resposta à requisição não resultará em uma substituição total da página, mas em uma substituição dinâmica com base nos critérios definidos nos atributos do elemento HTML. Isso é feito sem a necessidade de que o desenvolvedor escreva qualquer tipo de código em JavaScript, pois, uma vez que a interface de comunicação é uniforme, o htmx é capaz de implementar uma solução generalizada que não exige conhecimento interno da estrutura do servidor para atualizar a interface gráfica.

A sintaxe do htmx é fortemente baseada nas APIs já existentes no navegador, possui um tamanho pequeno (aproximadamente 14kB comprimido com gzip) e é capaz de implementar uma série de padrões modernos que são mais que suficientes para a implementação de um painel de administrador, como *prompts* de confirmação, sondagem,[11] *web sockets*, eventos no lado do servidor,[12] carregamento sob demanda,[13] *scroll* infinito, melhorias progressivas,[14] animações em elementos e trocas de páginas, integração com a API de histórico do navegador, integração com Web Components etc.

Por fim, um servidor que interage com um cliente que faz uso da biblioteca htmx deve responder às requisições HTTP com HTML ou texto e também deve servir arquivos estáticos, assemelhando-se bastante ao papel dos servidores nos primórdios da Web.

4.4.2 Motor de *templates*

O servidor já existente para o projeto BikeSP foi implementado na linguagem Python, com o suporte da biblioteca Flask. O Flask é uma biblioteca para o desenvolvimento de aplicações web baseada na especificação WSGI (*Web Server Gateway Interface*). No contexto do projeto, ele foi utilizado para a construção de uma API JSON responsável pela interação com o aplicativo móvel desenvolvido em React Native.

Além de suas funcionalidades básicas, o Flask adota o Jinja2[15] como motor de *templates* padrão. Embora seja possível integrar outros motores de *template*, ainda sim é necessária a instalação do Jinja2, pois habilita o uso de extensões avançadas no Flask. Portanto, optou-se pela utilização do Jinja2 para a construção do HTML gerado pela API de hipermídia do HDA desenvolvido. Essa escolha alinha-se com os objetivos do projeto, reforçando os valores de simplicidade e compatibilidade com as tecnologias já consolidadas no sistema, evitando a inserção de mais uma dependência.

[11] Do inglês: *polling*.

[12] Do inglês: *server-side events*.

[13] Do inglês: *lazy loading*

[14] Do inglês: *progressive enhancement*

[15] https://jinja.palletsprojects.com/en/stable/

4.5 Análise comparativa

4.5.1 Análise de métricas no código-fonte

Foram realizadas análises de métricas no código-fonte de ambos os projetos, com o objetivo de comparar os possíveis impactos decorrentes da adoção das diferentes arquiteturas. A síntese dos resultados obtidos está disposta na Tabela 4.5.

	SPA (React)	HDA (htmx)
Número de dependências adicionadas ao projeto	33	3
Total de linhas de código do cliente (ignorando dependências)	1477	849
Total de linhas de código alteradas no servidor	113	304
Total de novos arquivos criados (excluindo dependências)	39	17
Total de dados carregados na inicialização no navegador	3,52MB	86,76kB
Total de scripts carregados na inicialização no navegador	3,43MB	53,67kB
Tempo médio de compilação	8,12s	0 (não há)

Tabela 4.5: *Algumas métricas comparativas entre a implementação SPA e HDA (dependências, dados transferidos, linhas de código etc).*

Uma das métricas mais relevantes foi a redução significativa no número de dependências utilizadas. Enquanto a arquitetura baseada em SPA requeria 33 dependências externas, a implementação com HDA reduziu esse número para apenas 3: a biblioteca *htmx*, uma extensão e um normalizador de CSS.[16] Essa redução foi alcançada sem comprometer a funcionalidade da aplicação, demonstrando a possibilidade de minimizar o uso de código de terceiros e, consequentemente, reduzir a complexidade e os riscos associados a vulnerabilidades externas (Seção 2.3.7).

