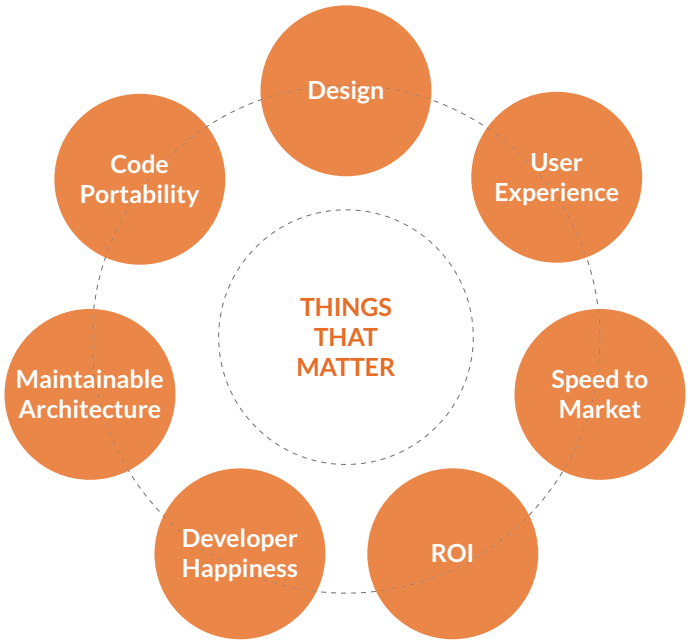
Diego Pacheco, Adrian Lemes, Lucas Veloso, Guilherme Guerra,
Thiago Paiva, Bárbara Becker - ilegra.com 2019

Microfront, SSR, GraphQL, BFF and WebAssembly e a Evolução de arquiteturas frontends para produtos digitais

Sumário

Quando pensamos em arquitetura de software, é comum acabarmos pensando em design patterns, estruturação de código ou no quanto o projeto está coberto de testes, não que não sejam tópicos importantes que devemos considerar, mas fazer software de qualidade e com uma arquitetura escalável envolve muito mais que isso.

Quando buscamos desenvolver produtos digitais que escalem e focados em qualidade, existem outros aspectos que devemos considerar, princípios que devemos seguir para escrever aplicações que realmente importam para o negócio, que resolvam um problema de fato e de forma eficaz. A imagem a seguir ilustra esses princípios:



Princípios de qualidade de software

Design e User Experience - Para construir produtos digitais de qualidade é importante termos o mindset focado no usuário, pois no final do dia é quem será a principal métrica de qualidade. Na ilegra, nosso foco é em desenvolver soluções customizadas para cada problema, que tenham o melhor fit para o negócio. O design da aplicação é centrado no usuário, pensado e desenvolvido para quem importa e quem irá dar as respostas que o negócio precisa.

Time to Market - Quanto mais curto for o ciclo de feedback do usuário, mais cedo podemos trabalhar em soluções e resoluções de problemas que resultem em uma melhor experiência de usuário. Speed to market é um aspecto importante a considerar na arquitetura de produtos digitais, buscando mecanismos que visam acelerar as entregas de uma aplicação focadas na experiência do usuário. É importante portanto garantir que as features que construirmos sejam entregues ao usuário de forma incremental e o mais cedo possível.

ROI(Retorno do Investimento) - Desenvolver software tem um custo alto, logo devemos garantir que todo investimento feito tenha um retorno. O ROI é um importante indicador de sucesso de qualquer arquitetura de software.

Developer Happiness/Experience - Achar profissionais qualificados para um projeto é um dilema de muitas organizações. É importante desenvolver uma arquitetura escalável e que seja motivo de orgulho por quem participa do desenvolvimento. Para isso é necessário prover um ambiente que favoreça o senso de progresso do time, através do desenvolvimento constante de features que agregam valor e que ajudem a empresa atingir os objetivos gradativamente.

Maintainable Architecture - A falta de uma arquitetura bem pensada pode ocasionar efeitos catastróficos para um projeto, resultando em desperdício financeiro, soluções com experiências ruins para o usuário final, escolhas técnicas inferiores que baixam a motivação da equipe e aumentam a chance de turn over. Ter uma arquitetura escalável é desenvolver algo que seja fácil de dar manutenção, onde desenvolver uma nova feature não impacte profundamente em outras e há várias formas de mitigar este problema. Gastar semanas e semanas para desenvolver uma feature em um projeto com baixa manutenibilidade é basicamente jogar dinheiro fora.

Code portability - Resolver os problemas que realmente importam é o objetivo da arquitetura de software. É importante desenvolver soluções que serão aproveitadas e possam inclusive ser usadas em outros projetos e partes do sistema, mas de forma consciente e planejada.

Baseado em alguns desses princípios a ilegra desenvolveu o modelo de arquitetura 4x4. Assim como descrito em Arquitetura valor e negócio (Pacheco, Diego. 2019), esse modelo de arquitetura visa alinhar o desenvolvimento de software com a visão de negócio. Através desse modelo, a ilegra como consultoria define quatro valores centrais e o cliente outros 4 valores, ajudando a entender porque certos movimentos tecnológicos são necessários na construção de produtos digitais.

Considerando os aspectos que tangem a criação de produtos digitais, o desenvolvimento do frontend é o mais próximo do usuário, é a parte tangível para quem está utilizando o sistema e é de extrema importância cuidarmos todos os aspectos que envolvem seu desenvolvimento. Nossos usuários a cada dia vem se tornando mais exigentes, pois não adianta ter o design mais elegante possível e não ter o que o usuário precisa. É importante termos a disposição recursos que nos possibilitam agregar valor ao negócio e conhecer esses recursos faz parte da arquitetura frontend, para podermos saber o momento certo de utilizá-los e como combiná-los quando necessário.

Nesse paper portanto, serão abordados assuntos que permeiam a construção de produtos digitais com foco no frontend, como single page application, utilização de micro-fronts, renderização via server side x renderização no client, agregação de dados com GraphQL, como fazer bom uso de Backend for Frontend (BFF) e diminuição dos custos do desenvolvimento de frontend através de códigos cross-platform, isto é, capazes de rodar na web e no mobile.

1 Single Page Application

Quando pensamos na construção de aplicações modernas para o frontend, logo pensamos em frameworks que trabalham no modelo Single Page Application (SPAs).

Mas o que são Single Page Applications?

Esse modelo de desenvolvimento frontend consiste em carregar e processar uma aplicação no browser do cliente através de templates, onde a única interação com o servidor é através de requests Ajax para buscar dados.

Aplicações nesse modelo ganharam tração devido a promessa de entregar uma melhor experiência e performance para o usuário, pois uma vez carregadas no browser é necessário apenas buscar dados do servidor, e não mais carregar layouts ou templates, deixando a experiência mais fluída e interativa.

Atualmente temos 3 principais players que trabalham no modelo single page application: React, Angular e Vue, onde o primeiro é apenas uma lib de construção de UI e os outros 2 são frameworks de fato.

Todos eles possuem um ecossistema que busca fornecer o necessário para a construção de uma aplicação de forma eficiente.

Podemos dizer que toda a aplicação que implementa o modelo SPA é em sua natureza um monolito. Essa abordagem de desenvolvimento para muitos cenários pode ser ideal, aplicações mais simples ou com ciclos de desenvolvimento mais curtos por exemplo. Porém para projetos mais complexos ou que tenham um ciclo de desenvolvimento mais longo pode não ser a melhor abordagem. A seguir elencamos os principais problemas que podem surgir no desenvolvimento de aplicações frontend utilizando SPA.

O primeiro problema que costumamos ver no modelo SPA é o acoplamento. Se dois times trabalhando no mesmo projeto precisam sincronizar mudanças, resolver conflitos de código e analisar impactos na mudança de um componente, provavelmente o acoplamento entre esses times é alto.

Quando há mais de um time ou desenvolvedores trabalhando na mesma codebase, com o passar do tempo e conforme a aplicação cresce, fica cada vez mais difícil de criar novas features. Cada pequena alteração requer que toda a aplicação seja deployada e, com isso aumenta a necessidade de uma alta cobertura de testes e de processos que visam controlar os possíveis impactos de uma nova atualização. Consequentemente, ao longo do tempo, esses problemas impactam de forma negativa, a produtividade do time.

