



Arquiteturas Front-End

Entenda a diferença entre SSR,
CSR e SSG



Olá!

Eu sou Willian Brito

- ❖ Desenvolvedor FullStack na Msystem Software
- ❖ Formado em Analise e Desenvolvimento de Sistemas
- ❖ Pós Graduado em Segurança Cibernética
- ❖ Certificação SYCP (Solyd Certified Pentester) v2018

Introdução

No início da World Wide Web, tudo era estático. Uma página na internet nada mais era do que um arquivo html, com texto e alguns links para outras páginas. Isso é o que chamamos de site estático. Esse modelo tinha a vantagem de ser extremamente performático, visto que o tempo de processamento de um arquivo html é muito pequeno.

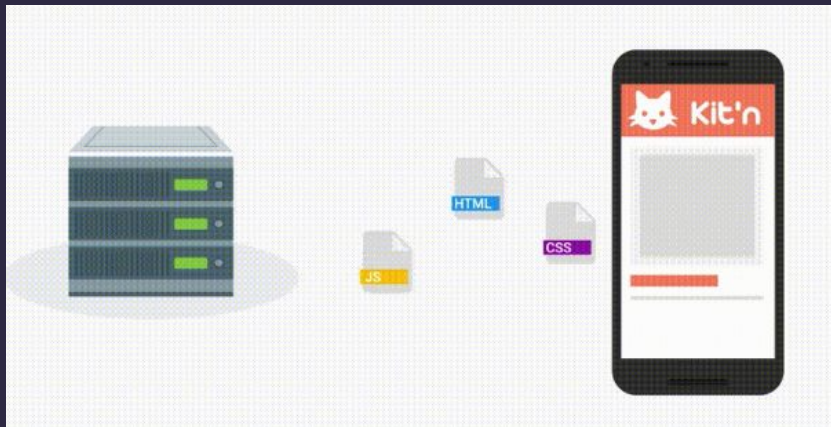
Porém, com o passar do tempo, as informações contidas nos sites foram ficando cada vez maiores, tornando impraticável a construção de um arquivo com tantas linhas. Outro problema era que se um site tivesse mil páginas, mil arquivos html deveriam ser criados na mão. Imagina se alguma informação tivesse que ser atualizada?

Além disso, a falta de dinamismo dos sites estáticos começou a ser um problema. A partir dessa necessidade surgiram as páginas dinâmicas, que nada mais eram do que páginas web que sofriam algum tipo de alteração no momento da requisição. Mas para entender um pouco dos formatos de renderização das aplicações web, temos que entender os princípios do carregamento que é realizado no seu navegador para ver o real benefício de se usar algum modelo de arquitetura que iremos falar adiante.

Carregamento da Página

Quando uma página é carregada ela necessita de input (que chamamos de assets). Estes são os primeiros conteúdos a serem entregues para o navegador para que, a partir dali, ele possa realizar o seu trabalho e renderizar a página o mais rápido possível para o usuário.

Para continuarmos a abordagem, vamos entender como o navegador recebe e faz o uso disso no nosso site.




Carregamento da Página



Inicialmente, quando acessamos um conteúdo na web, temos uma referência no documento HTML que nos indica o que será carregado e também será a primeira coisa que o navegador vai receber, esse documento contém todas as referências necessárias para os seguintes assets, como imagens, CSS e javascripts.

O navegador sabe que a partir disso ele precisa ir a algum lugar para localizar e baixar esses assets, enquanto ele vai construindo o documento. Então, mesmo que o navegador contenha toda a estrutura HTML, ele não poderá renderizar algo amigável até que o CSS correspondente, que contém toda a sua estilização, seja também carregado.

Uma vez feito, o seu navegador poderá ter algum conteúdo relevante para entregar ao usuário. Após esse processo finalizado, o navegador irá baixar todos os arquivos javascripts e é aí que costuma estar a maioria dos problemas.



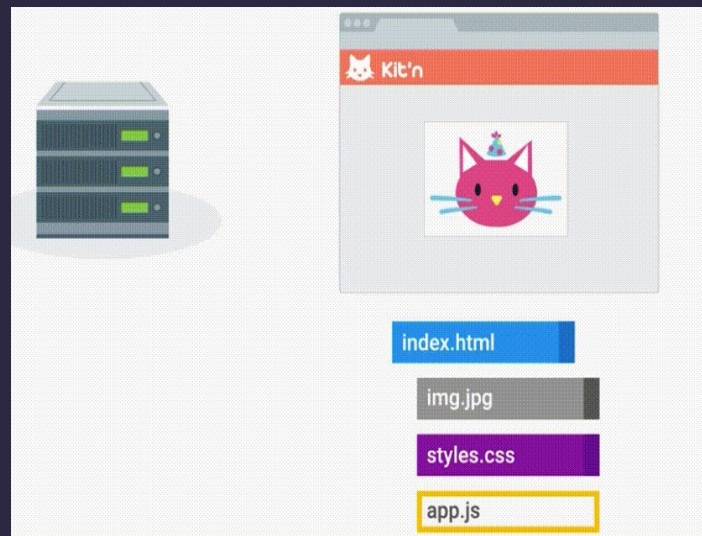
Carregamento da Página

Os arquivos poderão ser grandes e com o uso de uma internet ruim o tempo de carregamento poderá ser grande. Desta maneira, a experiência do seu usuário não será a ideal, o que pode piorar se a primeira renderização depender de algum arquivo javascript.