Outra métrica analisada foi a quantidade de linhas de código alteradas. Na implementação baseada em HDA, o número de alterações realizadas no servidor foi superior ao observado na arquitetura SPA, como esperado, uma vez que o servidor assume um papel central no processamento dos dados e na geração da representação dos recursos. Contudo, esse acréscimo foi compensado pela expressiva redução no volume de código escrito para execução no cliente, o que contribui para a simplificação da manutenção da aplicação.

Uma análise mais detalhada da distribuição de linhas de código por linguagem pode

[16] O normalizador de CSS utilizado foi o *reset.css*.

ser encontrada na Tabela 4.6, indicando a redução do número de linguagens utilizadas e a substituição de código em lógica de JavaScript ou TypeScript para simples arquivos de HTML e CSS.

SPA (React)	
Linguagem	Linhas de código
TypeScript	1262
JSON	131
JavaScript	57
CSS	13
HTML	13
SVG	1

(a) *Número de linhas de código por linguagem na implementação em SPA.*

HDA (htmx)	
Linguagem	Linhas de código
HTML	708
CSS	141

(b) *Número de linhas de código por linguagem na implementação em HDA.*

Tabela 4.6: *Comparação de linguagens e linhas de código entre as implementações do cliente do painel em SPA e em HDA.*

Além disso, a implementação em HDA demandou aproximadamente metade do número de arquivos necessários para a implementação do mesmo sistema em SPA. Essa simplificação também eliminou o processo de compilação (detalhado na Seção 4.5.5), resultando na eliminação de todo o tempo que esta etapa exigia na atualização do servidor ou na execução de testes, favorecendo a agilidade do desenvolvimento.

Por fim, a análise do volume de dados transferidos durante a inicialização da aplicação no navegador revelou resultados notáveis. A implementação baseada em SPA transferiu um total de 3,52 MB, sendo 3,43 MB exclusivamente de arquivos de *scripts*, para o carregamento inicial da primeira página. Em contraste, a implementação com HDA transferiu apenas 86,76 kB, dos quais 53,67 kB correspondem a *scripts*. Esse aspecto é discutido em maior detalhe na Seção 4.5.3.

4.5.2 Flexibilidade

Na implementação em HDA, foi possível construir a interface gráfica utilizando o motor de *templates* Jinja2, que combina HTML com a linguagem do servidor (Python), eliminando a necessidade da duplicação da lógica em JavaScript e a pressão para a adoção de JS como linguagem do servidor (Seção 2.3.9).

A partir do momento em que um sistema de hipermídia é adotado, **resgata-se a flexibilidade na escolha da linguagem do lado do servidor**. A capacidade de utilizar a mesma linguagem do servidor na construção da interface gráfica evita a duplicação de lógica em duas linguagens distintas e permite a escolha da linguagem do servidor com base em considerações técnicas, estéticas e outras, em vez do critério anterior que consistia no fato de o JavaScript ser a única linguagem disponível para ambos (Seção 2.3.9). Essa

característica é extensível a outras linguagens e ambientes de desenvolvimento, e não apenas ao Python, pois o modelo permite que a interface seja construída com qualquer motor de *templates* com suporte a HTML.

Ademais, com o auxílio de uma biblioteca orientada a hipermídia, como htmx, foi possível escrever uma interface dinâmica com pouco ou nenhum código em JavaScript (Tabela 4.6). Reduzir o número de linguagens do projeto facilita o entendimento da equipe sobre o código e diminui a complexidade do sistema ao reduzir o número de dependências.

Em contraste, na abordagem baseada em SPA, as estruturas da interface precisam ser obrigatoriamente definidas em JavaScript, devido às exigências inerentes aos arcabouços específicos dessa arquitetura, restando apenas a escolha do JavaScript como linguagem adotada pelo servidor como meio de evitar múltiplas linguagens e lógica duplicada no sistema.