Um outro problema comum do mundo de JavaScript é que constantemente tecnologias que são populares hoje, podem rapidamente se tornar obsoletas. Backbone, Jquery, DOJO, YUI são exemplos de tecnologias que tiveram um ciclo de vida curto. Outro caso mais memorável é o AngularJS, que veio para revolucionar a forma de lidar com o DOM, com o two way data binding, facilitando bastante o desenvolvimento e se popularizando rapidamente. Contudo, logo depois surgiu outros frameworks que lidavam com o DOM de maneira mais eficiente, utilizando conceitos de Virtual DOM e fazendo com que o AngularJS se tornasse obsoleto.

Quanto tempo de desenvolvimento e conhecimento foram jogados fora quando o AngularJS tornou-se obsoleto? Quão difícil é hoje encontrar profissionais capazes de dar suporte e manutenção a projetos existentes em AngularJS? Qual o custo de reescrever completamente do zero essas aplicações? Estes problemas comuns de JavaScript são resultado de uma arquitetura mal planejada de frontend, negligenciada e muitas vezes orientadas a hype e ao que é mais popular no momento.

Devemos trabalhar com uma arquitetura que possibilite uma evolução natural e orgânica ao longo do tempo e não deixe a aplicação defasada, necessitando de um grande esforço para refatorar. Assim como AngularJS que há 4 anos era um dos frameworks mais utilizados para o frontend e tornou-se obsoleto, o mesmo pode acontecer com os frameworks atuais. Por isso é importante buscarmos uma arquitetura que nos permita essa evolução gradativa.

Quando temos um monolito no frontend é muito mais difícil inovar. Como vou testar algo em produção ou uma tecnologia nova sendo que isso pode afetar todos os meus usuários? Como saber se o impacto de uma mudança será catastrófica?

Refatoração e reescrever código faz parte do processo Lean de desenvolvimento, porém em uma mesma organização, ter ciclos de reescrita de uma

aplicação por decisões ruins de design da aplicação ou porque surgiu um framework que resolve algo melhor, não é saudável para o negócio.

Para uma startup é perfeitamente normal reescrever sua base de código algumas vezes até ter um produto estável e que entregue valor, entretanto para sistemas corporativos de grandes empresas como bancos ou e-commerces, onde se tem uma expectativa de se manter no mesmo negócio por pelo menos uns 10 anos, reescrever a mesma homepage 3 ou 4 vezes nesse período é inviável e altamente custoso.

Não se preocupar com a arquitetura de software assim como descrito em Arquitetura valor e negócio (Pacheco, Diego. 2019), pode levar a vários problemas como:

- Sistemas que não escalam;
- Sistemas que não entregam uma experiência adequada ao usuário por cair a toda hora, não ter tolerância a falhas e nem entrega contínua e feedback constante;
- Criar novas features é um processo complexo e de avaliação de impactos;
- Impactar no custo da aplicação devido aos problemas de desenvolvimento.

Todos esses problemas são aspectos que norteiam o desenvolvimento de frontend e o uso de Single Page Applications com a abordagem de monolito é algo que deve ser repensado no aspecto de arquitetura frontend.

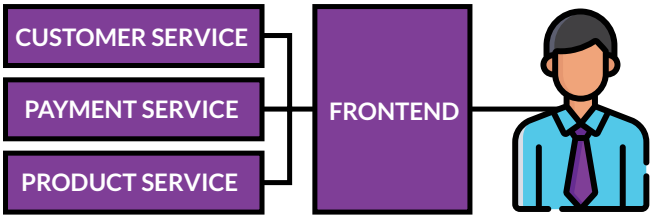
Microfrontend 2

Microservice é um conceito que ganhou tração em 2012, com o objetivo de promover uma divisão de uma aplicação complexa em serviços pequenos e autônomos trabalhando em conjunto. Essa abordagem foca em subdomínios do negócio de forma individual, promovendo isolamento entre os serviços, deploys separados e ownership de um time que consegue dominar um assunto e promover mudanças de forma mais assertiva.

Um outro aspecto importante a ser considerado em qualquer aplicação é relacionado a escalabilidade, onde escalar um monolito envolve uma complexidade gigante pois é um produto único, sendo que muitas vezes precisamos apenas escalar subsets da aplicação.

Algo comum em muitas empresas, é promover a divisão de times de backend por domínios do negócio mas ter um time único responsável pelo frontend. Mas como o foco da arquitetura deve ser em diminuir ciclos de feedback, entregar valor constantemente para o usuário final podendo melhorar o mais cedo possível, qual o valor de entrega de um microservice quando é necessário ter uma tela para consumi-lo?

Não adianta promovermos essa divisão de time em subdomínios se temos um monolito no frontend, pois com o tempo o frontend pode acabar se tornando um gargalo. A resposta padrão a este cenário é adicionar mais pessoas ao time de frontend, o que significa mais código e consequentemente aumentar o monolito. A cada nova feature que entra no monolito de frontend mais planejamento é necessário, maior a complexidade dos testes, aumentam as chance de aparecer bugs, torna-se necessário uma maior análise de impactos e consequentemente, menor é a flexibilidade a mudanças. Esse cenário acaba resultando em um ciclo de feedback cada vez mais demorado ao usuário final.

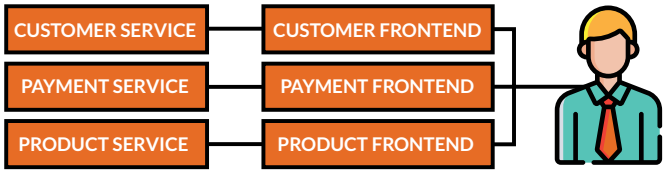


Abordagem monolítica no frontend

Uma arquitetura escalável deve permitir a entrada de mais desenvolvedores sem prejudicar a performance do time, deve permitir desenvolvimento paralelo com isolamento suficiente que evite que um time impacte o outro. É neste cenário que o microfront ganha tração.

Empresas que trabalham no modelo de microfrontend

Microfrontend é uma abordagem semelhante a microservices, porém no frontend. É uma arquitetura que vem sendo utilizada por empresas como Zalando, Hellofresh, Amazon, Spotify, Upwork, Allegro, entre outras. A ideia é dividir o desenvolvimento em pequenos times autônomos e isolados, responsáveis por apenas um domínio ou subdomínios do negócio, desenvolvendo e mantendo desde a base de dados até o frontend. Essa mudança de cultura incentiva ciclos de desenvolvimento separados, deploys isolados, menor impacto em outras partes do sistema e ownership por parte do time que se sente totalmente responsável pelo seu subdomínio, resultando em entregas mais rápidas, encurtando os ciclos de feedback com o usuário e consequentemente entregando uma melhor experiência.



Abordagem com microfrontend

Se o time de desenvolvimento não consegue entregar software de forma contínua, acaba por afetar diretamente a experiência do usuário.

É importante buscarmos constantemente inovar e trazer novas funcionalidades que agreguem na experiência do nosso usuário.

Através da utilização de microfronts é possível ter mais flexibilidade em inovar o produto de forma incremental sem impactar outras features, devido ao isolamento. Uma prática muito comum de melhoria contínua e feedback rápido do usuário, é testes A/B. Essa prática visa disponibilizar diferentes versões de partes do sistema, para verificar quais apresentam um maior engajamento e geram mais leads.

A arquitetura de microfront facilita a criação desses testes, pois no momento que diminui a complexidade de criar uma micro aplicação no sistema, facilita também criar outras versões dela.

2.1 Estratégias para arquitetura microfront

Ao optar trabalhar com uma arquitetura de microfront tem algumas assumptions que é necessário levantar, isto é, quais objetivos estamos buscando e o que queremos resolver. Elencamos algumas possíveis assumptions:

- Separation of concerns, poder ter aplicações separadas por domínio e responsabilidade única;
- Melhorias incrementais na construção de produtos digitais, que permite mais liberdade de analisar caso por caso sem se preocupar com o impacto em outras features;
- Processo de deploy e ciclo de vida de desenvolvimento separados;
- Apesar de não ser a principal vantagem utilizar diversas tecnologias, pode ser relevante a longo prazo, devido a possibilidade de adicionar ou alterar uma parte da aplicação feita em outro framework;
- Necessidade de isolamento entre as micro apps;
- Tamanho de bundle e compartilhamento de dependências;
- A produtividade de desenvolvimento.