O grande diferencial de usar SSR é que podemos entregar, quase que imediatamente, um conteúdo significativo para o usuário. Isto acontece porque o HTML e seus principais assets são carregados em um mesmo arquivo e entregue ao navegador do usuário.

Portanto, eliminamos algumas etapas no processo de download de assets e o navegador fica encarregado somente de carregar os componentes criados e seus fluxos que também ficam em um arquivo minificado. Ainda como boa prática, recomendo fortemente a utilização de algum CDN para realizarmos o cache desses arquivos.

Agora que entendemos o contexto dos sites estáticos e dinâmicos, vamos entender o que significam SSR, SSG e CSR.




Carregamento da Página



Como vimos anteriormente, uma página estática é um arquivo HTML que não sofre processamento no servidor. Ela é enviada para o usuário do jeito que está.

Já uma página dinâmica (ou processada no servidor) sofre alterações no momento em que o sistema recebe uma requisição, ela é gerada na hora e enviada para o usuário na sequência.

Agora nós sabemos o que é server side render (renderizar no servidor) e o que é uma página estática, vamos continuar no fluxo histórico e entender como o SSR funciona.



SSR

Server Side Rendering



Server Side Rendering

SSR (Server Side Rendering), é um formato de renderização de páginas já bastante conhecido. Como o nome diz, é uma **renderização do lado do servidor**, ou seja, nada mais é que utilizar as linguagens de programação do **backend** para **gerar o html das páginas**.

Desta forma, haverá a necessidade de uma estrutura no servidor responsável não apenas por servir os assets, mas também por gerar as páginas HTML já completas, com o conteúdo populado, cada vez que o usuário digita o endereço de um site, uma requisição é feita ao servidor que ficará responsável por gerar o html dessa página e devolver ao navegador.

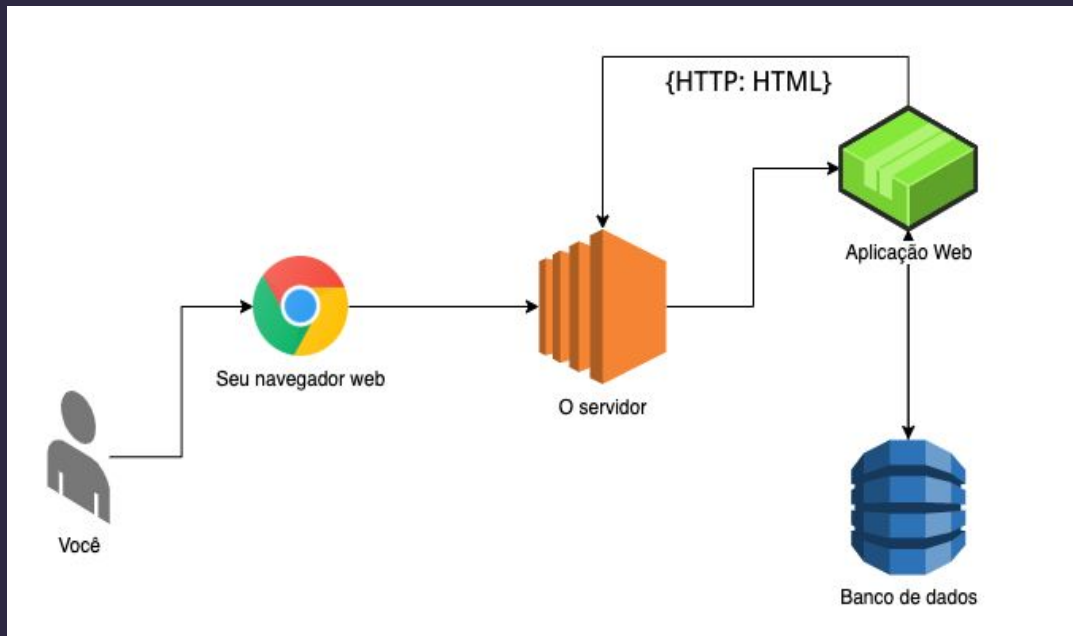
O servidor web irá se comunicar com o backend da aplicação web que será responsável por consumir dados de algum banco de dados, arquivo de textos, do sistema operacional ou até mesmo outras aplicações web, armazenar isso em memória (na memória do servidor mesmo) e na sequência injetar isso em algum template e criar as páginas HTML. Com essas páginas criadas, com os dados que foram solicitados, agora o servidor web recebe o texto e envia para o usuário que requisitou os dados.

