



Aprenda com quem faz

Tópicos Especiais em Desenvolvimento Front End

Prof. Danilo Ferreira e Silva

2022



SUMÁRIO

Capítulo 1. Introdução	5
Visão geral do módulo	5
Ferramentas de desenvolvimento.....	5
Capítulo 2. Web Components (Parte 1).....	8
Autonomous custom elements.....	9
Customized built-in elements.....	9
Estilização no Shadow DOM	10
Ciclo de vida do Web Component.....	10
Capítulo 3. Web Components (Parte 2).....	13
3.1. HTML Template	13
Slots	14
Web Components encapsulados em módulos.....	15
Capítulo 4. SVG.....	17
Figuras básicas.....	17
O elemento path.....	18
Transformações	19
Capítulo 5. Canvas	22
Desenhando no Canvas.....	22
Transformações	23
Desenhando imagens	24
Capítulo 6. Armazenamento local.....	26
localStorage.....	26
sessionStorage.....	27
Capítulo 7. Geolocalização	29
Capítulo 8. Captura de áudio e vídeo.....	31

Capítulo 9. Notificações	33
Capítulo 10. Svelte.....	35
Criando um projeto.....	35
Criando um componente	35
Reagindo a mudanças no estado.....	36
Comando condicionais e de repetição	36
Manipulação de formulários.....	37
Comunicação entre componentes	38
Declarações reativas.....	38
Capítulo 11. Introdução a Micro Frontends	41
Motivação.....	41
Desvantagens	42
Estratégias de integração de Micro Frontends.....	42
Referências	46



XPe

> Capítulo 1



Capítulo 1. Introdução

Este capítulo dá uma visão geral do módulo Tópicos Especiais em Desenvolvimento Front End e descreve as ferramentas de desenvolvimento necessárias para acompanhar o curso. É importante destacar que esta apostila não contém todo o conteúdo das videoaulas, mas apenas informações de referência rápida, dando apoio as mesmas.

Visão geral do módulo

Neste módulo, veremos diversos assuntos relacionados como desenvolvimento Front End:

- Web Components.
- Scalable Vector Graphics (SVG).
- A API Canvas.
- Armazenamento local.
- Geolocalização.
- Captura de áudio e vídeo.
- Notificações.
- O framework Svelte.
- Arquitetura Micro Frontends.

Ferramentas de desenvolvimento

Para acompanhar o módulo, é importante ter instalado as seguintes ferramentas já utilizadas em módulos anteriores:

- Visual Studio Code.

- Node.js.
- Navegador Google Chrome.

Além disso, utilizaremos a ferramenta live-server, que é um servidor web voltado para o desenvolvimento. Para instalar o live-server, usaremos o *npm*, o gerenciador de pacotes do Node.js. Execute o comando abaixo no prompt de comando:

```
npm install -g live-server
```

Feito isso, o executável live-server estará instalado globalmente e disponível no prompt de comando. Por exemplo, supondo que você execute:

```
C:\meu\diretorio>live-server
```

O servidor iniciará no diretório corrente, servindo os arquivos presentes no mesmo ou em subdiretórios. Além disso, ao fazer alterações nos arquivos, a página aberta no navegador se recarrega automaticamente, o que ajuda a tornar o desenvolvimento mais fluido (esse recurso é conhecido como *live reload*). Mais detalhes sobre tal pacote estão disponíveis em:

<https://www.npmjs.com/package/live-server>



XPe

> Capítulo 2



Capítulo 2. Web Components (Parte 1)

Web Components é um padrão composto por um conjunto de tecnologias padronizadas pelo W3C que permite desenvolver elementos customizados reutilizáveis para a plataforma web. Utilizando tais tecnologias, é possível desenvolver componentes sem depender de frameworks ou de bibliotecas externas. As tecnologias que compõem tal padrão são descritas a seguir:

- Custom elements: API para registrar e descrever elementos customizados.
- Shadow DOM: API que permite criar uma árvore de elementos “ocultos”, ou seja, privados ao elemento.
- HTML Templates: Permitem gerar marcação HTML que não é diretamente renderizada na página, mas sim referenciada e utilizada quando necessário pelo seu elemento.