4.5.3 Processo de inicialização

O processo de inicialização no cliente desenvolvido em SPA (Figura 2.3) exige três trocas de mensagens com o servidor para exibir o conteúdo da página inicial. Das três trocas de mensagem, as duas primeiras servem apenas para alterar a interface gráfica de uma página em branco para um estado de aparente carregamento, e a última requisita à API os dados a serem exibidos, propriamente. Este processo está descrito com mais detalhes na Seção 2.3.4.

Além do número de etapas, vale ressaltar que toda a aplicação é transferida no momento da inicialização, pois todas as funcionalidades precisam ser carregadas *a priori*. Ao longo do uso da aplicação pelo usuário, o cliente dispara requisições somente para troca de dados puros à API JSON. Portanto, quanto maior o código da aplicação, mais demorado o processo de inicialização, pois maior será o pacote gerado na compilação e carregado integralmente na inicialização do cliente.

A inicialização do cliente em HDA (Figura 4.2) reduz o número de trocas de mensagem com o servidor de três para uma. Não há estágio intermediário de aparente carregamento pois a página populada é enviada de uma vez na representação HTML gerada pelo servidor, e o navegador a exibe instantaneamente.

Além do número de etapas reduzido, vale ressaltar que no modelo HDA não é necessário enviar todo o código do cliente na inicialização. Ou seja, diferentemente do SPA, o processo de inicialização do HDA não é afetado pelo aumento da base de código nas outras partes do sistema.

A mudança neste processo causou uma notável otimização no volume de dados transferido (Tabela 4.5), reduzindo o uso da rede de 3,52MB (SPA) para apenas 86,76kB (HDA) transferidos para inicializar a aplicação. A solução em HDA, neste aspecto, é mais rápida, mais direta e mais eficiente para a rede.

4.5.4 Fluxo de uma requisição

Após a inicialização, o cliente continua a se comunicar com o servidor para atualizar o conteúdo da página. Na abordagem em SPA,o cliente se comunica com o servidor através

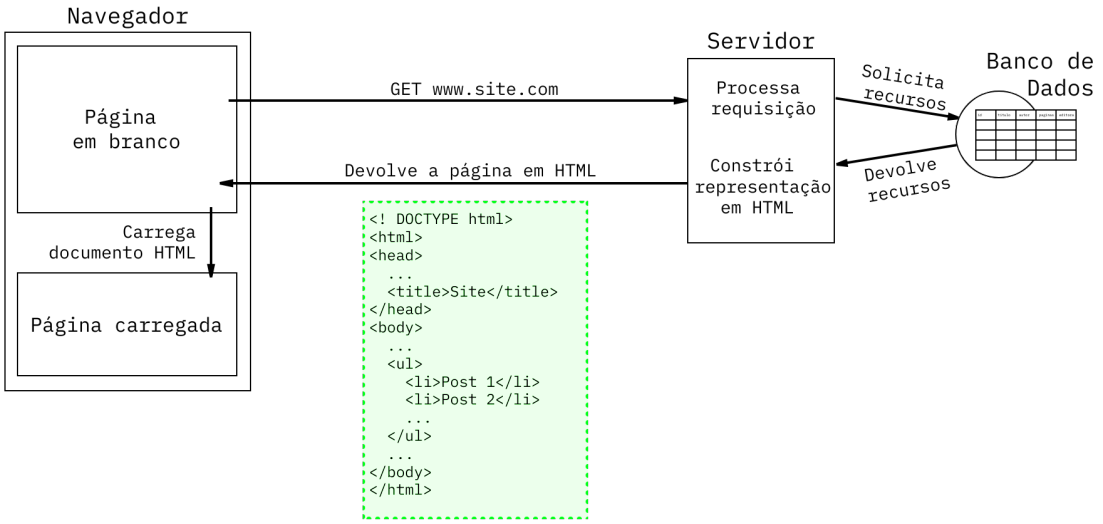


Figura 4.2: *Diagrama do processo de inicialização de uma aplicação HDA no cliente.*