É importante ter em mente que nem sempre é possível obter todas essas assumptions, por isso é necessário elencar quais são as mais importantes e o que o negócio necessita. As formas de criar microfront são variadas e tendo bem claro os objetivos que queremos atingir fica mais fácil selecionar uma arquitetura mais assertiva. Uma escolha errada neste momento pode trazer mais complexidade e dificuldade de manter a arquitetura e, como consultoria, é papel da ilegra avaliar esses pontos e fornecer opções que atendam o negócio como um todo. Um dos principais desafios de trabalhar com microfronts, vem no momento que buscamos isolamento entre as micro apps.

Mas do contrário de microservices, esse isolamento não existe quando a aplicação está no browser, primeiro porque a aplicação tem que ser uniforme, segundo que dependendo da abordagem podemos ter conflito de código javascript, classes de css, escopo, localStorage, entre outros problemas.

Ao optar por uma arquitetura de microfronts é importante avaliar os trade off da solução abordada e se realmente é necessário a utilização de microfronts. Devemos estar cientes de quais objetivos estamos buscando atingir com essa arquitetura. Trabalhar com uma solução que compõe uma UI a partir de múltiplas aplicações, traz diversas complexidades. É vital em uma arquitetura de microfronts ter uma cultura de DevOps, pois manter a solução em produção exige essa maturidade devido a complexidade que pode trazer.

Na ilegra a arquitetura de frontend anda lado a lado com DevOps, buscando usufruir dos benefícios de delivery continuous, observability, pipeline com testes, entre outras práticas. Outro ponto importante é que não necessariamente é preciso escolher apenas

uma abordagem, mas também podemos utilizar mais de uma e conseguir potencializar a solução porém, com isso é altamente provável um aumento da complexidade na arquitetura de frontend gerada.

A seguir será abordado as principais soluções de microfronts que temos atualmente.

2.1.1 Iframe

Quando o principal objetivo a ser atingido é o isolamento entre as aplicações, o uso de iframes possivelmente é a solução mais indicada. Ao utilizar iframes as aplicações rodam de forma standalone e são carregadas dentro de uma aplicação principal onde, para realizar a comunicação entre elas é necessário uma solução de mensageria como o postmessage.

Apesar da vantagem das aplicações estarem completamente isoladas uma das outras no browser, o tamanho do bundle será comprometido, pois com iframes não é possível compartilhar dependências entre aplicações durante as etapas de build.

Outro problema de iframe é quando necessitamos que a aplicação de dentro use scroll, podendo ser bastante dificultoso fazer o iframe funcionar.

Apesar dos tradeoff do uso de iframe, se justifica a escolha em alguns cenários de migração de código legado para uma arquitetura escalável. Neste cenário, quanto menor forem as modificações que fizermos na base de código legado, menor vai ser o custo dessa migração.

2.1.2 Single SPA

Single SPA é um meta framework criado pela empresa CanopyTax e mantido de forma open-source pela comunidade. Ele permite que aplicações construídas em diferentes frameworks possam co-existir como uma só no browser. O Single SPA tem suporte a AngularJS, React, Vue, Ember, Angular, entre outros frameworks.

Quando buscamos rodar múltiplas aplicações mantendo o comportamento de um SPA, isto é, sem a necessidade de dar refresh no browser, este framework se encaixa bem. Ele carrega as aplicações utilizando a prática de lazy-loading, que consiste em carregar código JavaScript aos poucos conforme a necessidade de uso pelo browser. Ele também possui um life cycle para cada micro app, que é responsável por decidir o que fazer com elas em diferentes momentos, como desativar quando não está em uso ou manter em espera para não ter que carregar novamente.

É uma abordagem muito boa e aplicável também em cenários de migração de código legado de forma incremental, mas para outros cenários pode ser complicado individualizar deploys, trabalhar com server side rendering nas aplicações ou garantir um isolamento bom de JavaScript.

Contudo, se não utilizado de maneira correta pode facilmente deixar a aplicação lenta e pesada no browser, principalmente quando carregamos três ou mais frameworks de uma vez só.

2.1.3 Mosaic stack

Mosaic é um conjunto de ferramentas criadas por uma empresa de e-commerce chamada Zalando, com objetivo de trabalhar com microservices no frontend para projetos de larga escala. Ele trabalha com conceitos de fragments que são múltiplas micro apps que em conjunto formam a aplicação. A composição de UI que eles fornecem é relativamente simples e permite um controle maior de cache e manipulação de requests. Segue um diagrama da arquitetura do Mosaic.

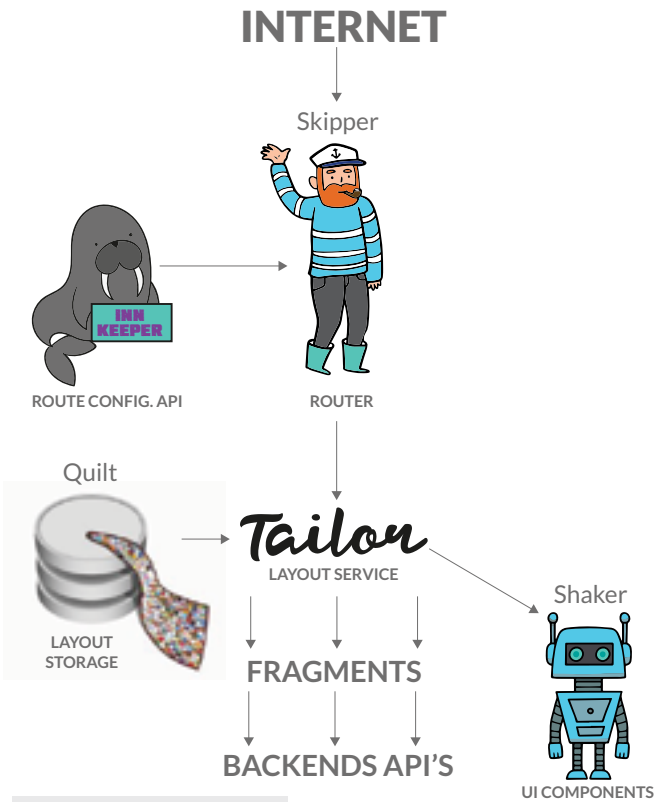


Diagrama da stack do Mosaic

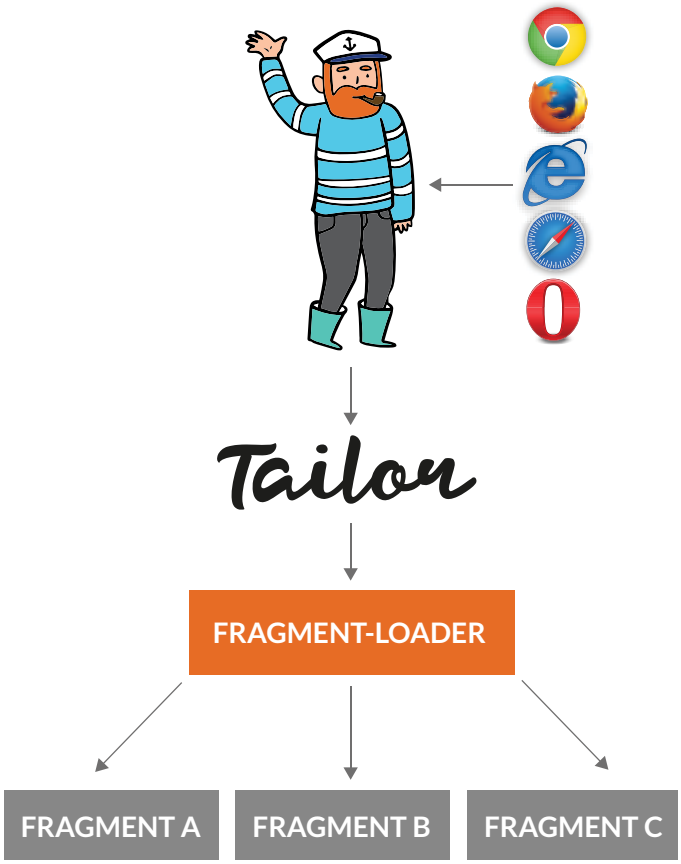
O Skipper e o Tailor são o coração dessa arquitetura, onde o primeiro é responsável por todo o roteamento das partes envolvidas e o segundo é responsável por montar templates através da junção de múltiplos fragments.