Web Components são definidos por meio de uma classe e devem ser registrados via `customElements.define`. Além disso, opcionalmente, podemos construir a estrutura do elemento usando Shadow DOM. A seguir, temos um exemplo de um Web Component:


```
class HelloWorldElement extends HTMLElement {  
  constructor() { super();  
  
    this.attachShadow({ mode: "open" });  
    const span = document.createElement("span");  
    span.textContent = "Hello world!";  
    this.shadowRoot.append(span);  
    //  
  }  
} customElements.define("hello-world", HelloWorldElement);
```

Note que a classe deve estender `HTMLElement` e chamar o construtor da superclasse em seu construtor. É importante ressaltar que devemos chamar `this.attachShadow` para criar o Shadow DOM do elemento, com isso, podemos usar `this.shadowRoot` para obter o nó raiz do Shadow DOM.

Autonomous custom elements

Componentes definidos como o exemplificado anteriormente são chamados de Autonomous custom elements, pois eles definem elementos novos. Para utilizá-los, devemos usar a tag com o nome registrado no HTML.

```
<hello-world></hello-world>
```

Customized built-in elements

Outra opção que temos é customizar um elemento existente. Nesse caso, devemos estender o elemento desejado e indicar, na chamada, a `customElements.define`, qual elemento estamos customizando, conforme exemplo a seguir, onde customizamos o elemento `UL`:

```
class ListaColorida extends HTMLUListElement { constructor()  
{ super();  
  /* customização do elemento UL */  
}  
customElements.define("lista-colorida", ListaColorida, { extends: "ul" });
```

Além disso, a maneira de usar o elemento é diferente. Devemos usar o atributo `is` no elemento customizado, indicando o nome registrado via chamada a `customElements.define`:

```
<ul is="lista-colorida">  
  <li>Item 1</li>  
  <li>Item 2</li>  
  <li>Item 3</li>  
</ul>
```

Estilização no Shadow DOM

Quando criamos o Shadow DOM em um elemento, as regras de estilo do documento principal não afetam. Portanto, devemos criar elementos `style` dentro do Shadow DOM ou importar arquivos CSS por meio do elemento `link` para definir estilização. De forma análoga, estilos definidos dentro do Shadow DOM não afetam o DOM externo.

É importante ressaltar que, embora regras CSS externas não se apliquem a elementos dentro do Shadow DOM, eles podem herdar propriedades de seu elemento raiz.

Ciclo de vida do Web Component

Podemos implementar métodos na classe do Web Component para atuar em momentos específicos do ciclo de vida do componente, tais como os descritos a seguir:

- `connectedCallback`: chamado quando o elemento é adicionado ao DOM.

- `disconnectedCallback`: chamado quando o elemento é removido do DOM.
- `attributeChangedCallback`: chamado quando um atributo é adicionado, removido ou modificado. Apenas atributos indicados via propriedade estática `observedAttributes`, definida na classe do Web Component, são considerados.

Quando desejamos ler atributos passados para o elemento, devemos fazê-lo no `connectedCallback`, pois seus valores ainda não estarão definidos no construtor.



XPe

> Capítulo 3



Capítulo 3. Web Components (Parte 2)

3.1. HTML Template

Usando o recurso HTML Template, podemos definir fragmentos de HTML no documento que não são imediatamente renderizados, mas podem ser utilizados para criar elementos no DOM de forma facilitada, clonando o conteúdo do template. Um HTML template é definido via o elemento `template`, conforme exemplo a seguir:

```
<template id="template-wc">
  <style>:host { display: block;
background-color: #666; color: white;
padding: 1em; margin: 1em;
}
</style>
<h2>Este é o título</h2>
<p>Este é um conteúdo definido dentro do template</p>
</template>
```

Após definido o template, podemos utilizá-lo em um script da seguinte forma:

```
<script>
const tpl = document.getElementById("template-wc");
document.body.append(tpl.content.cloneNode(true));
</script>
```

É importante observar que usamos a função `cloneNode` na propriedade `content` do template para criar os elementos a serem adicionados na página. Essa função recebe um parâmetro booleano que indica se a clonagem é profunda ou rasa. Neste caso, devemos usar a clonagem profunda. Outra observação importante é que não devemos usar o template antes dele ser processado no documento.