da requisições AJAX para a API JSON (Figura 4.3). Após receber a solicitação do recurso, o servidor constrói a representação dos dados deste recurso em formato JSON e a envia ao cliente. Ao receber a resposta, o cliente processa os dados de acordo com o estado previamente armazenado, calcula e armazena o novo estado resultante da interação e, com o conhecimento do estado atual e dos dados recebidos, contrói a representação do recurso em HTML para em seguida atualizar o DOM dinamicamente, inserindo o novo recurso na interface gráfica do usuário.

O processamento exigido para o fluxo de uma requisição também foi otimizado no sistema em HDA (Figura 4.4). Como o sistema em HDA é *RESTful* e não necessita do armazenamento e gerenciamento do estado no cliente, pois a hipermídia atua como motor do estado da aplicação (Seção 3.3), torna-se dispensável a representação intermediária de dados puros em JSON para fazer esse gerenciamento. Basta apenas que a representação em HTML seja construída no servidor e enviada ao cliente para atualizar a interface gráfica diretamente.

A adoção dessa arquitetura traz como consequência direta a redução significativa da complexidade do cliente. Nesta abordagem, o cliente não é mais responsável por manter estruturas altamente sofisticadas para a realização de operações complexas, como o gerenciamento e a sincronização de estados ou a construção dinâmica de elementos HTML.

A responsabilidade destas operações foi mantida totalmente no servidor, que assume o papel principal no processamento e na geração de conteúdo. Um cliente menos carregado, como é o caso do HDA, facilita o desenvolvimento, a manutenção e a escalabilidade da aplicação como um todo.

4.5.5 Processo de despacho

Uma aplicação em SPA exige um processo de compilação (Seção 2.3.5). Este processo é composto por uma série de etapas, cada uma complexa por si só, em que serão gera-

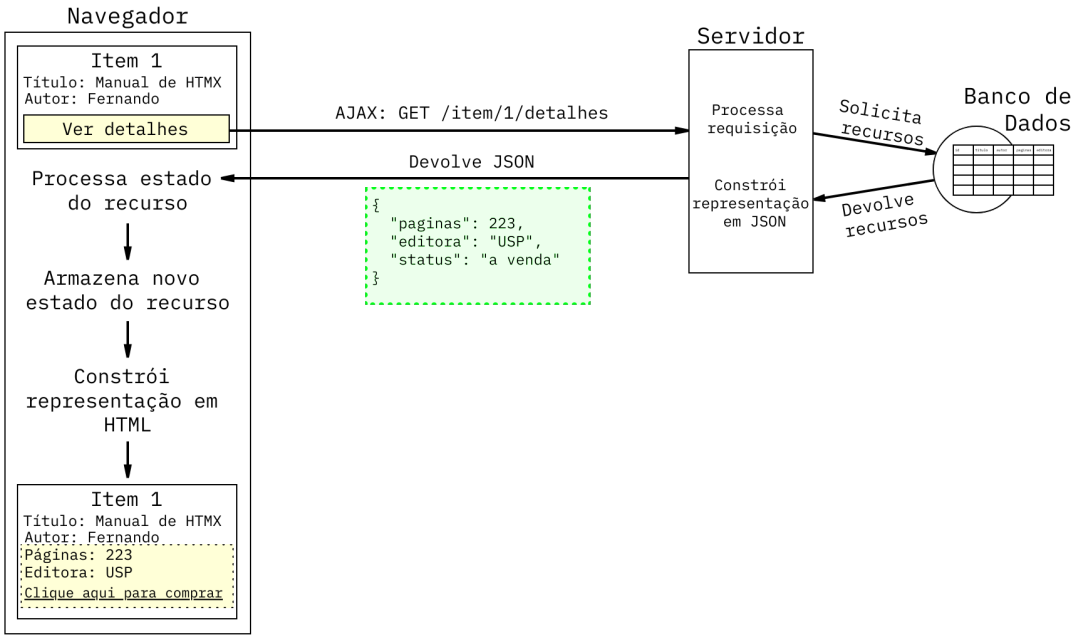


Figura 4.3: *Diagrama do fluxo de uma requisição do cliente em uma aplicação SPA.*