Na ilegra acreditamos em conceitos logo, não há necessidade de utilizar toda a stack do Mosaic. A junção das micro apps com Tailor, provavelmente é o mais atraente dessa stack, pois o isolamento das micro apps no browser é uma tarefa complexa de resolver. O compa tilhamento de componentes que possam ser utilizadospor

múltiplos fragments não necessariamente precisa ser o Shaker. É possível compartilhar com um registry próprio e instalar via Package.

O Mosaic é uma solução viável para utilização de microfrontends e é facilmente customizada para que se adapte às necessidades que o negócio precise. Um problema que o Mosaic apresenta, mas que pode ser contornado, é que cada fragment necessita de um servidor node para rodar e fornecer os assets. Devido a necessidade de fornecer alta disponibilizada para cada node, rodamos de 2 a 3 instâncias para cada fragment e a infra acaba ficando com um valor bem elevado, devido ao número de fragments e instâncias rodando simultaneamente.

Um trabalho que fizemos na ilegra, foi construir uma solução de arquitetura em cima dos fragments, responsável por fornecer múltiplas micro apps com um servidor apenas, diminuindo os custos de infra.



Utilização do Mosaic de forma customizada

Essa solução que criamos denominamos de fragment-loader. Nele podemos fornecer múltiplos fragments, tanto para ser resolvido via client-side quanto pre-renderizarmos cada fragmento via server side. Ao invés de ter um nodejs para cada fragment teremos apenas um fragment-loader ou teremos alguns fragment-loader dividido por domínio, que irá depender do cenário que queremos aplicar nossa arquitetura de microfrontend utilizando partes do Mosaic.

2.1.4 Web components

Web components é uma especificação no qual muitos dos frameworks front end se baseiam, porém criando sua própria implementação. Consiste no encapsulamento de uma estrutura semântica (HTML), comportamento (JavaScript) e layout (CSS) em uma tag HTML que possa ser reaproveitada em outros lugares.

Ela se baseia em padrões da web e vem sendo evoluída gradativamente em cima das especificações de HTML e DOM para que possam ser utilizada de uma forma genérica e independente de framework JavaScript. Web componentes se baseiam em 4 principais especificações:
Custom Elements **-** É um meio que permite construir e definir novos elementos a partir do DOM.

Shadow DOM **-** É responsável pelo encapsulamento do código que irá ficar no web component.

ES Modules **-** É o que permite a inclusão e reuso de código javascript de uma forma modular e performática.

HTML Template **-** É o que permite definir fragments de HTML que ficam inativos no load da página mas que podem ser carregados a qualquer momento.

Atualmente os principais frameworks tem algum tipo de suporte com web componentes, mas os mais evoluídos é Angular elements e Vue. Para uma utilização de web components deve ser levado em consideração a compatibilidade com o framework a ser utilizado e a necessidade de uso de polyfills para garantir a compatibilidade em todos os browsers visto que até o momento que esse artigo foi escrito, web components não é suportado de forma nativa em todos os browser. Um outro ponto de arquitetura que deve ser levado em consideração é que é necessário ter um projeto (shell app) que irá integrar todos esses web components e a cada update de uma micro app encapsulada em um web component, é necessário atualizar a versão no shell app.

2.2 Melhores Práticas para Arquiteturas Frontend

Trabalhar com microfronten ds como já citado anteriormente pode trazer diversos benefícios para projetos de larga escala, mas é necessário ter alguns cuidados para ser bem sucedido nessa abordagem. Por isso elencamos alguns pontos chaves que é importante nos preocuparmos Testes End2End - Para garantir a qualidade das entregas e que todas as micro apps irão funcionar de maneira adequada, testes E2E acabam ganhando uma relevância bem alta. Esse

tipo de teste é capaz de garantir regressão através da simulação, repetição e execução de comportamentos semelhantes ao do usuário.

Common dependencies - Quando múltiplas aplicações são carregadas no browser, devemos sempre nos preocupar com o tamanho do bundle. O reaproveitamento de dependências em comum entre as aplicações no browser se torna bem relevante e isso só é possível em algumas abordagens de microfrontend - Single SPA e Mosaic suporta, mas iframe não. É possível obter isso com a propriedade externals do webpack e carregar as dependências em comum ou via CDN ou de uma micro app especializada nisso.

Observability - Assim como uma arquitetura de microservices, garantir a interoperabilidade entre os serviços é uma tarefa complexa. Sem um sistema eficiente de logs e feedbacks de erros do sistema, a complexidade de manter essa arquitetura cresce consideravelmente.

Isolamento de CSS - Em algumas abordagens de microfrontend, quando chega no browser, o isolamento entre as aplicações se perde, por isso é importante utilizar o mínimo possível de CSS global e garantir de alguma forma o isolamento de CSS, seja através de name conventions com o uso de prefixos ou técnicas mais avançadas como CSS modules.

Promover features nativas - A complexidade de trabalhar com microfrontends é relativamente alta, então quanto mais promovermos o uso de features nativas do browser, mais fácil é de manter essa arquitetura. Ao invés de criar um sistema complexo de mensageria podemos utilizar custom events que é uma feature nativa do browser, por exemplo.

Devops e cloud native - É vital em uma arquitetura de microfronts ter uma cultura de devops, pois manter a solução em produção exige essa maturidade devido a complexidade que pode trazer. Na ilegra a arquitetura de frontend anda lado a lado com devops também, buscando usufruir dos benefícios de delivery continuous, observability, pipeline com testes, entre outras práticas.

2.3 Style Guide e interface coerente

Escalar o design em uma arquitetura de microfronts com times distribuídos é outro desafio a ser resolvido. Diferente de microservices, em microfrontend é necessário juntar as apps no browser, logo é necessário ter uma coerência e concordância de estilos. Por mais isolados e autonomia que um time tenha, é necessário ter uma comunicação e alinhamento ou pelo menos todos os times devem concordar em seguir um padrão.
Uma das formas de trabalhar para resolver esse problema é possuir um design system, que é uma iniciativa que a ilegra vem adotando em alguns projetos e tem cases significativos. A ideia do design system é ser um repositório de componentes e micro-fronts que possam compor as páginas e aplicações. o Design system pode ser uma estrutura colaborativa, onde todos os times podem contribuir para evolui-lo, facilitando com que coisas já testadas e validadas possam ser reaproveitadas de forma eficaz em outras partes do sistema.

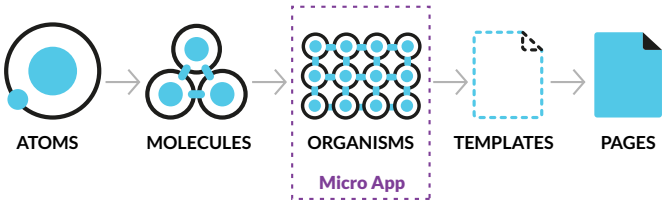
O design system deve ser uma ferramenta em constante mudança e melhoria de forma colaborativa, evitando criar a cultura de que o design system é a única fonte da verdade e limitando a criatividade dos times. É importante os componentes e style guides servirem apenas como ponto de partida, podendo ser evoluídos e retro alimentar o design system.

2.4 Granularidade de um microfrontend

Quando quebramos o front end adicionamos um overhead em cima de cada micro app. Devemos lembrar que components também compõe o frontend e sempre que possível devem ser compartilhados entre as partes do sistema.

Não existe receita de bolo na hora de decidir como fragmentar o frontend e cada caso deve ser analisado cuidadosamente.