Templates são uma forma conveniente para construir o Shadow DOM de um elemento, conforme exemplo a seguir:

```
class HelloWorldElement extends HTMLElement { constructor() { super(); this.attachShadow({ mode: "open" }); const tpl = document.getElementById("template-wc"); this.shadowRoot.append(tpl.content.cloneNode(true)); }}
```

Slots

O conteúdo inserido dentro de um elemento customizado não é exibido no documento, a não ser que utilizemos o recurso de slot:

```
<hello-world>
<div>Este é conteúdo do elemento.</div>
</hello-world>
```

Para exibir tal conteúdo, nosso elemento precisa incluir, dentro do Shadow DOM, um elemento slot, que marca a posição na qual o conteúdo será inserido, por exemplo.

```
<template id="template-wc">
<p>Aqui vai o conteúdo do elemento:</p>
<slot>valor padrão</slot>
</template>
```

Quando não há elementos a serem inseridos dentro do slot, seu conteúdo é exibido no lugar.

Slot nomeado

Quando um slot não tem nome, ele é considerado o slot padrão e abrigará todo o conteúdo do elemento que não tiver um slot especificado. No entanto, devem definir slots nomeados com o seguinte código:

```
<template id="template-wc">
<slot name="titulo">
<h2>Título padrão</h2>
</slot>
</template>
```

Feito isso, ao usar o elemento, devemos especificar o slot que ele será inserido.

```
<hello-world>
  <h2 slot="titulo">Este é o título</h2>
</hello-world>
```

Um slot pode receber mais de um elemento. No exemplo a seguir, tanto o h2 quanto o h3 serão exibidos dentro do mesmo slot:

```
<hello-world>
  <h2 slot="titulo">Este é o título</h2>
  <h3 slot="titulo">Outro título</h3>
</hello-world>
```

Web Components encapsulados em módulos

Para declarar um componente autocontido em um arquivo JavaScript, podemos utilizar a seguinte abordagem:

```
const tpl = document.createElement("template"); tpl.innerHTML = `
  <p>Aqui vai o conteúdo do elemento:</p>
  <slot></slot>
`; class HelloWorldElement extends HTMLElement {
  constructor() { super();
    this.attachShadow({ mode: "open" }); this.shadowRoot.append(tpl.content.cloneNode(true));
  }
} customElements.define("hello-world", HelloWorldElement);
```

Nesse exemplo, definimos o template dinamicamente, evitando depender de sua existência no documento HTML. Além disso, podemos encapsular o script como um módulo, para não poluir o escopo global.



XPe

> Capítulo 4



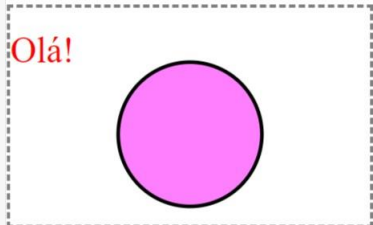
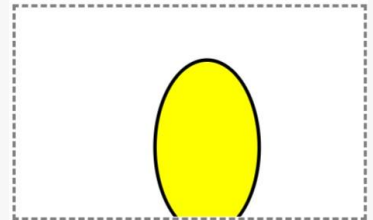
Capítulo 4. SVG

Scalable Vector Graphics (SVG) é uma linguagem de marcação para descrever gráficos vetoriais que é suportada nativamente na plataforma web. Com essa tecnologia, podemos criar imagens vetoriais que podem ser escaladas para qualquer resolução em um documento HTML e, também, manipuladas via JavaScript, por meio do DOM.

Figuras básicas

A seguir, temos uma série de exemplos de elementos SVG para desenhar retângulos, círculos, textos e elipse.