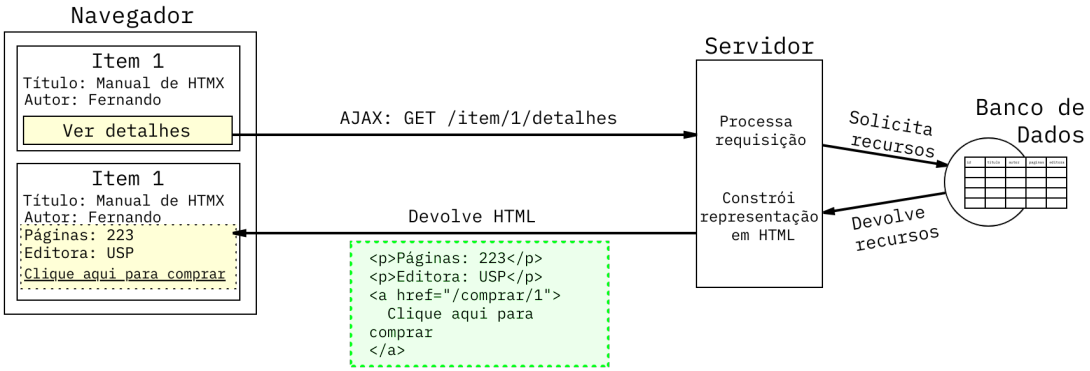


Figura 4.4: *Diagrama do fluxo de uma requisição do cliente em uma aplicação HDA.*

dos arquivos compilados que deverão substituir totalmente os arquivos do servidor em produção (Figura 2.4). A cada pequena mudança, mesmo que seja um único caractere inserido no código, é necessário que toda a base passe novamente por todas as etapas do processo de compilação, e que todos os arquivos em produção sejam substituídos pelos novos arquivos gerados.

Na implementação em HDA, pelo fato de a inicialização ser enxuta e a interface gráfica já ser construída diretamente em HTML e CSS através do motor de *templates* (em vez de asbtrações complexas), não é necessário nenhum tipo de processo de otimização prévia, pois os navegadores já são capazes de interpretar os arquivos do jeito que foram escritos. Não há a necessidade de um processo de compilação, muito menos da substituição de todos os arquivos do servidor em produção no processo de despacho. A cada mudança

realizada na base, basta apenas a atualização dos arquivos alterados no diretório em produção (Figura 4.5).

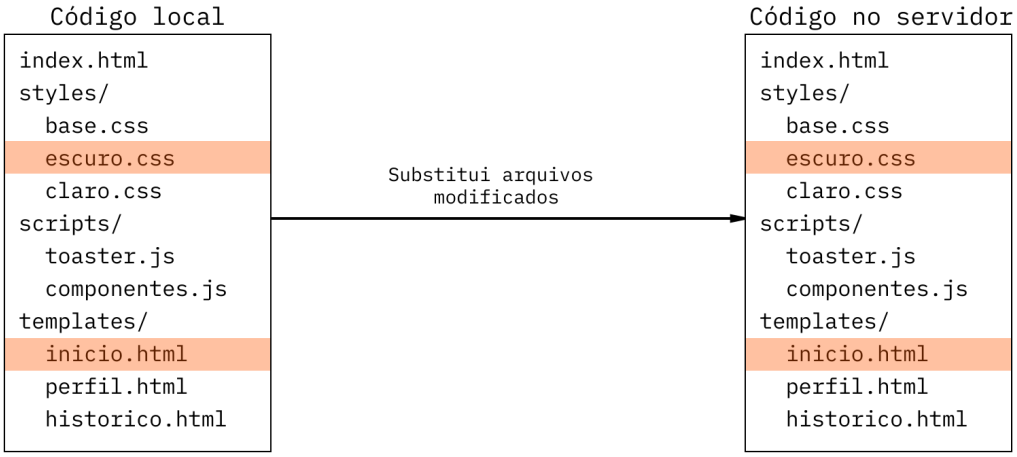


Figura 4.5: *Diagrama do processo de despacho de uma aplicação HDA no cliente, sem a necessidade de compilação.*