Mas uma abordagem que podemos utilizar e que possa fazer sentido em alguns casos é através do uso de atomic design.



Atomic design com microfrontend

Atomic design é uma técnica utilizada para definir a granularidade dos componentes de um design system. Ela faz um paralelo como a forma que um organismo é montado, onde átomos é a menor granularidade, que quando agrupados formam moléculas e por sua vez organismos. Nesta divisão podemos definir os componentes em moléculas e as micro apps em organismos. Com isso conseguimos formar templates para serem aproveitados como páginas.

2.5 Complexidade

Martin Fowler descreve o termo Microservice Premium, onde menciona que a utilização de microservices pode trazer overhead e complexidade a arquitetura como um todo. Quebrar partes de um problema traz complexidades como eventual consistência, alta disponibilidade, lidar com múltiplos times, garantir comunicação, entre outros. Essa complexidade deve ser medida e somente quando ela for menor que o problema complexo que estamos tentando resolver é que vale a pena utilizar microservices.

Esse mesmo conceito pode ser considerado para a arquitetura de microfrontend. O processo de decompor em vários componentes, rotas, templates e delegar para diferentes sistemas, pode ser um overhead para aplicações de pequeno ou médio porte. É papel da ilegra como consultoria e especialistas em arquitetura de software analisar esses aspectos e escolher as soluções que melhor se adequa na resolução do problema.

Se precisamos de isolamento, deploy separado e independência entre times, single-spa, Mosaic e mesmo iframes são fortes candidatos, porém isso aumenta a necessidade do uso de um design system ou alinhamentos de design afim de oferecer uma identidade única para o usuário.

Se queremos flexibilidade de framework, poder trabalhar com tecnologia agnóstica, talvez web components seja a melhor pedida. É preciso analisar cada ponto e objetivos a serem atingidos para realizar a melhor escolha.

A utilização de uma arquitetura de microfrontends vem com um preço, a utilização dessa arquitetura pode gerar duplicação de dependências, aumentar o bundle necessário para o cliente baixar, impactar em problemas de UX e inconsistência de telas, entre outros problemas, mas na ilegra acreditamos que esses riscos possam ser mitigado e até contornado, conhecendo as ferramentas certas e sabendo quando aplicar a melhor solução para o problema certo.

3 The loading journey

Com a migração dos modelos de páginas estáticas renderizadas totalmente no servidor e a chegada das Single Page Application, a experiência do usuário melhorou consideravelmente, as interações ficaram mais dinâmicas e uma vez carregada a página, o usuário consegue ter quase que uma resposta imediata em sua navegação.

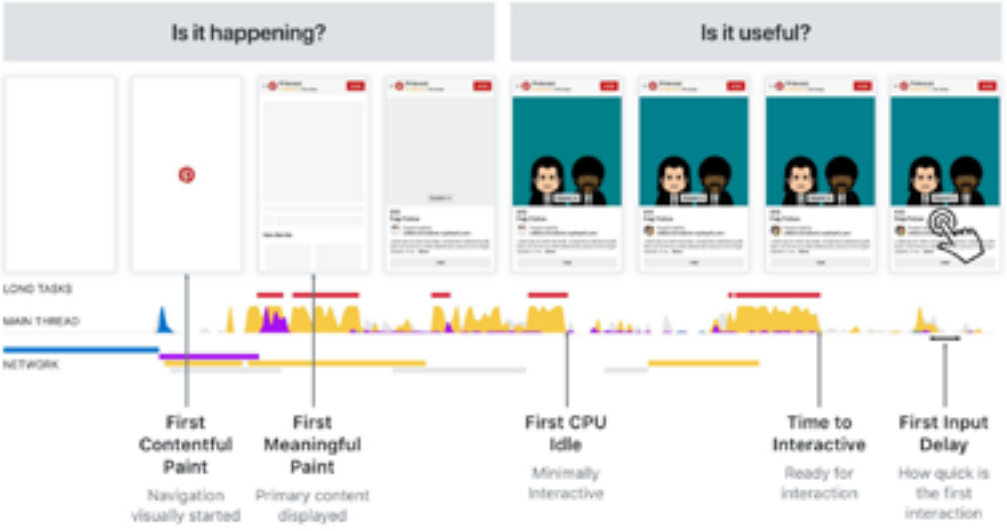
Desenvolver aplicações frontend envolve basicamente enviar código JavaScript ao browser, às vezes já pré-processado (SSR) ou para o cliente processar (CSR). Quanto maior for a eficiência de entrega deste código JavaScript, maior o impacto positivo na experiência do usuário.

A importância de entregar o JavaScript de forma eficiente, aumenta ainda mais quando consideramos que uma grande parte dos acessos da maioria dos websites é feita pelo mobile, que normalmente possui um processamento inferior que desktop e redes de dados de velocidade inferior a 4g.

O tamanho do bundle do facebook para desktop por exemplo, gira em torno de 7mbs e para mobile gira em torno de 2mbs.

Os sites atuais, normalmente fazem download de um framework JS (I.E. React, Vue ou Angular), alguma lib de gerenciamento de estado como mobx ou redux, polyfills afim de garantir a cross-compatibilidade, componentes de UI e libs de terceiro (como lodash, moment, etc). Quanto mais código JS carregarmos no browser ao mesmo tempo, maior vai ser o impacto negativo na experiência do usuário.

O carregamento de uma aplicação frontend é como um filme, é uma jornada que envolve 3 atos: It's happening? it's useful? It's interactive?



Etapas do carregamento frontend

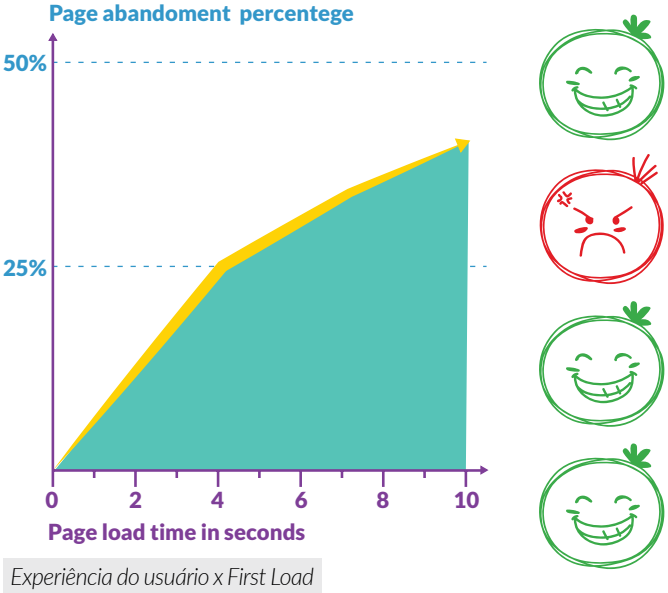
A etapa it's happening indica o tempo que o servidor demora para realizar a primeiro entrega ao browser, é a percepção do usuário de que algo está acontecendo.

A etapa it's useful diz respeito a entrega de algum conteúdo significativo que de alguma forma entregue valor.

E a etapa de it's interactive indica o tempo que o usuário leva para poder interagir com a página.

A medida responsável por indicar quanto tempo o usuário espera para conseguir interagir com a aplicação é o First Meaningful Paint, quanto tempo o usuário demora para poder interagir com o site? Existem estudos que apontam que quanto maior for esse tempo para o usuário, maior a chance dele desistir de acessar a página. As métricas do Google através da ferramenta Lighthouse apontam que o tempo adequado desse loading varia de 1.5s a 3s.

A Amazon realizou alguns estudos sobre o carregamento inicial da página e constatou que eles poderiam perder 1.6 Bilhões de receita em venda, se sua página demora-se 1s a mais para carregar.



Experiência do usuário x First Load

A Walmart por sua vez chegou a conclusão que a cada 100ms de otimização do carregamento inicial, impacta em 1% a mais de receita.

Outro exemplo importante relacionado ao first load vem do Pinterest que diminuiu o carregamento inicial de 23s para 5.6s resultando em um aumento nas vendas de 43% e em novos cadastros de usuário em 753%.