<pre><svg width="200" height="120"> <rect x="10" y="10" width="100" height="40" /> </svg></pre>	
<pre><svg width="200" height="120"> <circle cx="100" cy="60" r="30" fill="rgba(255, 0, 255, 0.5)" stroke="black" stroke-width="5" /> <circle cx="120" cy="70" r="30" fill="rgba(255, 0, 255, 0.5)" stroke="black" stroke-width="5" /> </svg></pre>	

<pre> <svg width="200" height="120"> <circle cx="100" cy="70" r="40" fill="rgba(255, 0, 255, 0.5)" stroke="black" stroke-width="2" /> <text x="0" y="30" style="fill: red; fontsize: 20px">Olá!</text> </svg> </pre>	
<pre> <svg width="200" height="120"> <ellipse cx="110" cy="80" rx="30" ry="50" style="fill: yellow; stroke: black; stroke-width: 2" /> </svg> </pre>	

O elemento path

Além de elementos para desenhar figuras básicas, como retângulos e círculos, o SVG possui o elemento *path*, que permite traçar linhas e arcos formando qualquer tipo de figura desejada. Podemos apenas traçar a figura, usando o atributo *stroke*, ou preencher seu interior, usando o atributo *fill*.

O atributo *d* recebe a especificação do *path*, que é uma *string* composta por uma série de comandos para traçá-lo. Os comandos suportados são descritos na tabela a seguir:

Comando	Descrição
M x y	Move para x, y
L x y	Desenha linha até x, y
H x	Desenha linha horizontal até x
V y	Desenha linha vertical até y
Z	Fecha o path (desenha linha até ponto inicial)

A ...	Arcos
C ou Q ...	Curva de Bézier

Para cada comando da tabela, existe uma variante com letra minúscula que aceita coordenadas relativas à posição atual do cursor, em vez de uma coordenada absoluta. Como exemplo, considere o *path* a seguir:

```
<svg width="200" height="120">
  <path d="M 50 20 h 50 q 80 10, 0 50" fill="none" stroke="red" />
</svg>
```

Nele, especificamos os seguintes comandos:

1. Move para a coordenada absoluta (50, 20).
2. Traça linha horizontal, movendo 50 unidades no eixo X.
3. Traça uma curva de Bézier quadrática até a coordenada relativa (0, 50).
4. Usa como ponto de controle a coordenada relativa (80, 10).

Como resultado, teremos a seguinte figura:



Transformações

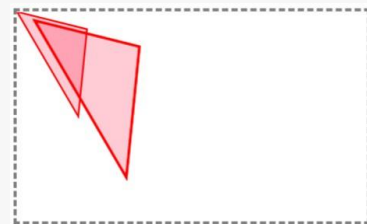
Além de desenhar figuras, podemos usar o elemento *g* para agrupar outros elementos e aplicar transformações ao sistema de coordenadas. As três transformações mais comuns são:

Comando	Descrição
translate (x, y)	Move o sistema de coordenadas para x, y.
rotate (a)	Rotaciona o sistema de coordenadas em <i>a</i> graus.

scale (sx, sy)	Aplica escala nos eixos x e y de sx e sy.
----------------	-------------------------------------------

Abaixo, temos um exemplo de transformações de translação e escala sendo aplicadas em conjunto em um elemento *path*:

```
<svg width="200" height="120" fill="rgba(255, 0, 50, 0.2)" stroke="red">  
<path d="M 0 0 l 40 10 l -5 50 Z" />  
  
<g transform="translate(10, 5) scale(1.5)">  
<path d="M 0 0 l 40 10 l -5 50 Z" />  
</g>  
</svg>
```





XPe

> Capítulo 5



Capítulo 5. Canvas

Canvas é uma API web moderna para desenhar gráficos 2D, ou mesmo 3D, em páginas web. Essa API tem duas diferenças fundamentais com relação a SVG:

1. Canvas é baseado em pixels, portanto, não pode ser escalado sem perda de qualidade para diferentes resoluções.
2. Desenhamos apenas via API JavaScript, e apenas o resultado é exibido como uma imagem. Não existem elementos no DOM para representar os gráficos desenhados no Canvas.

