willianjusten.com.br

Como fazer seu site funcionar offline com PWA

Autor Willian JustenDesenvolvedor Front Endgithub.com/willianjusten

14-19 minutos

Índice

Introdução

Fala pessoal, continuando a série sobre Performance Web, hoje vou falar sobre um assunto que tem bombado bastante, principalmente pelo suporte que a Google dá para essa iniciativa, que são os Progressive Web Apps, mais conhecido como PWA. Você vai ver que é tãaaaao fácil transformar seu site num PWA que nem vai acreditar que não tinha feito até hoje.

Enquanto vou escrevendo, vou ouvindo uma playlist no Spotify chamada Relax & Unwind, é bem calminha e está sendo útil para mim hoje.

A ideia desse post é fazer uma pequena introdução sobre o conceito e os passos para fazer no seu site. Como é um assunto bem amplo, eu vou colocar vários links ao decorrer do post, para você ver mais a fundo os detalhes também.

Mesmo assim o post deve ficar um pouquinho grandinho, mas não tenha preguiça não xD

O que é Progressive Web App?

Aqui tem o link oficial da Google sobre PWA, mas podemos resumir em:

> Progressive Web Apps são experiências que combinam o melhor da Web e o melhor dos aplicativos.

E para ser considerado um PWA, ele precisa seguir os seguintes requisitos:

- **Progressivo** - Funciona para qualquer usuário, independentemente do navegador escolhido, pois é criado com aprimoramento progressivo como princípio fundamental.
- **Responsivo** - Se adequa a qualquer formato: desktop, celular, tablet ou o que for inventado a seguir.
- **Independente de conectividade** - Aprimorado com service workers para trabalhar off-line ou em redes de baixa qualidade.
- **Semelhante a aplicativos** - Parece com aplicativos para os usuários, com interações e navegação de estilo de aplicativos, pois é compilado no modelo de shell de

aplicativo.

- **Atual** - Sempre atualizado graças ao processo de atualização do service worker.
- **Seguro** - Fornecido via HTTPS para evitar invasões e garantir que o conteúdo não seja adulterado.
- **Descobrível** - Pode ser identificado como “aplicativo” graças aos manifestos W3C e ao escopo de registro do service worker, que permitem que os mecanismos de pesquisa os encontrem.
- **Reenvolvente** - Facilita o reengajamento com recursos como notificações push.
- **Instalável** - Permite que os usuários “guardem” os aplicativos mais úteis em suas telas iniciais sem precisar acessar uma loja de aplicativos.
- **Linkável** - Compartilhe facilmente por URL, não requer instalação complexa.

  Aí você pode ouvir algo como:

  > “Nossa, tudo isso? Deve ser difícil para caramba fazer isso! Vou fazer não.” - Sobrinho

  Mas fique tranquilo, segue o post que eu vou explicar o passo-a-passo e é mais fácil do que você imagina.

O checklist para se ter um PWA

A Google criou uma checklist para ter um PWA que diz o mínimo necessário para ser considerado um PWA e também algumas coisas a mais que podem ser interessantes.

E o Lighthouse, que é aquela ferramenta que vimos nesse post usa essa checklist para analisar seu site para ver se ele está de acordo ou não. Vai ser através dessa lista que iremos partir =D

1 - Use HTTPS e redirecione sempre para HTTPS

O HTTPS é uma implementação do protocolo HTTP já conhecido, sobre uma camada adicional de segurança que utiliza o protocolo SSL/TLS. Essa camada adicional permite que os dados sejam transmitidos por meio de uma conexão criptografada e também é feita uma verificação de autenticidade do servidor e do cliente por meio de certificados digitais.

Se você estiver usando o Github Pages, eu escrevi um post ensinando como fazer isso. Se você usa outro servidor, verifique como habilitar o SSL no seu domínio, isso é uma regra **obrigatória**. E, por favor, lembre de habilitar o

redirecionamento 301 de `http` para `https`, senão as pessoas vão acabar podendo acessar através dos 2 protocolos e a Google pode acabar até punindo por considerar conteúdo copiado.

2 - Tenha um site Responsivo e Rápido

Se a ideia é dar uma interação como se fosse de um app, isso é uma regra mais que fundamental. E dentro dela, não se esqueça de definir a meta tag correta para a viewport:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Sobre performance, só ficar ligado na série Performance Web e seguir as dicas que o seu site com certeza irá passar.

3 - Tenha uma cor tema no site

Eu escrevi um post sobre essa Theme Color lá em 2015, na época era meio que uma novidade para quem usava Android Lollipop, mas hoje funciona em todos os Androids atuais e também é uma regra. E para fazer isso é só adicionar essa meta tag:

```
<meta name="theme-color" content="cor em hexadecimal">
```

4 - Tenha um Manifesto com informações do seu site

O que significa isso? É um arquivo chamado `manifest.json` que vai conter informações relevantes do seu site, permitindo que o browser entenda que seu site é um PWA e vai inclusive habilitar uma mensagem para o usuário para que ele possa instalar o site na home do celular, o que faz com o site fique com um visual igual de um app mesmo.