O carregamento inicial da página afeta diretamente a experiência do usuário, cada segundo a mais esperando aumenta a chance de perder lead. Existem diversas técnicas que podem ser utilizadas para melhorar o loading de uma aplicação:

Tree-shaking **-** Devemos estar constantemente preocupados com o código que entra e sai do projeto, essa preocupação aumenta ainda mais no uso de libs que visam acelerar o desenvolvimento do projeto. Nesses casos é importante sempre realizar a técnica realizada tree-shaking, que consiste em mapear para o bundle final somente o necessário, não sobrecarregando o browser. Não precisamos carregar toda uma library se precisamos de apenas uma parte dela.

Lazy loading **-** É uma técnica que consiste em dividir os fontes (JavaScript, CSS, Assets) da página em vários blocos e carregar no primeiro load apenas o necessário e ir carregando o resto desses blocos de forma assíncrona ou conforme o usuário for requisitando. Com isso carregamos só o que é necessário para o usuário no momento atual, o que garante uma melhora significativa de performance.

Above the first **-** Uma outra técnica comum para melhorar o engajamento do usuário é o Above the first, que consiste em priorizar o carregamento dos elementos de cima da página como Header, Call to Action, menu de navegação, que são os primeiros elementos visuais que o usuário irá ver.

Server Side Rendering ou SSR **-** é uma técnica que consiste em processar a aplicação web no lado do servidor, que tende a ter um poder de processamento maior que o cliente. Essa técnica traz também benefícios como SEO (Search Engine Optimization), que aliado a algumas técnicas consegue melhorar o rankeamento da página em sites de pesquisa. Quando trabalhamos com microfrontends ou temos um modelo de componentes eficiente aliados ao server side rendering, é possível ir disponibilizando elementos interativos pro usuário conforme estes são carregados. Ao invés dele esperar que toda a página carregue ele poderá interagir antes.

Todas essas técnicas podem ser utilizadas em conjunto ou separadas, com a arquitetura de microfrontends, agilizando o processo de carregamento de cada micro app da aplicação.

Portanto o carregamento de código JavaScript é de extrema relevância, principalmente em um cenário que temos tantos dispositivos móveis ativos pela web. Carregar dados no client também é uma necessidade de preocupação, fornecer somente os dados que o client precisa. Nesse âmbito a próxima seção irá abordar a camada de dados na construção de aplicações frontend.
Um outro importante aspecto da arquitetura de

front end é a camada de dados. Em um mundo cada vez mais competitivo garantir uma primeira interação do usuário no menor tempo possível (Time to First Interaction) é uma prioridade na entrega de usabilidade e experiência, o que torna seu produto mais competitivo e atraente para os usuários.

Não adianta desenhar uma boa experiência para seu usuário se sua arquitetura não permite que ela seja executada com excelência. Uma das formas de garantir isso é otimizar ao máximo a entrega dos dados da aplicação e fornecer mecanismos que tornem flexível a captura de informações oriundas de APIs e outras fontes de informação. Assim como diferentes clientes possuem diferentes requisitos, uma aplicação web que roda em um dispositivo móvel pode necessitar de dados diferentes de uma aplicação rodando em desktop.

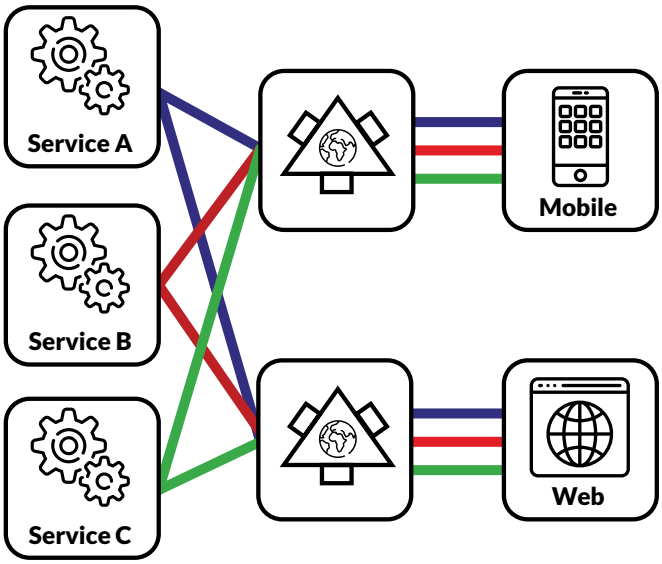
Esses e outros desafios podem ser resolvidos com o desenvolvimento de um BFF (Backend For Front end) e utilizando o GraphQL, mas para isso precisamos entender o que são cada um e para que servem. Entender que um não substitui o outro e que, na realidade, podem ser usados em conjunto. Vamos nos aprofundar mais nesses termos inicialmente para podermos ver eles funcionando juntos e quais são as suas vantagens e desvantagens, pois afinal, nada é uma bala de prata que irá resolver todos os problemas!

É importante entender o propósito de cada ferramenta e quais problemas que elas solucionam, onde eles vão potencializar a arquitetura e quando vão só adicionar mais complexidade. Conhecer as opções de arquitetura que podem ser utilizadas em conjunto ou separadas vai permitir extrair o máximo dessas tecnologias para seu negócio.

4.1 BFF (Backend For Front end)

Ao desenvolver aplicações de front end muitas vezes consumimos dados de serviços antigos ou de terceiros. Às vezes, por algum motivo, eles não podem ser alterados ou essa alteração pode impactar somente um cliente, por tanto, não faz sentido modificar o serviço para atender a essa demanda específica.

BFF é um pattern utilizado no desenvolvimento de frontend que tem como objetivo fornecer uma camada intermediária de serviço entre o cliente e as APIs de consumo, resultado em um maior isolamento das APIs, que passam a trabalhar orientadas ao negócio e ao domínio e não aos clientes que a consomem.



Exemplicação do uso de BFF

É comum que a linguagem utilizada pelo BFF seja a mesma linguagem utilizada pelo cliente. Se temos um cliente Android o BFF pode ser em Java ou Kotlin e se for IOS pode ser feito em swift por exemplo. Em uma arquitetura de microfronts podemos ter um BFF por módulo ou conjunto de micro apps.

O uso de BFF promove autonomia do time de front end, liberdade e flexibilidade para as equipes. Dessa forma cada equipe vai poder focar mais em entregas e no negócio uma vez que cada uma tem a liberdade e as ferramentas necessárias para realizar seus trabalho, isso permite que as interações entre equipes sejam mais objetivas e claras uma vez que muitos conflitos vindo das especificidades de cada uma foi isolado em seus respectivos BFFs.

Em diversos cenários é possível encontrar um código que não faz muito sentido ficar nos serviços, mas também não se encaixa nas aplicações de front end. Esse código acaba por ficar "órfão" ou seja, há alguns códigos e regras que podem ser melhor usados se ficarem em um meio termo entre serviços e front end, sendo assim, colocados em um BFF. Com isso, na maior parte das vezes uma regra que estava perdida acaba por encontrar um meio termo onde só interage e impacta o necessário.

4.2 GraphQL

Como dito anteriormente a camada de dados é um aspecto muito importante da arquitetura front end, em um mundo onde os dispositivos móveis são cada vez mais presentes se faz mais do que nunca necessário, pensar a quantidade de dado que se trafega.

O GraphQL é uma linguagem de consulta de dados que foi criada pelo Facebook que visa entre outras pontos muito importantes essa otimização dos dados trafegados. Em uma comparação, de um modo resumido, podemos dizer que o GraphQL pode ser comparado com o SQL, o SQL é utilizado para fazer consultas e manipulações em bancos de dados relacionais, já o GraphQL, é utilizado para consulta e agregação de APIs WEB.

Desse modo, assim como podemos utilizar o SQL em conjunto com qualquer outra linguagem para consultar dados, podemos utilizar o GraphQL em conjunto com qualquer outra linguagem para consumir APIs.