Desenhando no Canvas

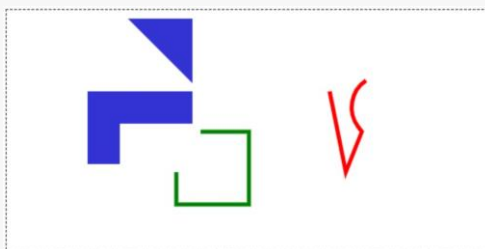
Para usar a API, primeiro devemos inserir um elemento `<canvas>` no documento. Com uma referência ao elemento, chamamos a função `getContext` para obter o objeto `context`, que fornece todas as funções para desenhar gráficos, conforme exemplificado a seguir:

```
<canvas width="600" height="300" id="canvas1"></canvas>
<script> const canvas = document.getElementById("canvas1");
const ctx = canvas.getContext("2d");
ctx.fillStyle = "rgba(0, 0, 200, 0.8)"; ctx.fillRect(100,
100, 130, 90);
ctx.strokeStyle = "green"; ctx.lineWidth = 5;
ctx.strokeRect(210, 150, 90, 90);

ctx.clearRect(140, 140, 100, 60);
ctx.strokeStyle = "red"; ctx.beginPath();
ctx.moveTo(400, 100); ctx.lineTo(420, 200);
ctx.lineTo(440, 150); ctx.arcTo(410, 110,
500, 50, 40);
```

```
//ctx.closePath(); ctx.stroke();  
//ctx.fill();  
let p = new Path2D("M 150 10 h 80 v 80 Z"); ctx.fill(p);  
</script>
```

Neste exemplo, primeiro obtemos o objeto `ctx` via chamada `canvas.getContext("2d")`. Com esse objeto, chamamos a função `fillRect` para preencher retângulos, `strokeRect` para traçar retângulos e `clearRect` para apagar uma região retangular. Além disso, usamos as funções `beginPath`, `moveTo`, `lineTo` e `arcTo` para criar um *path*. Podemos usar as funções `fill` ou `stroke` para preenchê-lo ou traçá-lo. Por fim, com o objeto `Path2D`, podemos criar *paths* usando a mesma notação do SVG. O resultado é exibido na figura a seguir:



Transformações

Assim como em SVG, podemos aplicar transformações ao sistema de coordenadas na API Canvas. No entanto, nesse caso, as transformações são aplicadas ao objeto `context`. Para conseguir desfazer as transformações, é conveniente usar a função `save` antes de uma transformação e `restore` após as operações para voltar o `context` ao estado antes do último `save`.

O exemplo abaixo demonstra as três transformações mais comuns: `translate`, `rotate` e `scale`:

```
const ctx = canvas.getContext("2d");
ctx.fillStyle = "rgba(0, 0, 200, 0.8)"; ctx.save();
ctx.translate(200, 100); ctx.fillRect(0, 0, 100,
100); ctx.restore();
ctx.save(); ctx.translate(200,
100); ctx.rotate(Math.PI / 8);
ctx.scale(4, 2); ctx.fillRect(0, 0,
50, 50); ctx.restore();
```

Uma diferença com relação ao SVG é que as transformações de rotação recebem o ângulo em radianos, não em graus.

Desenhando imagens

Além de desenhar primitiva gráficas, a API Canvas nos permite desenhar imagens carregadas a partir de arquivos. Para isso, temos duas opções: desenhar a partir de elementos no documento, ou instanciar uma imagem via o construtor `new Image()`. Em ambos os casos, podemos desenhar a imagem com a chamada:

```
ctx.drawImage(img, 0, 0, img.width / 2, img.height / 2);
```

A função `drawImage` recebe, como parâmetros, a imagem, as coordenadas x e y, a largura e a altura da imagem.



XPe

> Capítulo 6



Capítulo 6. Armazenamento local

A plataforma web nos oferece APIs para armazenamento de dados localmente, ou seja, no navegador do usuário, por meio de dois objetos disponíveis no escopo global: `localStorage` e `sessionStorage`. Ambos os objetos oferecem acesso armazenamento no formato chave/valor por meio das seguintes operações:

<code>setItem(chave, valor)</code>	Salva um valor com a chave informada. O valor deve ser uma <i>string</i> .
<code>getItem(chave)</code>	Recupera o valor com a chave informada, ou null, caso não exista.
<code>removeItem(chave)</code>	Exclui a entrada com a chave informada.
<code>clear()</code>	Exclui todas as entradas.