E como esse `manifest.json` se parece? Segue abaixo o exemplo do meu blog (sem todos os ícones):

```
{
  "name": "Willian Justen Blog",
  "short_name": "WJusten",
  "theme_color": "#005f97",
  "background_color": "#005f97",
  "display": "standalone",
  "scope": "/",
  "start_url": "/?utm_source=homescreen",
  "lang": "pt-BR",
  "orientation": "any",
  "icons": [
    {
```

```
      "src": "/assets/img/icons/favicon-
512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Esse `manifest.json` possui algumas regras muito importantes que são:

- O `short_name` não pode exceder 12 letras, pois é o nome que vai aparecer abaixo do ícone do app.
- O `background_color` é obrigatório.
- O array de `icons` precisa ter pelo menos um ícone de `512x512`.
- O ícone precisa ser em `png`.

Existem sites que fazem a criação para você:

- App Manifest
- PWA Builder

Essa configuração serve para 2 momentos. O primeiro é para a criação do ícone do site igualzinho de um app. E o segundo momento é no momento de abertura do site, que

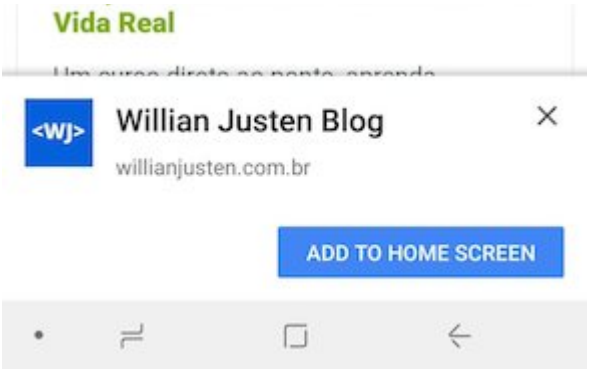
vai mostrar uma splash screen se tudo estiver configurado corretamente.

Repare que eu tenho também a propriedade `"start_url": "/?utm_source=homescreen"`, que serve para podermos verificar através do nosso Analytics se a página foi aberta via app, o que é bem útil para quem é doido de dados, como eu.

Com o `manifest.json` criado corretamente, não esqueça de adicionar no `head` do projeto a seguinte linha:

```
<link rel="manifest" href="/manifest.json">
```

Tudo estando certinho, se você tiver visitantes assíduos, é possível que eles vejam uma telinha como a seguinte:



5 - Registre um Service Worker e responda 200 quando offline

É aqui onde começa a brincadeira legal e que representa o título! E para essa parte, eu vou separar um pouquinho, pois é um assunto com mais detalhes.

O que é um Service Worker?

Um **Service Worker** é um script que seu navegador executa em segundo plano, separado da página da Web, possibilitando recursos que não precisam de uma página da Web ou de interação do usuário. Atualmente, os service workers já incluem recursos como notificações push e sincronização em segundo plano. No futuro, os service workers permitirão outras ações como sincronização periódica ou geolocalização. O que mais vamos discutir aqui é a capacidade de interceptar e tratar solicitações de rede, respondendo com um cache caso tivermos.

Eu vou explicar algumas coisas de forma básica, mas tem

um curso **MARAVILHOSO** e **GRÁTIS** lá na Udacity feito por nada menos que o Jake Archibald. Sério, se você se interessou pelo assunto, faz o curso, ele é rápido, leve e muito didático!

Informações Importantes do Service Worker

- O Service Worker funciona numa thread separada no browser, com isso não tem acesso ao DOM.
- O arquivo do SW precisa sempre ter o mesmo nome e ficar sempre no mesmo lugar. Caso contrário, vai gerar uma duplicação de Service Worker.
- O arquivo do SW não pode cachear de forma alguma, senão ele pode agir contra você e deixar um cache infinito na máquina do usuário. Porém não use `sw.js?hash_aqui`, pois será considerado outro SW. O certo é dentro do seu servidor você definir o max-age e colocar para sempre carregar de novo, sem cache.
- Fique atento ao ciclo de vida do SW, ao mesmo tempo que ele ajuda, também pode atrapalhar. Lembre sempre de deletar o cache antigo quando atualizar algo no site.

  Existe esse post do Jake Archibald que é bem completo e explica sobre os ciclos de forma bem detalhada. Abaixo eu

vou mostrar de forma simples e rápida como construir seu Service Worker do zero.

Detectando se o Browser suporta

O site Is Service Worker Ready? mostra os browsers que já suportam e quais coisas que já funcionam. E como você pode ver, nem todos os browsers suportam 100% ainda, então, nós fazemos um snippet bem simples.

```
if ('serviceWorker' in navigator) {
  console.log('ServiceWorker é suportado, vamos usar!');
} else {
  console.log('ServiceWorker não é suportado.');
}
```

Registrando

Com o snippet dali de cima que já faz a verificação, precisamos registrar o nosso arquivo de Service Worker para ele poder começar a funcionar. Coloque esse registro no head do site, pois é muito importante inicializar o worker antes de tudo.