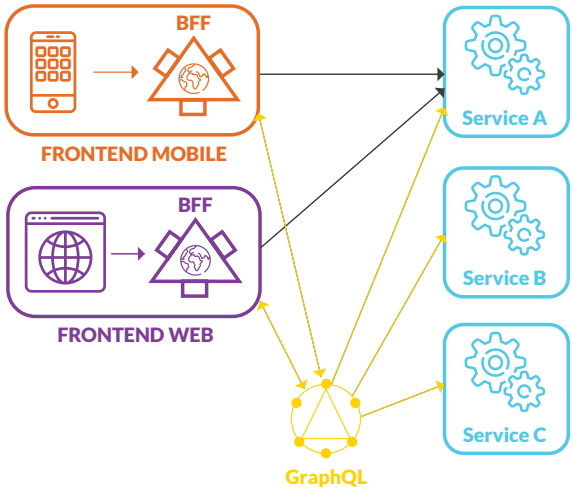
GraphQL tem sido muito relacionado a JavaScript (muitos desenvolvedores acham que só pode ser usado com JavaScript), mas não, GraphQL pode ser usado com muitas linguagens.

Um exemplo real de GraphQL, é o próprio Facebook (seu criador!). Imagine que para exibir a listagem de posts da sua linha do tempo fosse necessário consultar uma API, e dentro de cada post é necessário buscar todos os usuários que curtiram o post, todos os usuários que comentaram e seus respectivos comentários, e dentro de cada informação dessas tem que vir mais um monte de informações. Mas será que é necessário trazer todas as informações que irão vir nessa requisição? Como por exemplo, é necessário trazer a data de nascimento dos usuários, sendo que vamos utilizar apenas o nome, email e foto? A resposta seria não, para isso poderíamos pedir para o servidor não nos enviar esses dados (até por que vai ser economizado uma parte dos dados que são trafegados), e para evitar isso, poderíamos ter um BFF que se encarrega de fazer essa "conversão" da resposta para os clientes que irão consumir esses serviços, porém fazer isso de forma manual no BFF pode ser custoso.

E é nesse momento que o GraphQL entra em cena, ao invés de termos que ficar criando um BFF para cada estrutura de dados diferente e/ou ficar fazendo consultas para cada banco de dados e depois juntar tudo manualmente, não seria mais fácil dizer para um serviço o que é necessário de informações?

O GraphQL é uma linguagem de consulta que facilita e muito o nosso trabalho na hora de fazer requisições, pois tudo o que nós precisamos fazer é dizer quais os dados nós iremos precisar naquela consulta que ele se encarrega de trabalhar esses dados para nós.

Dessa forma em um cenário que é necessário consultar diversos endpoints de diversos serviços e ou preciso de apenas alguns dados de vários serviços o GraphQL encaixa como uma luva, agregando dados de diferentes fontes e entregando somente o necessário.



BFF e GraphQL são tecnologias diferentes mas que trabalham muito bem juntas. A proposta aqui é utilizar o GraphQL como um gateway e agregador das API's. Nesse caso o BFF fica responsável por direcionar o cliente para o GraphQL ou direcionar diretamente para um API ou endpoint específico caso não precise de agregação, dessa forma seguimos com o que foi proposto anteriormente, utilizando o GraphQL somente quando necessário para otimizar as requisições.

A principal vantagem dessa arquitetura é permitir que os serviços que compõem o backend possam ser construídos independentes dos clientes que estão consumindo e serem apenas orientados ao domínio, a resolver o negócio. Isso possibilita que possam ser consumidos pode quem está consumindo.

Desse modo, torna muito mais fácil e produtivo o desenvolvimento do lado dos times que desenvolvem os clients, pois para cada requisição que é feita, podem ser passados os dados que serão necessários naquela determinada parte do sistema, evitando assim o uso de várias requisições e consumo de dados que talvez não devesse ter, sendo assim, as aplicações que consomem os dados ganham mais performance e gastam menos dados.

Hoje o mercado frontend tem divisões claras com base nas plataformas cujo as aplicações desenvolvidas rodam. Existe o front end web com seus milhares de frameworks e peculiaridades de cada browser. Existe o front end mobile com os gigantes Android e iOS e suas respectivas linguagens e estruturas nativas. Existe o front end desktop que com o passar dos anos evoluiu para o uso de tecnologias mais diversas e existem ainda interface de outras possíveis plataformas: De TV a geladeiras, de smartwatch a painéis de carros.

Qualquer sistema hoje em dia que precisa apresentar ou interagir com o usuário passa pelo processo de criação de uma interface digital interativa. O front end está na base de muitas soluções por aí, mas da mesma forma que seu campo é diverso as tecnologias para aplicá-la também.

Por isso, o que acontece quando uma aplicação precisa rodar em mais de uma plataforma? O que acontece quando o meu negócio é único e precisa rodar na web, no Android e no iOS, por exemplo?

Esse é o cenário de muitas empresas nessa leva de transformação digital com a necessidade de atingir o maior público com seu produto. A solução padrão em muitos casos é desenvolver três aplicações distintas, nativas e até com times separados.

Não há nada de errado em distribuir um time nessa divisão de plataforma, na realidade só há ganhos ao desenvolver um aplicativo nativo em questão de performance e experiência. Mas ter três code bases com tecnologias distintas, infras distintas e demandando profissionais especializados tem um custo.

Naturalmente uma aplicação, cuja a interface e solução podem ou devem ser a mesma mas que acabam sendo divididas puramente pela limitação técnica, tende a ser um problema no ponto de vista empresarial. Assim surge a dúvida: existe a possibilidade de fazer essa aplicação apenas com um time? Com a mesma tecnologia? Com um code base compartilhado entre várias frentes?

"Write once, run anywhere", o antigo slogan do Java que já vendia muito bem sua característica cross-platform ilustra um dos desejos mais importantes do frontend moderno. O conceito de cross-platform se tornou o santo graal dessa área. Essa necessidade moveu a comunidade para a criação de soluções que atendessem o desejo. Muitos tentaram e poucos conseguiram. A missão não foi completa, mas a jornada é animadora.

Começamos falando em uma das soluções propostas no meio web que ajudam uma aplicação a ter uma experiência mobile agradável, o PWA.

Não é de hoje que aplicações web tem como requisito importante a responsividade. Acessar um site no browser do seu celular é algo comum mas sabemos que não tão usual para uso quanto um aplicativo. PWA (Progressive Web Apps) é uma metodologia de desenvolvimento fortemente difundida pelo Google, que adiciona a uma aplicação web uma séries de características que faz se assemelhar com um aplicativo nativo (Cache offline, Splash Arte, Ícone mobile, entre outras). Além da aplicação ser responsiva, ela pode ser instalada no celular, atualizar sozinha e enviar push notification. Obviamente ela não deixa de ser uma aplicação web que apesar de seus benefícios (SEO, linkable), pode deixar a desejar quando é necessário utilizar recursos nativos dos dispositivos móveis.

Essa abordagem permite a criação fácil de uma aplicação em tecnologia web garantindo uma boa experiência para usuários mobiles. Salvo alguns problemas ainda nesse uso na plataforma iOS, o PWA ganhou mercado e tende a ser uma boa alternativa de arquitetura front em alguns cenários.

Outra abordagem que tem ganhado força é o uso do Ionic para criação de aplicações híbridas mobile ou PWAs. O Ionic é um framework sobre o Cordova que utiliza Angular para desenvolvimento e que permite, com base na plataforma, acessar APIs nativas e fazer uso de recursos dos devices. Além disso, o Ionic possui uma série de componentes que permite, também com base na plataforma, assumir um look feel similar ao nativo.

Usando Ionic conseguimos criar uma aplicação híbrida que resolve muitos dos problemas que são levantados ao tentarmos buscar o sonhado cross-plataform, como codebase, tecnologia e times únicos. Além de garantir um experiência quase similar ao nativo, mas não igual. Uma aplicação mobile híbrida tem grandes diferenças de performance e até de acesso a recursos se comparado a uma solução nativa, o que nos leva a soluções mais modernas que se aproximam das plataformas alvos.

Indo para algo mais próximo do nativo temos o React Native, framework mantido pelo Facebook, é uma opção para a criação de aplicativos mobile usando javascript.