Embora seja possível armazenar apenas valores do tipo *string*, podemos codificar *arrays* ou objetos como JSON para contornar essa limitação:

```
localStorage.setItem("chave-1", JSON.stringify(valor));
```

De forma análoga, podemos decodificar os valores com `JSON.parse`:

```
const valor = JSON.parse(localStorage.getItem("chave-1"));
```

`localStorage`

Dados armazenados via objeto `localStorage` persistem por tempo indefinido, até que sejam excluídos explicitamente. O armazenamento está atrelado à origem da página que o salvou, ou seja, ao protocolo/host/porta da aplicação. Por questões de segurança, uma página é incapaz de ler o `localStorage` de outra origem.

Além de ler e escrever dados, podemos monitorar alterações no localStorage eternas à página carregada, escutando o evento storage no objeto window, por exemplo.

```
window.addEventListener("storage", () => { console.log("storage foi alterado externamente"); });
```

sessionStorage

Dados armazenados via objeto sessionStorage persistem pelo tempo da sessão de navegação, que corresponde ao tempo de vida de uma “aba” do navegador. Ou seja, os dados persistem ao recarregar a página ou mudar de endereço, mas desaparecem ao fechar a aba.



XPe

> Capítulo 7



Capítulo 7. Geolocalização

O navegador nos oferece suporte à API `navigator.geolocation`, que nos permite obter a posição geográfica do dispositivo, usando seu GPS ou outro mecanismo de localização disponível. No entanto, o acesso a essa API requer autorização explícita do usuário, por questões de privacidade e segurança.

Existem duas formas de acessar a localização. A primeira delas é por meio da chamada `navigator.geolocation.getCurrentPosition`, que recebe, como parâmetro, um call-back de sucesso e, opcionalmente, um call-back de erro. O callback de sucesso recebe, como parâmetro, o objeto `position`, que possui, entre outras informações, a propriedade `coords`, com a latitude e a longitude.

```
navigator.geolocation.getCurrentPosition(  
  (position) => { const { latitude, longitude } = position.coords;  
  },  
  (error) => { console.error(error);  
  }  
);
```

Uma segunda forma de acessar a API é via função `watchPosition`. Neste caso, também passamos call-backs, mas eles são chamados toda vez que a posição do dispositivo muda ao invés de uma única vez. Com isso, podemos monitorar a posição. Caso desejemos interromper o monitoramento, devemos chamar `clearWatch`, passando o id retornado pela chamada de `watchPosition`.

```
const id = navigator.geolocation.watchPosition(  
  (position) => { const { latitude, longitude } = position.coords;  
  },  
  (error) => {  
    console.error(error);  
  }  
); navigator.geolocation.clearWatch(id);
```



XPe

> Capítulo 8



Capítulo 8. Captura de áudio e vídeo

O navegador nos oferece suporte à API `navigator.mediaDevices`, que nos permite utilizar o microfone e a câmera do dispositivo. No entanto, o acesso a essa API requer autorização explícita do usuário, por questões de privacidade e segurança.

Para capturarmos áudio e/ou vídeo, devemos chamar a função `getUserMedia`, passando, como parâmetro, um objeto de opções no qual especificamos se queremos capturar as faixas de áudio, vídeo ou ambos. A chamada à função retorna uma *promise*, que resolve em um objeto *stream*. Podemos usar tal objeto como fonte de um elemento `<audio>` ou `<video>` no documento, para reproduzir a mídia. No exemplo abaixo, capturamos o *stream* de vídeo e o exibimos em um elemento `<video>` existente na página. Vale ressaltar que precisamos chamar `play` no elemento vídeo, após o carregamento dos metadados.

```
const video = document.getElementById("video");
navigator.mediaDevices.getUserMedia({ audio: false, video:
true,
})
.then((stream) => { video.srcObject = stream;
video.onloadedmetadata = () => { video.play();
};
});
```

Quando desejamos interromper a captura, devemos chamar a função `stop` em cada faixa de áudio/vídeo, conforme exemplo a seguir:

```
stream.getTracks().forEach((track) => track.stop());
```



XPe

> Capítulo 9



Capítulo 9. Notificações

O navegador nos oferece suporte à API Notification, que nos permite emitir notificações no sistema operacional, da mesma forma que aplicativos nativos da plataforma. Assim como outras APIs, o acesso a ela requer autorização explícita do usuário. No entanto, diferentemente do acesso à geolocalização ou a captura de áudio e vídeo, o pedido de permissão não é feito automaticamente ao usar a API.