Server Side Rendering

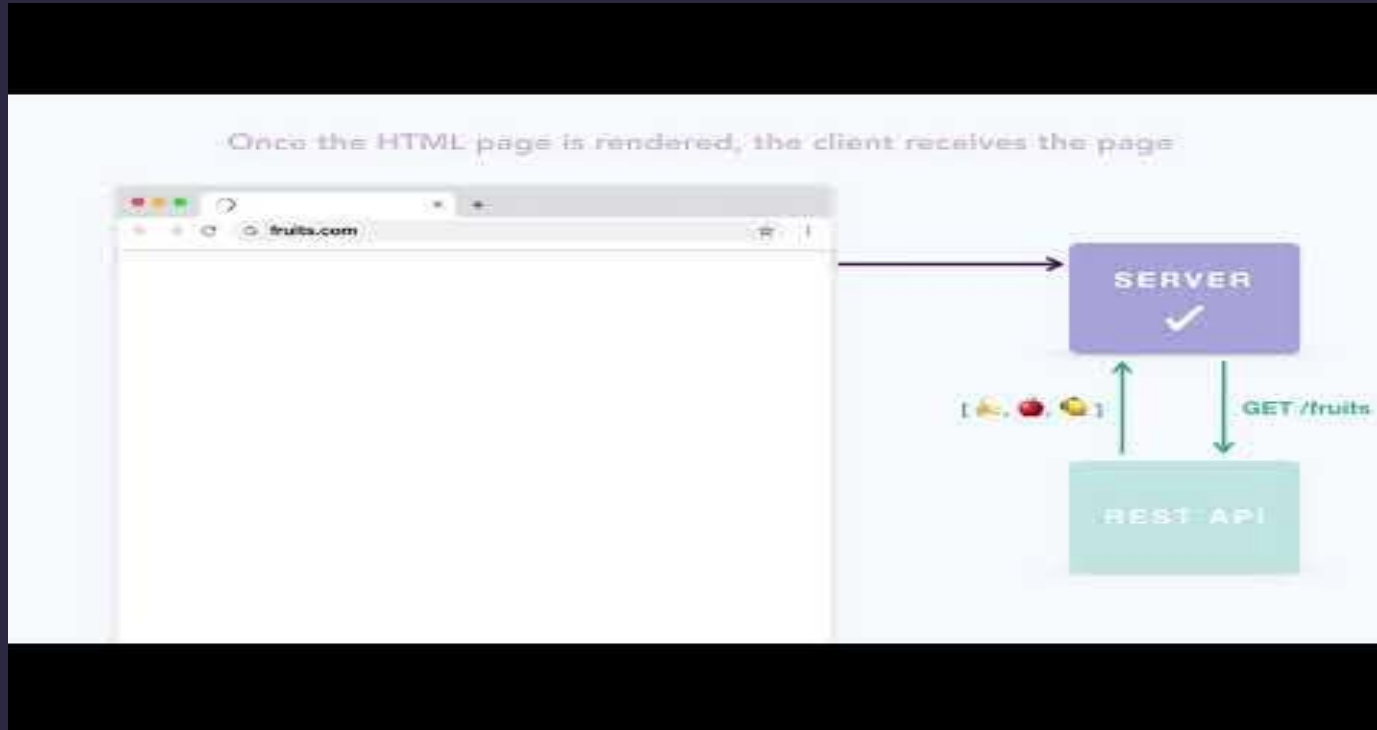
Entendendo Melhor a Arquitetura Web (SSR)

Para ilustrar melhor o que explicamos até aqui, desenhei alguns diagramas que vão explicar como funciona uma aplicação via SSR.

De uma maneira muito, mas realmente muito simplificada: temos você acessando um site com o seu navegador, o servidor utilizando a aplicação que bate em um banco de dados, processa a informação, devolve para o servidor e o servidor te envia uma página **html**.



Fluxo de Execução SSR



Server Side Rendering

❏ Quais problemas o SSR resolve?

Primeiramente, questões relacionadas a **SEO** (Search Engine Optimization). Como numa **SPA** (iremos ver adiante) a renderização é feita no browser, e alguns web crawlers não tem capacidade de executar código JavaScript, apenas HTML, o web crawler captura uma página com praticamente nenhuma informação semântica, o que é ruim para o SEO.

Segundo, há as questões relacionadas à performance. Uma página com o HTML com o conteúdo necessário já servido é bem melhor do que você ter este mesmo conteúdo em um JavaScript que será baixado, parseado e executado em um momento posterior. Não só isso, em um contexto em que as pessoas usam mais os seus smartphones do que seus computadores para ter acesso a informação na internet, ter uma menor quantidade de código JavaScript é melhor. Lembre-se: performance também é uma métrica para a experiência do usuário.

Server Side Rendering

❏ Quando utilizar Server Side Rendering?

- Quando tem necessidades de um SPA, porém precisa de um loading mais rápido.
- Quando o conteúdo muda frequentemente.
- Quando trabalha com um número muito grande de páginas.
- Quando precisa de uma boa indexação no Google.

Exemplos: E-commerce, Sites de Notícias.

Server Side Rendering

❏ Vantagens

- Exige menos da máquina do cliente, pois parte da renderização é feita no servidor.
- Redução do risco da página em branco (flicker) quando ela ainda está carregando.
- Melhor indexação, já que o servidor renderiza parte da página. Assim, esse mecanismo colabora com os motores de busca e pode melhorar o posicionamento no Google.

❏ Desvantagens

- O usuário vê a página sendo carregada, pois o TTFB (time to first byte) é maior, já que o servidor deve pré-carregar uma parcela do conteúdo para mostrar a página ao cliente.
- Quando essa resposta inicial é mais lenta, pode gerar uma experiência negativa no usuário final, mesmo que o carregamento seja mais rápido que o normal.
- Reload completo nas mudanças de rota.
- Dificuldade com múltiplos Front-End.

CSR

Client Side Rendering



Client Side Rendering

Será que é responsabilidade do Backend gerar html? Com o avanço das tecnologias e dispositivos no “lado do cliente (client-side)” como TVs e Smartphones, o **Frontend** começou a ganhar mais responsabilidades e isso se deve ao surgimento de um novo formato de renderização que é o **CSR** (**C**lient **S**ide **R**endering).

Mas antes de falar de CSR ou SPAs (**S**ingle **P**age **A**pplication), vamos falar de tecnologias que vieram antes e que contribuíram muito com o surgimento das SPAs de hoje.