Com ele podemos descrever nossa UI com JSX fazendo uso de componentes já mapeados do environment mobile. O "bridge" da plataforma permite que esses mapeamentos renderizar UI elements nativos, ou seja, toda a performance e recursos desses componentes são idênticos ao uso da linguagem nativa.

Ele tem sido um dos framework mais populares no desenvolvimento cross-platform mobile graças a sua comunidade e facilidade de desenvolvimento, abstraindo a necessidade de conhecer a fundo o desenvolvimento específico de cada plataforma. Uma variante do React Native é o React Native Web que é uma lib que tem a proposta de gerar código web e mobile com a mesma code base. O código é escrito em React Native e no processo de compilação é gerado a versão web. O problema de fazer uso dessa biblioteca é que dependemos da comunidade começar a fazer tratamentos nos componentes de react-native para seu similar na web e que para podermos usar as libraries web precisamos que elas comecem a suportar essa visão cross-platform e fazer tratativas quando rodadas em plataformas mobile.

Outra tecnologia que surgiu não só como solução, mas que elevou o nível do desenvolvimento cross-platform mobile foi o Flutter, desenvolvida pelo Google.

Usando uma linguagem criada especificamente para seu uso chamada Dart, o Flutter compila seu código para binário nativo, além de se vender como uma solução extremamente rápida para o desenvolvimento de apps.

O Google tem tido uma abordagem diferente para Flutter em comparação com seus concorrentes. Em maio deste ano eles anunciaram oficialmente o interesse de expansão da solução para outras plataformas. Flutter for web, desktop e embedded devices revelou o interesse do framework de compilar seu código Dart para todas as plataformas, um marco importante para o sonhado cross-platform completo.

E como dito anteriormente, desenvolver aplicações nativas é um ótima abordagem pois todos os recursos da plataforma são disponíveis e a performance, fluidez e experiência são elevadas ao máximo que o ambiente permite. Mas o mercado cross-platform frontend evoluiu muito e temos hoje muitas possibilidades e níveis de abstração. Temos recursos de fácil e rápido desenvolvimento, tecnologia que conseguem atingir o nativo e por vezes se igualar.

WebAssembly and Beyond 6

Quando falamos de front end é indiscutível a presença e importância do Javascript. Criado em 1995 por Brendan Eich, essa linguagem passou por muitas mudanças e influenciou todo o mercado. Na web sua influência é muito clara e tem um longo histórico, mas hoje em dia o Javascript atinge o mobile, desktop e outras plataformas capazes de renderizar interfaces.

A linguagem foi criada em poucos dias e com um propósito restrito, mas tomou forma e se expandiu incontrolavelmente. Um dos motivos foi que em 2008 o Google lançou a engine V8, que além de permitir o uso da linguagem em outras plataformas mudou o formato como ele é processado.

Antes, ele era uma linguagem de script puramente interpretada, hoje a engine consegue aplicar a compilação just-in-time (JIT) sobre o código e gerar uma série de melhorias performáticas.

Basicamente no V8 o código javascript é carregado e transformado em uma árvore chamada Abstract Syntax Tree (AST). Esse será o código a ser utilizado nas operações a seguir bem como a nível de interpretação.

A engine então realiza um trabalho de classificação dos trechos do código. Se um trecho do código for interpretado muitas vezes ele se torna warm. Ele passa então pelo baseline compiler que, dependendo do SO/plataforma, gera uma versão compilada do trecho com uma certa otimização, reduzindo o tempo de interpretação.

Se esse código warm for executado muitas vezes ele se torna hot e entra em ação o optimizer compiler. Ele vai gerar uma versão mais otimizada do código usando algumas coisas como premissas, como o tipo das variáveis ou formato dos objetos.

Se uma dessas premissas for quebradas iniciasse a fase de deoptimization. Basicamente a versão otimizada é jogada fora e o trecho volta a ser executado com a versão baseline ou até interpretada.

Toda essa fase de otimização garante que o javascript consiga ter um nível performático aceitável e garantiu a ele tomar o espaço que tem hoje no mercado frontend.

Mas hoje quando paramos para analisar as soluções do mercado frontend o javascript ainda sofre um pouco em certas operações. Mesmo ele tem suas peculiaridades e limitações que gargalam em performance.

Sabendo disso, podemos nos perguntar:
Quando chegamos neste estágio, temos hoje alguma solução? Existe algo no mercado que ajude nossas plataformas frontend a superar isso? A resposta é sim, e ela se chama WebAssembly (WASM).

O wasm é um compiler target (o código que o compilador gera) de formato binário, que roda códigos como C, C++ e Rust no navegador com uma performance bem próxima a de um código nativo. Ou seja, é uma forma de criar códigos performáticos fazendo uso de linguagens compiladas/tipadas ao lado do Javascript.

Isso mesmo, ao lado. Podemos carregar um código wasm usando uma API no javascript e compartilhar funcionalidade entre os dois.

Código wasm são por definição mais performáticos que código javascript, mas isso não é verdade 100% das vezes. Conhecendo muito bem a engine javascript que seu código vai rodar você pode traçar uma série de melhorias performáticas e usufruir de todos os benefícios do JIT como explicado anteriormente, mas isso é algo complexo e hoje em dia os browsers diferem muito suas engines, tornando naturalmente o processo mais difícil.

O WebAssembly no processo de processamento no browser tem uma série de benefícios quando comparado com o processo do Javascript. Basicamente ele tem bundle menor, realiza a decodificação antes mesmo do javascript parsear, possui otimizações na pré-compilação e compila em stream algumas partes.

Visualizamos o wasm como uma solução clara para os glitches performáticos que muitas aplicações podem ter com javascript e hoje seu uso é uma realidade. Mas e o futuro? Existem melhorias e mudanças vindo aí?

A primeira coisa que apostamos é a adoção de novas linguagens compilando para wasm. Hoje temos apenas C, C++ e Rust. Provavelmente com a adição de novas linguagens a influência do wasm cresça na comunidade e amplie as formas de atuação.

Outro grande passo que está sendo tomado é o WebAssembly System Interface (WASI). Basicamente uma iniciativa de padronizar uma interface capaz de rodar códigos wasm tendo dois conceitos importantes: A portabilidade e a segurança. Ou seja, compilando seu código para rodar em wasi o código é capaz de rodar em múltiplos devices, plataformas e sistemas operacionais bem como não quebrar qualquer limitante de segurança da plataforma alvo.

Essa iniciativa é de grande importância para o mercado frontend, afinal trás indícios de que estamos se aproximando de uma solução cross-platform completa, o futuro do mercado frontend.

Referências

https://www.quora.com/What-is-the-technology-stack-behind-the-Spotify-web-client/answer/Andreas-Blixt

https://engineering.hellofresh.com/front-end-microservices-at-hellofresh-23978a611b87

https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

https://www.upwork.com/blog/2017/05/modernizing-upwork-micro-frontends/

https://martinfowler.com/articles/micro-frontends.html#Downsides

http://www.agilechamps.com/microservices-to-micro-frontends/

https://micro-frontends.org/

https://github.com/vuza/micro-frontends-docu

https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

https://gustafnk.github.io/microservice-websites/#example-architecture

https://www.softwarearchitekt.at/aktuelles/a-software-architects-approach-towards/

https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154

https://blog.logrocket.com/the-state-of-react-native-web-in-2019-6ab67ac5c51e/

https://medium.com/flutter-community/ins-and-outs-of-flutter-web-7a82721dc19a

https://developers.googleblog.com/2019/05/Flutter-io19.html

https://hackernoon.com/getting-started-with-cross-platform-app-development-in-2019-dd2bf7f6161b

https://codeburst.io/native-vs-cross-platform-app-development-pros-and-cons-49f397bb38ac

https://medium.com/@areai51/microfrontends-an-approach-to-building-scalable-web-apps-e8678e2acdd6

http://atomicdesign.bradfrost.com/chapter-2/

https://medium.com/trainingcenter/webassembly-a-jornada-f5aec56c507f

https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

https://500tech.com/blog/all/why-do-small-frontend-teams-fail

Porto Alegre +55 51 3212-8444 | São Paulo +55 11 3171-0555

comercial@ilegra.com
www.ilegra.com