Devemos fazer o pedido explicitamente via chamada à função `Notification.requestPermission`, que retorna uma *promise*.

Após solicitar permissão, podemos usar a propriedade `Notification.permission` para determinar se a permissão foi concedida (`granted`) ou negada (`denied`). Para enviar notificações, basta instanciar um objeto via construtor `Notification`, conforme exemplo a seguir:

```
const botaoDisparar = document.getElementById("disparar");

if (Notification.permission !== "granted") {
  Notification.requestPermission(); } botaoDisparar.onclick = ()
=> { if (Notification.permission === "granted") { const
notificacao = new Notification("Olá", { body: "Este é o
conteúdo da notificação.", icon: "icon.png",
});
notificacao.onclick = () => { window.focus();
}; } else { alert("Notificação não permitida.");
}
};
```

- Além de um título, a notificação pode receber um objeto com uma série de opções, entre elas o corpo (`body`), um ícone (`icon`), uma imagem (`image`) etc. Caso desejemos reagir quando o usuário clicar na notificação, podemos registrar um `handle` de `click` via propriedade `onclick` de `Notification`. No exemplo dado, estamos dando foco na janela da aplicação que enviou a notificação.



XPe

> Capítulo 10



Capítulo 10. Svelte

O Svelte (<https://svelte.dev/>) é mais um framework para desenvolvimento

Front End de aplicações web, oferecendo funcionalidades semelhantes ao Angular, React, Vue e outros. No entanto, esse framework usa uma abordagem essencialmente diferente dos demais, pois ele oferece atualização automática do DOM por meio de um processo de compilação e análise do código fonte, gerando um código final instrumentado. Na prática, a aplicação final não depende da biblioteca Svelte, o que gera aplicações de tamanho reduzido e muito eficientes.

Criando um projeto

Para criar um projeto, utilize o comando:

```
npx degit sveltejs/template nome-do-projeto
```

Em seguida, execute `npm install` dentro do diretório criado para instalar as dependências, visto que se trata de um projeto Node.js. Para iniciar o servidor de desenvolvimento, execute `npm run dev`. Para gerar os artefatos otimizados para implantação, utilize o comando `npm run build`.

Criando um componente

Uma aplicação é composta por um ou mais componentes. Cada componente é definido por meio de um arquivo `.svelte`, que pode conter HTML, CSS e JavaScript.

Temos, a seguir, um exemplo do componente `Contador.svelte`:

```
<script>
  let contador = 0;
</script>

<div>
  Valor: {contador}
  <button on:click={() => contador++}>Incrementar</button>
</div>
```

```
<button on:click={() => contador--}>Decrementar</button>
</div>
<style> button {
color: blue;
}
</style>
```

Note que o componente usa a sintaxe de chaves dentro do HTML para interpolar valores definidos em variáveis JavaScript. Esses valores são atualizados automaticamente quando alterados.

Reagindo a mudanças no estado

Através de um processo de análise e instrumentação do código, o Svelte é capaz de atualizar a interface quando o estado é modificado. Na prática, uma expressão dentro do template é reavaliada quando alguma das variáveis que ela utiliza é atribuída. No entanto, esse critério possui as seguintes limitações:

- Se modificarmos um array por meio de funções que o alteram, como push ou splice, a mudança não será detectada.
- Se houver mais de uma variável apontando para o mesmo objeto, as mudanças só serão detectadas se referenciarmos o objeto no template usando a mesma variável que foi usada na atribuição.

Comando condicionais e de repetição

Para exibir elementos condicionalmente, podemos usar o comando #if, passando uma expressão JavaScript que será reavaliada sempre que uma variável utilizada mudar.

```
{#if itens.length === 0}
<div>A lista está vazia</div>
{:else}
<div>...</div>
{/if}
```

Podemos adicionar vários comandos intermediários `{:else if ...}`, se for necessário tratar vários casos.