❏ A Era do Ajax

Focando no desenvolvimento de sistemas e não em sites comerciais, as coisas eram razoavelmente simples. Um framework back-end se encarregava da estrutura **MVC** (**M**odel **V**iew **C**ontroller), onde o back-end era **fortemente acoplado** ao front-end e as páginas do sistema iam sendo construídas pelo back-end. Não raramente algum javascript era usado para validar um formulário ou criar um pedido **ajax**, porém pouco a pouco o uso do ajax foi se tornando cada vez mais primordial.

Client Side Rendering

❏ V8 Engine

Nesse período o javascript ainda era tratado apenas como uma coisa que resolvia alguns problemas no navegador. Até que em setembro de 2008 a Google lança o **V8 Engine**, o motor de javascript capaz de executar o seu código javascript com performance nativa. Isso foi uma verdadeira **revolução**, o mercado de navegadores, até então devagar, tomou um novo fôlego vislumbrando novas possibilidades.

Depois dele outras empresas criaram seus motores. Não sendo o bastante, em Maio de 2009 veio à tona o projeto **Node.js**, usando o V8 por baixo do capô, ele nos deu a possibilidade de executar códigos javascript também no lado do servidor.

Com a evolução e popularização de ferramentas javascript, tarefas antes mais complexas se tornaram mais simples e algumas vezes mais leves.

❖ **Ajax** passou a ser usado em larga escala para as mais diversas situações, e com o passar do tempo, o usuário ficava cada vez mais tempo na mesma página, sem nunca dar um **refresh**.

Client Side Rendering

❏ As Primeiras Ferramentas

Algumas empresas começaram a criar projetos com comportamentos até então não “documentados”. A página da aplicação era carregada apenas uma vez durante todo o **ciclo de uso do usuário**, e mesmo que o usuário recarregasse a aplicação ela possuía a habilidade de se manter na última tarefa feita pelo usuário, em muitos momentos com a tela exatamente como o usuário a deixou.

A mais “**famosa**” dessas aplicações foi o **GMail**. A cada ação executada pelo usuário a página não recarregava, porém era possível ver o valor na barra de endereços do navegador mudando conforme o usuário ia navegando. Olhando com atenção era possível ver que apenas os valores após **# (hashtag)** eram alterados.

Com o passar do tempo técnicas e ferramentas foram sendo criadas. Como tudo na web, a evolução foi constante. Algumas dessas ferramentas foram: **Backbone.js**, **Ember**, **Knockout**, **Batman.js**, **Marionette.js**, **AngularJS** e muitas outras que eu não conseguiria listar.

Cada uma dessas ferramentas tinha sua filosofia e maneira de resolver as demandas propostas por SPAs. Algumas tentavam ser uma solução completa, outras soluções mais pontuais. Enquanto algumas eram mais rígidas em como a aplicação deveria ser criada, outras eram mais flexíveis.

Client Side Rendering



É interessante ver como é a evolução do ecossistema javascript, tudo que o angular fazia não era uma completa novidade, porém ao trazer a filosofia de dar mais “poder” ao HTML, ele pegou o que já existia e deu uma forma simples de usar.

Alguns anos depois, com o surgimento de novas demandas, onde nem todas puderam ser atendidas pelo AngularJS, novas ferramentas surgiram.

Agora que entendemos como as SPAs surgiram vamos começar a falar sobre o que é, e o que define uma aplicação SPA. Como você já percebeu, trata-se de uma aplicação que **não faz refresh** no navegador ao mudar de página. Essa é a definição mais simplista, porém há muito mais do que isso por baixo dos panos.



Client Side Rendering

❑ CSR (Client Side Rendering)

Todos os processos estão sendo executados no **lado do cliente**, ou seja no navegador. A comunicação com o back-end se dá através de chamadas ajax para **end-points** de uma **API**.

Nesse momento as aplicações SPA mostram todo o seu brilho, pois uma aplicação SPA bem feita é totalmente **desacoplada** do **back-end**, ela não sabe nem se importa se a aplicação foi feita em node, C#, php, java, ruby, python, go, elixir, rust... Ou até mesmo se possui mais de uma linguagem de back-end ou se são múltiplas aplicações.

Isso facilita muito a migração de tecnologias, porém exige uma atenção sobre como essa API é construída, para que mudanças futuras não sejam experiências traumatizantes.

Tudo acontecer no front-end também possui suas desvantagens. A fragmentação de regras de negócio é extremamente comum, além até mesmo da duplicação de regras. É difícil escapar disso, um exemplo dessa situação é um formulário simples, onde a validação foi feita no cliente, porém a mesma validação precisa ser feita na API.

Client Side Rendering

❏ Carregamento da Página

O carregamento de uma aplicação SPA pode ser (e muitas vezes é) feito apenas uma vez. É comum colocar uma barra de load/progresso até que tudo esteja pronto. Projetos SPA podem ter desvantagens nesse quesito, pois essa etapa pode ser muito demorada, além de complexa.

Nessa etapa várias coisas acontecem, como carregamento de assets como css, javascript e imagens, como avaliar se o usuário está logado ou fazer requisições para preencher a página antes da primeira interação do usuário. Muitas coisas acontecem nesse momento, e após essa etapa, esse tipo de aplicação mostra quais as vantagens ela oferece como a **experiência do usuário**. Se o projeto for bem construído ele se mostrará rápido e fluido para o usuário.