4.5.6 Estratégia de estilização

Na implementação baseada em SPA, a estratégia de estilização adotada utilizou classes de utilidade fornecidas pela biblioteca TailwindCSS,[17] aplicadas diretamente no atributo `className` dos elementos JSX do React.

O estilo é definido na estrutura de cada elemento em vez de ser gerido por folhas de estilo separadas. Essa abordagem promove a componentização da interface gráfica, mesmo em cenários em que não há uma justificativa estrutural para isso. Quando um elemento estilizado dessa maneira é reutilizado em outro contexto, torna-se necessário replicar manualmente suas definições de estilo na nova marcação, levando a repetições frequentes no código.

A própria documentação do TailwindCSS reconhece a possibilidade de duplicação na combinação de classes ao estilizar componentes semelhantes.[18] Para mitigar esse problema, recomenda-se o uso de recursos específicos do editor de código, como edição com múltiplos cursores,[19] ou funcionalidades próprias da linguagem, como laços.[20] No entanto, essas soluções introduzem limitações adicionais: nem todas as linguagens oferecem suporte a laços, especialmente o HTML puro, e depender de funcionalidades do editor restringe a flexibilidade do desenvolvedor na seleção de ferramentas de trabalho.

Por outro lado, na implementação baseada em HDA, optou-se pelo uso de CSS moderno, evitando o emprego de abstrações como classes de utilidade e bibliotecas externas. Apesar

[17] https://tailwindcss.com/

[18] https://tailwindcss.com/docs/reusing-styles\#using-editor-and-language-features

[19] https://tailwindcss.com/docs/reusing-styles\#multi-cursor-editing

[20] https://tailwindcss.com/docs/reusing-styles\#loops

disso, a abordagem tradicional de declarar estilos em CSS por meio de folhas de estilo apresenta desafios. Ao utilizar folhas de estilo, é necessário definir identificadores ou classes únicas para estilizar componentes específicos ou reduzir o escopo das regras CSS. Essa prática frequentemente resulta em uma proliferação de termos específicos e estilos indesejados aplicados globalmente, dificultando a manutenção e organização do código.

Apesar de este comportamento estar em conformidade com o princípio de separação de conceitos (SoC), esta separação total da marcação com o estilo nem sempre é ideal para cenários que demandam modificações rápidas ou mudanças pontuais de estilo. Como alternativa, a estilização na arquitetura HDA utilizou a regra modena `@scope`,[21] inserida em uma *tag* `<style>`. Essa abordagem permite criar escopos de estilização diretamente no navegador, restringindo os estilos a uma ramificação específica da hierarquia do documento HTML ou de um *template*, mantendo os estilos encapsulados e organizados sem aplicá-los globalmente, sem a necessidade de criar classes e identificadores personalizados e sem depender de camadas externas de abstração.

Um exemplo desta estratégia pode ser visto no Programa 4.1. Os estilos definidos dentro de `div` apenas serão aplicados naquele escopo, não afetando os elementos fora da hierarquia em que se encontra a *tag* `style`.