```
if ('serviceWorker' in navigator) {
```

```
  navigator.serviceWorker.register('sw.js')
    .then(reg => console.info('registered
sw', reg))
    .catch(err => console.error('error
registering sw', err));
}
```

Ciclo de Vida

O Service Worker trabalha numa estrutura já determinada de ciclo de vida. Entendendo como funciona cada evento e suas respostas é a melhor forma de atualizar e cachear os arquivos.

O Service Worker possui então as seguintes etapas:

- install
- activate
- fetch
- message
- sync
- push

Eu vou cobrir somente os 3 primeiros eventos, que são os necessários para fazer nossas páginas funcionarem em

modo offline. Então, mãos na massa, vamos criar o arquivo `sw.js` na raiz do nosso site e ir preenchendo.

Install

O evento de `install` é ativado somente uma vez, quando você registra a versão do `sw.js` pela primeira vez. Se o `sw.js` muda uma única coisa, o `install` é chamado novamente. Use esse evento para preparar tudo que seja necessário. Abaixo eu vou mostrar um exemplo com alguns detalhes que fiz para o meu caso, que é em Jekyll:

```
---
layout: null
---

const staticCacheName = 'willian-justen-
2019-03-11-14-09';

const filesToCache = [
  { % for page in site.pages_to_cache % }
    '{ { page } }',
  { % endfor % }
  { % for post in site.posts limit: 6 % }
    '{ { post.url } }',
```

```
  { % endfor % }
  { % for asset in site.files_to_cache % }
    '{ { asset } }',
  { % endfor % }
];


this.addEventListener("install", event => {
  this.skipWaiting();

  event.waitUntil(
    caches.open(staticCacheName)
      .then(cache => {
        return cache.addAll(filesToCache);
    })
  )
});
```

Bom, primeiro de tudo, eu defini um front matter ali acima como `layout: null`, para que eu pudesse utilizar o liquid template e chamar algumas variáveis do meu Jekyll.

E aí entram coisas importantes em qualquer site, não só sites em Jekyll. Precisamos definir uma variável para nomear o nosso cache. Que é o caso do

`staticCacheName`. Como você pode ver, eu passo o tempo de quando o site foi gerado, o que me garante que é um valor único e para aquele build. Precisamos disso, pois se o valor não mudar, o Service Worker nunca irá atualizar o cache, caso a gente atualize.

Definido o nome do cache, precisamos definir quais os arquivos que vamos salvar! No meu blog, eu defino algumas variáveis no meu _config.yml e também itero para pegar os últimos 6 posts do meu blog.

Activate

O evento `activate` é ativado somente uma vez também, quando uma nova versão do `sw.js` foi instalado e não tem nenhuma versão anterior rodando em outra aba. Então você basicamente utiliza esse evento para deletar coisas antigas de versões anteriores.

```
this.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames
          .filter(cacheName =>
```

```
(cacheName.startsWith('willian-justen-')))
          .filter(cacheName => (cacheName
!== staticCacheName))
          .map(cacheName =>
caches.delete(cacheName))
      );
    })
  );
});
```

Verifique que eu filtro o cache para ver se ele inicia com `willian-justen-` que é o que eu defini no nome do meu cache e também se o nome é diferente. Caso o nome seja diferente, eu deleto os arquivos antigos. Isso é **muito** importante, cache é a coisa mais braba do Service Worker, se você não deletar corretamente, o usuário vai estar vendo coisas velhas.

Fetch

O evento de `fetch` é ativado toda vez que uma página é requisitada.

```
this.addEventListener("fetch", event => {
  event.respondWith(
```

```
    caches.match(event.request)
      .then(response => {
        return response ||
fetch(event.request);
      })
      .catch(() => {
        return caches.match('/offline
/index.html');
      })
  )
});
```

O `fetch` irá verificar se o arquivo requisitado contém no cache e caso sim, ele retorna o arquivo direto do cache. E uma coisa muito legal é que se o arquivo não existir, você pode redirecionar para alguma página, no meu caso eu criei uma página offline, onde tem um joguinho e uma lista para os 6 posts salvos. Você pode clicar no link acima para ver a página ou ficar offline e tentar abrir alguma página bem antiga, como essa aqui. E abaixo segue a imagem da página:

Yeeeey! Agora nosso site salva algumas páginas e também tem redirect para um joguinho super legal caso não tenha aquele arquivo salvo! Se quiser, aqui está o arquivo completo do `sw.js`.

Conclusão

É isso galera! Espero que tenham curtido o post, eu tentei fazer algo mais simples e rápido, mas não sem passar os passos necessários para você ter sua PWA. Espero que se divirtam com o joguinho também! Nos próximos posts irei falar um pouco sobre o Netlify, não, não é aquela plataforma de séries que você assiste todo dia, mas é algo tão legal quanto!