Muitas dessas aplicações possuem partes real-time, dando um feedback instantâneo ao usuário.

Client Side Rendering

❏ Armazenamento

É muito comum salvar informações localmente no navegador do usuário. Em geral é armazenado um **token** ou qualquer informação que possa identificar o usuário e sua máquina.

É incomum e pouco recomendado o uso de **cookies** ou **sessão** para essa tarefa, o mais comum é usar **localStorage**, um recurso dos navegadores modernos. Funciona basicamente como um banco chave-valor. No caso do armazenamento e uso de tokens há também a recomendação do uso de **HTTP-Only Cookie** para a transmissão do token.

❏ Back-End

Essa é uma parte que muitos não dão a devida importância. Não falo sobre a linguagem ou ferramentas utilizadas no back-end, mas sobre **Arquitetura da API**.

Uma API bem feita é independente de linguagem. Uma API sólida seguindo o modelo **RESTful** corretamente agregará valor e facilidade ao projeto. Porém uma API mal projetada pode comprometer todo o projeto.

Client Side Rendering

❏ Frameworks

Essa é uma parte delicada da construção de aplicações **SPA**, escolher a ferramenta certa. Com o passar do tempo um padrão foi estabelecido na arquitetura das aplicações. Alguns chamavam de **MVVM** (**M**odel-**V**iew-**V**iew**M**odel) outros de **MVW** (**M**odel-**V**iew-**W**hatever). O **AngularJS** e outras ferramentas tentavam simular algo bem próximo do **MVC**, tendo partes da aplicação conhecidas como **controllers**.

Depois disso vieram ferramentas como o **React** e **Vue** com uma proposta diferente, baseada em **web-components**. Web-components já não são novidade, e estão ganhando cada vez mais espaço, resolvendo problemas que ferramentas como o Angular não conseguiam resolver com facilidade.

O entendimento da superioridade dos web-components para a arquitetura de criação de aplicações complexas tem dado cada vez mais destaque para ferramentas que têm eles como filosofia central. Vendo essa migração em massa de desenvolvedores para esse tipo de ferramentas, a nova versão do AngularJS adotou essa filosofia dos web-components.

Client Side Rendering

❏ SPA

Com as SPAs, toda a interface da aplicação fica nas mãos do Frontend, deixando para o Backend somente a responsabilidade de disponibilizar os dados, do banco de dados, via json. Além da parte gráfica, o roteamento também fica sob a responsabilidade do Frontend.

Apesar do nome ser “aplicação de página única”, isso não quer dizer que as SPAs terão somente uma página. Esse termo se deve ao fato de todo o esqueleto da aplicação ser renderizado apenas uma vez, mudando somente o conteúdo principal das páginas.

Por exemplo, imagine que temos uma página web com um menu superior, uma div no meio da tela e um rodapé na parte inferior. O menu e o rodapé serão os mesmos para todas as páginas do site. Não faz sentido gerar o html do menu e do rodapé, toda vez que o cliente acessa uma página deste site.

Com SPAs conseguimos alterar somente a div do meio, mantendo o menu e o rodapé fixos. Isso aumenta a performance e proporciona uma melhor experiência ao usuário.

Client Side Rendering

❏ Fluxo de Execução

Você primeiro cria todo o seu site utilizando o **Create React App** ou qualquer outra estrutura parecida. E na etapa de **build**, é gerado normalmente um arquivo como **app.js**, que vai ser o **coração** da sua aplicação.

No momento que o usuário abrir o seu site, ele irá baixar o esqueleto da aplicação (HTML sem conteúdo), o mesmo terá as chamadas para o seu **app.js** e outras coisas necessárias. Assim que esses arquivos carregarem, o **app.js** fará as chamadas para API e com o retorno dos dados, irá preencher o site com as informações.

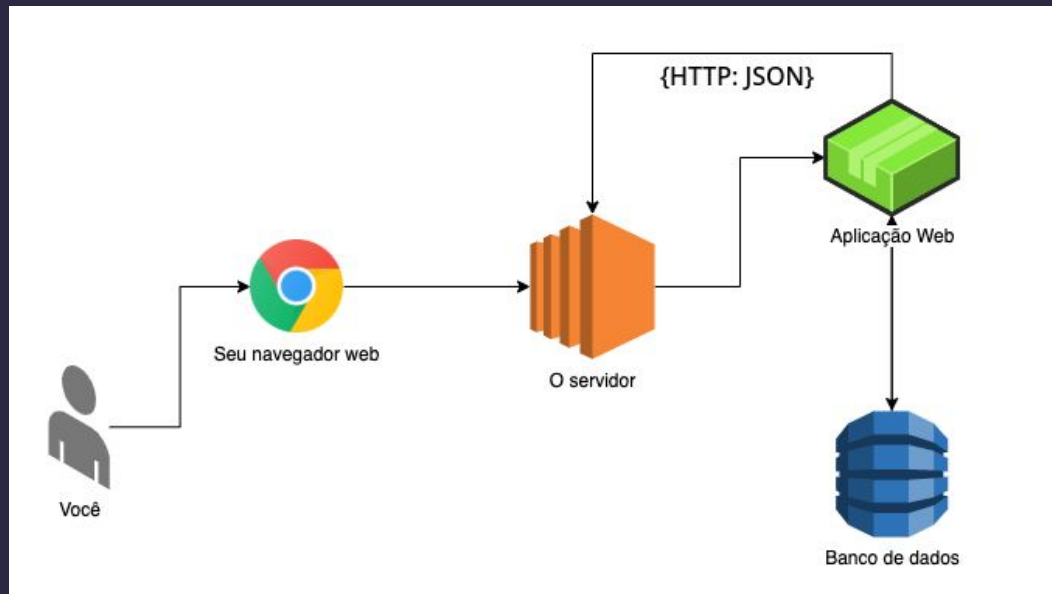
Depois disso, as próximas mudanças de rotas serão mais rápidas, já que o javascript principal (**app.js**) já foi baixado na primeira interação. Esse roteamento também será feito no lado do cliente e não no lado do servidor, o que vai permitir transições mais suaves.