Programa 4.1 Exemplo de implementação da regra de CSS @scope localizada no HTML

```
<section>
  <div>
    <style>
      @scope {
        p {
          color: red;
        }

        p+p {
          color: green;
        }
      }
    </style>

    <p>Este parágrafo évermelho.</p>
    <p>Este parágrafo éverde.</p>

  </div>

  <p>Este parágrafo não possui estilo algum.</p>
  <p>Este também não.</p>
</section>
```

[21] Mais detalhes em: https://developer.mozilla.org/en-US/docs/Web/CSS/@scope

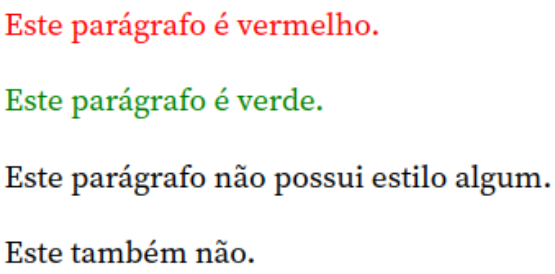


Figura 4.6: *Interface gráfica exibida no exemplo da aplicação da regra @scope no CSS do Programa 4.1.*

4.6 Limitações de uma aplicação orientada a hipermídia

A arquitetura HDA requer a presença de um servidor, uma vez que essa abordagem depende intrinsecamente de requisições ao servidor para qualquer atualização da interface do usuário. Consequentemente, **o suporte ao comportamento offline não é viável**, dado que o cliente de hipermídia não possui conhecimento interno das regras de negócio da aplicação para operar de forma independente.

A abordagem baseada em hipermídia também apresenta limitações em casos que demandam atualizações intensas na interface. Para alterações pontuais e localizadas, como em dois ou três elementos, a solução é prática e de baixa complexidade. Contudo, à medida que o número de alterações simultâneas aumenta, por exemplo, para mais de 20 elementos, a complexidade cresce exponencialmente, **tornando-se um desafio gerir muitas interdependências dinâmicas simultaneamente**.

No entanto, é relevante destacar que esta arquitetura se comporta como o uso de HTML puro, o que possibilita a integração de bibliotecas de JavaScript utilizadas para desenvolver SPAs por meio de processos de hidratação em ilhas iterativas. Pode-se, por exemplo, carregar uma aplicação em React em apenas um bloco específico de uma página de um HDA para implementar funcionalidades que exigem maior interatividade no cliente ou comportamento offline.

As aplicações orientada a hipermídia demonstram bastante eficiência nos casos em que há necessidade de manipulação de muitos textos e imagens, de implementação do modelo CRUD (*Create, Read, Update and Delete*), de suporte a ligações profundas,[22] de desempenho no primeiro carregamento (Seção 4.5.3) e de atualizações concentradas em blocos bem definidos da interface gráfica.

Por outro lado, **apresenta limitações em cenários que demandam funcionalidades offline, múltiplas interdependências dinâmicas espalhadas pela página, e interfaces com estados altamente dinâmicos que demandam atualizações frequentes.**

[22] Do inglês: *deep links*.

4.7 SPA nem sempre é indesejado

Cabe destacar que, apesar dos resultados demonstrados anterioremente, a estratégia arquitetural baseada em *Single-Page Application* pode ser adequada em determinados contextos. A adoção dessa abordagem é apropriada quando os requisitos do sistema demandam um comportamento altamente dinâmico, similar ao de aplicativos nativos, em direção oposta a de páginas estáticas da Web compostas majoritariamente por textos, tabelas e figuras.

Sistemas que exigem interatividade avançada e atualizações constantes da interface em múltiplos pontos, com interdependências dinâmicas, se beneficiam significativamente da arquitetura SPA. Além disso, a capacidade de operação offline é um fator determinante em muitos casos de uso, justificando sua aplicação.

Um exemplo de uma aplicação adequada de um arcabouço de SPA para o desenvolvimento de um sistema Web é o Google Docs.[23] Em todas as suas ferramentas, a interface atualiza dinamicamente, em intervalos curtos, em diferentes blocos de interação, gerando várias interdependências dinâmicas. E, principalmente, preza por oferecer o funcionamento offline aos usuários. Não há outra forma de obter sucesso no cumprimento destes requisitos a não ser por um cliente robusto, como o de um SPA.