Porém, nem tudo são flores. Por serem carregadas utilizando rotinas Javascript, as SPAs trazem um problema de SEO. Isso ocorre pois normalmente os **algoritmos de busca** executam com o **javascript desabilitado**.

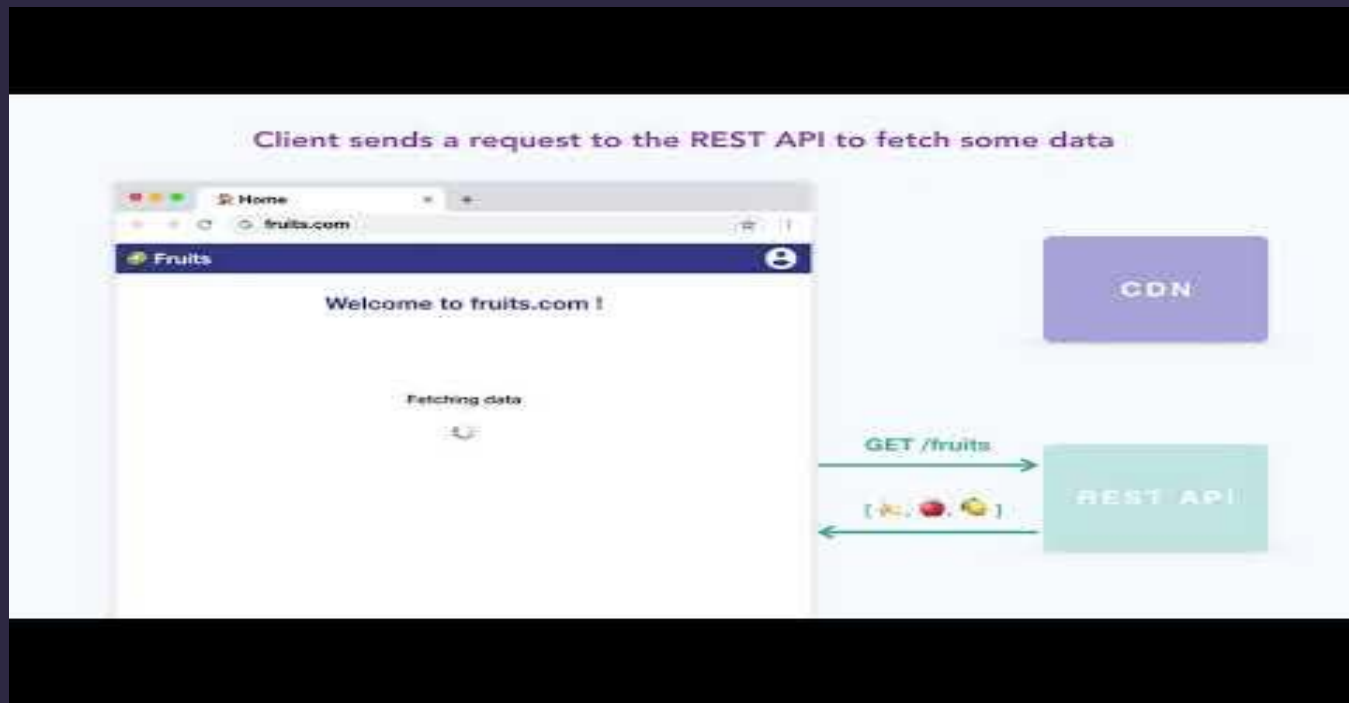
Client Side Rendering

Entendendo Melhor a Arquitetura Web (SPA)

Para ilustrar melhor, de uma maneira muito simples, temos você acessando um site com o seu navegador, que pode ser em um desktop, celular ou TV, o servidor utilizando a aplicação, bate em um banco de dados, devolve para o servidor somente os dados em formato **json** e o navegador renderiza todas as informações no lado cliente.



Fluxo de Execução SPA



Client Side Rendering

❏ Quais problemas o SPA resolve?

Os principais problemas que um SPA resolve é a questão de uma aplicação ter **múltiplos front-end (Desktop, TVs, Smartphones)**, pois o conteúdo é renderizado no lado do cliente e isso só é possível graças a evolução do front-end que citamos anteriormente e o back-end que retorna apenas dados e não html, isso faz com que cada front-end renderiza as informações de maneira específica para o respectivo dispositivo.

Um outro problema que um SPA resolve, que vale a pena citar é a questão da melhora na experiência do usuário, pois as chamadas e a dependência em um servidor foram reduzidas para minimizar os atrasos causados pela latência do servidor, de modo que o SPA se aproxime da capacidade de resposta de um aplicativo nativo.