No entanto, o presente trabalho procurou enfatizar que a adoção de SPAs deve ser pautada por uma análise criteriosa de sua adequação aos requisitos funcionais e comportamentais do sistema, evitando sua utilização como uma solução universal. A decisão de incorporar essa arquitetura deve considerar os impactos que ela pode trazer ao sistema, assegurando que sua complexidade inerente contribua para alcaçar os objetivos do projeto.

[23] https://docs.google.com/.

Conclusão

Este trabalho abordou criticamente o panorama do desenvolvimento de aplicações para a Web, expondo as limitações que emergem do modelo *Single-Page Application* (SPA) a partir do momento que rompe com o estilo arquitetônico REST. Foi proposta uma alternativa por meio de aplicações orientadas a hipermídia (HDA). Baseado nas restrições REST, foi demonstrado que é possível aliar simplicidade arquitetural à funcionalidades modernas através da adoção de estratégias de transclusão de documentos para substituir padrões modernos de usabilidade nas interfaces gráficas de sistemas para a Web.

Os resultados obtidos com o protótipo desenvolvido para o projeto BikeSP reforçam a viabilidade técnica e os benefícios aos desenvolvedores em relação à manutenção e à visibilidade do código na adoção da arquitetura de HDA, especialmente em cenários onde a complexidade inerente ao SPA não traz vantagens significativas ao sucesso do projeto, como em sistemas majoritariamente estáticos. A análise comparativa destacou a eficiência no uso de recursos, a menor carga de dependências e a simplificação do processo de desenvolvimento.

Dessa forma, este estudo contribui para a reflexão sobre os paradigmas de desenvolvimento para a Web e sugere uma retomada de práticas mais alinhadas com os princípios fundamentais da WWW. Futuros trabalhos podem expandir este debate, avaliando a escalabilidade das HDAs em projetos de maior envergadura e aprofundando o impacto dessa arquitetura na experiência do usuário e na integração com ferramentas de desenvolvimento.

Referências

[Berners-Lee e Connoly 1995] Tim Berners-Lee e Daniel Connoly. *Hypertext Markup Language 2.0 - Specification*. RFC 1866. Nov. de 1995. url: https://www.rfc-editor.org/rfc/rfc1866 (acesso em 25/10/2024) (citado na pg. 30).

[Brooks 1987] Frederick P. Brooks. "No silver bullet - essence and accidents of software engineering". *Computer* 20 (1987), pp. 10–19. url: https://www.cs.unc.edu/techreports/86-020.pdf (citado na pg. 3).

[Fielding 2000] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Tese de dout. Irvine: University of California, 2000. url: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (acesso em 06/04/2024) (citado nas pgs. iii, v, 1–6, 10, 18).

[Fielding 2008] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. 2008. url: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (acesso em 13/11/2024) (citado na pg. 18).

[Gross *et al.* 2023] Carson Gross, Adam Stepinski e Deniz Akşimşek. *Hypermedia Systems*. Publicação independente, jul. de 2023. url: https://hypermedia.systems/book/contents/ (acesso em 25/10/2024) (citado nas pgs. 9, 14, 16, 18–20, 27, 29–32).

[Nelson 1977] Theodor Holm Nelson. *Selected Papers. Papers on Computers and Interaction, from 1965 to 1977*. Fev. de 1977. url: https://archive.org/details/SelectedPapers1977/mode/2up (acesso em 25/10/2024) (citado na pg. 30).

[Prokopov 2024] Nikita Prokopov. *JavaScript Bloat in 2024*. 2024. url: https://tonsky.me/blog/js-bloat/ (acesso em 25/11/2024) (citado na pg. 22).

[WHATWG 2024] WHATWG. *Hypertext Markup Language Living Standard*. Out. de 2024. url: https://html.spec.whatwg.org/ (acesso em 25/10/2024) (citado nas pgs. 27, 31).