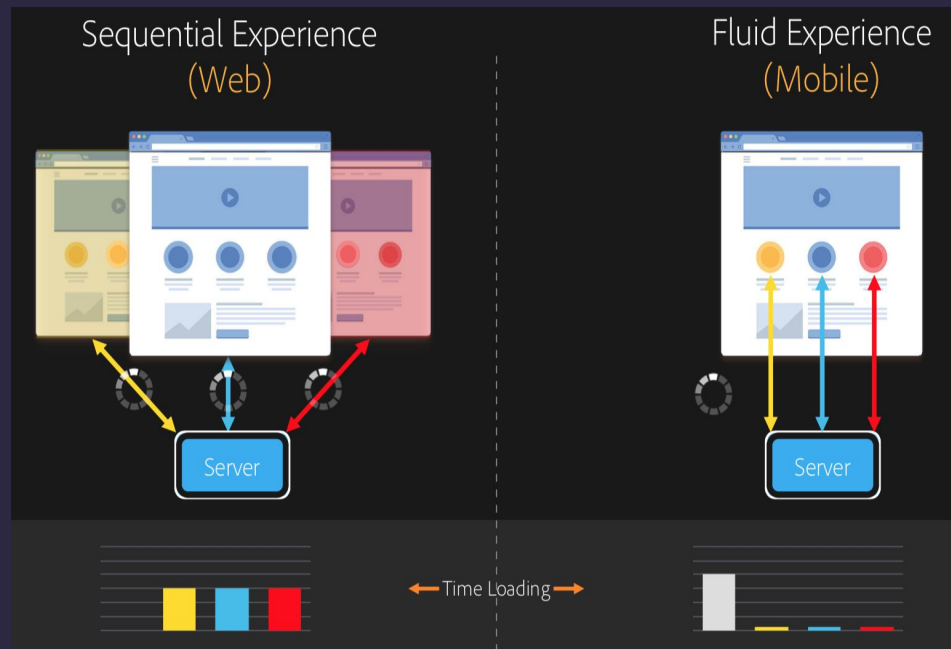
Em uma página da Web tradicional e sequencial, somente os dados necessários para a página imediata são carregados. Isso significa que quando o visitante se move para outra página, o servidor é chamado para os recursos adicionais. Pode ser necessário realizar chamadas adicionais, pois o visitante interage com elementos na página. Essas várias chamadas podem dar uma sensação de atras, pois a página precisa acompanhar as solicitações do visitante.

Client Side Rendering

❑ Quais problemas o SPA resolve?

Para uma experiência mais fluida, que se aproxima do que um visitante espera de aplicativos móveis e nativos, um SPA carrega todos os dados necessários para o visitante na primeira carga. Embora isso possa demorar um pouco mais no início, elimina a necessidade de chamadas de servidor adicionais.

Ao renderizar no lado do cliente, os elementos da página reagem mais rapidamente e as interações com a página pelo visitante são imediatas. Quaisquer dados adicionais que possam ser necessários são chamados de forma assíncrona para maximizar a velocidade da página.



Client Side Rendering

❏ Quando utilizar Single Page Application?

- Quando não tem tanta necessidade de indexar informações no Google.
- Quando o usuário faz muitas interações na página e não quero refreshes.
- Quando as informações vão ser diferentes para cada usuário (autenticação, por exemplo).

Exemplos: Trello, Pinterest, Facebook Web, Spotify Web.

Client Side Rendering

❏ Vantagens

- Melhor experiência de usuário, pois permite páginas ricas com muitas interações sem recarregar.
- Site rápido após o carregamento inicial.
- Ótimo para aplicações web, onde o usuário tem que se autenticar.
- Possui diversos frameworks.

❏ Desvantagens

- Carregamento inicial pode ser muito grande.
- Performance imprevisível.
- Dificuldades no SEO, pois a maioria do conteúdo não é indexado por motores de buscas.

Client Side Rendering

❏ Conclusão

Devido a todas as vantagens técnicas e de negócio são excelentes, mas como dito antes não são todos os projetos que precisam delas. Então é necessário ter certeza das necessidades do projeto antes de optar por construir uma aplicação SPA.

Em geral, projetos onde se há demanda de criação de APIs ou de crescimento a longo prazo são fortes indicadores da necessidade da criação de uma aplicação SPA. Porém (como sempre) também são indicadores da necessidade de profissionais qualificados.

Como uma boa qualificação vem de boas experiências, é bom começar com projetos mais simples, entendendo bem as nuances do projetos.

❖ É importante cada aspecto da aplicação ser tratado de modo isolado (API e Client) pois cada um possui necessidades únicas e bem definidas e por isso é necessário um time para desenvolvimento back-end e outro para front-end.

SSG

Static Site Generation



Static Site Generation

Agora que sabemos que as aplicações web têm o poder de consumir dados e gerar páginas em tempo de execução, fica desnecessário escrevermos HTML simples, páginas estáticas, e guardar em um servidor, certo?

Errado! Existem sim momentos em que é excelente criarmos páginas estáticas ao invés de páginas dinâmicas e você vai entender o porquê agora.

Um servidor de aplicações web precisa ter uma certa quantidade de memória ram, processador e espaço em disco para as coisas acontecerem. E tudo isso, adivinha, tem um custo. Hoje em dia, com os serviços de provedores de computação em nuvem, manter uma aplicação web é muito mais barato, porém é sempre legal conseguirmos economizar um pouco mais, não é mesmo?

A criação dinâmica das páginas web se torna desnecessária quando um dado quase nunca sofre alteração. Ou seja, imagina que você tem um site pessoal, como um blog, ou o site institucional de uma empresa. Essas páginas recebem dados uma vez e isso só vai mudar depois de muito tempo, não é algo que precisa ser recriado ("re-gerado") a cada requisição que o servidor web receber. E é em casos parecidos com esse que os geradores de sites estáticos entram em cena para nos ajudar.

Static Site Generation

❏ Como Funciona

Em um site estático (ou gerado estaticamente), o fluxo é bem simples:

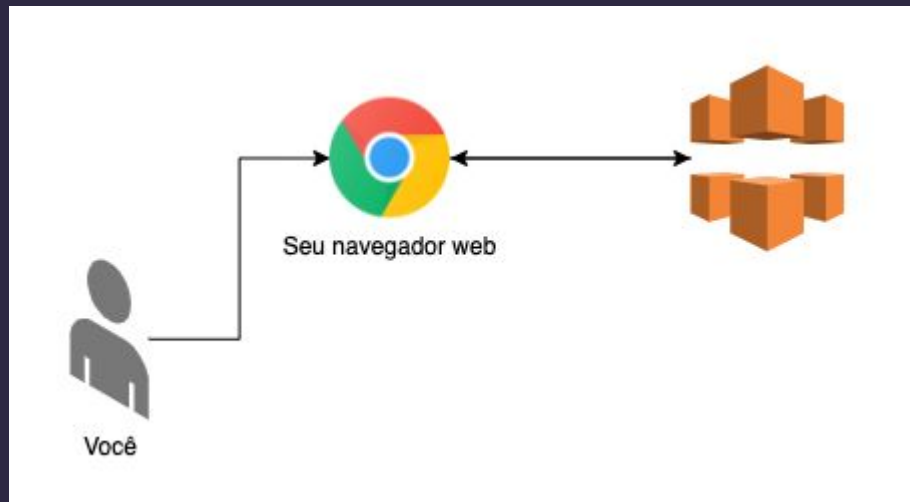
Caso seja um site gerado **estaticamente**, no momento do **build** (uma versão do nosso produto final) nós iremos fazer as requisições para uma API, vamos pegar esses dados e "colar" com os arquivos de template/componentes e iremos gerar a partir dali os arquivos de HTML, CSS e JS.

A partir daí, nós subimos esses arquivos em um servidor (que pode inclusive ser uma CDN) e não haverá nenhum processo mais ocorrendo no lado do servidor. Toda vez que um usuário requisitar uma página, nós vamos pegar o conteúdo daquela página e vamos entregar, como os arquivos foram gerados estaticamente, nenhuma chamada extra será feita e a página já vai ter incluso o HTML total da página.

Static Site Generation

Entendendo Melhor a Arquitetura Web (SSG)

Agora é o mesmo processo que vimos anteriormente, porém não temos mais um servidor processando informações em tempo de execução, somente o envio do conteúdo estático para o seu navegador.



Fluxo de Execução SSG



Static Site Generation

❏ Quando utilizar Static Site Generation?

- Site simples sem muita interação do usuário.
- Se você é a única pessoa que publica conteúdo.
- Se o conteúdo muda pouco.
- Se o site é simples, sem tantas páginas.
- Quando a performance é extremamente importante.

Exemplos: Landing Page, Blogs, Sites de Portfólios.

Static Site Generation

❏ Vantagens

- Uso quase nulo do servidor.
- Pode ser servido numa CDN (melhor cache).
- Melhor performance entre todos, pois não tem renderização no lado cliente e nem no servidor.
- Flexibilidade para usar qualquer servidor.
- Ótimo SEO

❏ Desvantagens

- Tempo de build pode ser muito alto.
- Dificuldade para escalar em aplicações grandes.
- Dificuldade para realizar atualizações constantes.

Static Site Generation

❏ Conclusão

Chegamos ao entendimento de que um site estático é uma página que não é gerada em tempo de execução de uma aplicação web, eles são criados por um programa chamado **static site generator**.

No momento do **build** da aplicação, uma chamada a API é feita e todos os dados são carregados, gerando assim todas as páginas de forma estática. A partir daí, nós subimos esses arquivos em um servidor.

Esse formato de renderização é o mais performático de todos apresentados neste conteúdo, pois nem o cliente e nem o servidor precisa renderizar o conteúdo, porém este modelo só pode ser utilizado em casos específicos, em que não tenha necessidade de consumo de dados com muita frequência e de forma dinâmica.








Obrigado!

Alguma Pergunta?

willian_brito00@hotmail.com
www.linkedin.com/in/willian-ferreira-brito
github.com/willian-brito



Refências

- ◇  <https://willianjusten.com.br/nextjs-gatsby-ou-create-react-app-entendendo-os-conceitos-de-ssr-ssg-e-spa> ◇
-  <https://woliveiras.com.br/posts/qual-diferenca-server-side-render-ssr-e-static-site-generator-ssg/#ArquiteturadeumaaplicaoSSG>
-  <https://blog.codecasts.com.br/single-page-applications-onde-vivem-e-o-que-comem-4fc9a44f3de>
-  <https://medium.com/techbloghotmart/o-que-%C3%A9-server-side-rendering-e-como-usar-na-pr%C3%A1tica-a840d76a6dca>
-  <https://www.youtube.com/watch?v=2LS6rP3ykJk&t=4300s>
-  https://www.youtube.com/watch?v=X3W-YFe2_io
-  https://www.youtube.com/watch?v=VM_ncQcQdmY