Para exibir componentes repetidamente, usamos o comando `#each`, passando um array como parâmetro. Em seguida, usamos a palavra-chave `as`, seguida do nome da variável que receberá o elemento do array. Opcionalmente, podemos capturar uma segunda variável com o índice do elemento.

```
{#each itens as item, i}  
  <div>{item.descricao}</div>  
{/each}
```

Manipulação de formulários

Uma forma de capturar o valor de um input no estado do componente é utilizar expressões no atributo `value`, e tratar os eventos disparados quando ele for alterado, por exemplo.

```
<script> let minhaVar = "abc";  
</script>  
<input value={minhaVar} on:input={e => minhaVar = e.target.value} />  
<div>Valor atual: {minhaVar}</div>
```

No entanto, existe uma maneira muito mais conveniente de fazer isso, usando a diretiva `bind`:

```
<script> let minhaVar = "abc";  
</script>  
<input bind:value={minhaVar} />  
<div>Valor atual: {minhaVar}</div>
```

Podemos usar `bind:value` para elementos input de texto, select, textarea etc.

Também podemos usar `bind:checked` para input do tipo radio ou checkbox.

Comunicação entre componentes

Componentes podem trocar informações por dois mecanismos básicos. O primeiro deles é a passagem de atributos de um componente pai para o filho que, no Svelte, ganham o nome de props. O segundo é o disparo de eventos do filho para o pai. No exemplo abaixo, temos um componente `ListaItem.svelte` que recebe dois props: `descricao` e `comprado`. Note que props são definidos como variáveis com a palavra-chave `export`. Além disso, o componente dispara o evento `itemremovido`, usando a função `dispatch`, criada por meio da chamada a `createEventDispatcher`.

```
<script> import { createEventDispatcher } from "svelte"; const
dispatch = createEventDispatcher(); export let descricao; export
let comprado;
</script> <div>
  <label>
    <input type="checkbox" bind:checked={comprado}>
    {descricao}
  </label>
  <button on:click={() => dispatch("itemremovido")}>Excluir</button>
</div>
```

O componente acima pode ser utilizado da seguinte forma:

```
<ListaItem bind:comprado={item.comprado}
descricao={item.descricao}
on:itemremovido={() => removerItem(item)} />
```

Note que podemos usar a diretiva `bind:prop` em qualquer prop do componente, fazendo uma vinculação de leitura e escrita. Além disso, note que a sintaxe para registrar *handler* de eventos usa a diretiva `on:nomeevento`.

Declarações reativas

Vimos que as expressões dentro de templates são reavaliadas quando suas variáveis mudam, mas o mesmo não acontece em atribuições

dentro da tag script. Por exemplo, a variável `itensPendentes` declarada a seguir não será atualizada quando modificarmos a variável `itens`.

```
let itens = [];  
let itensPendentes = itens.filter(item => !item.comprado).length;
```

Se desejarmos que isso ocorra, devemos marcar a declaração como reativa, usando a seguinte sintaxe:

```
let itens = [];  
$: itensPendentes = itens.filter(item => !item.comprado).length;
```

De forma análoga, podemos marcar um bloco de código como reativo usando a marcação `$:`.

```
let itens = [];  
$: {  
  // Este bloco executa novamente sempre que itens mudar  localStorage.setItem("lista-compras", JSON.stringify(itens));  
}
```



XPe

> Capítulo 11



Capítulo 11. Introdução a Micro Frontends

Micro Frontends é um estilo arquitetural no qual aplicações Front End desenvolvidas e implantadas de forma independente são compostas como uma única grande aplicação. Essa arquitetura é inspirada na arquitetura de microsserviços, que é utilizada no Back End.

Motivação

Em uma arquitetura tradicional, onde o Front End é uma única aplicação, podemos ter os seguintes problemas quando a aplicação cresce em tamanho e complexidade.

- Torna-se difícil mudar a arquitetura.
- Torna-se difícil trocar tecnologias ou experimentar novas ideias.
- Não é possível substituir uma parte sem parar a aplicação como um todo.
- Torna-se difícil paralelizar o trabalho em equipes distintas.
- Um problema em uma parte de aplicação pode parar toda a aplicação.

Para solucionar esses problemas, a arquitetura Micro Frontends consiste em subdividir o Front End em aplicações menores, integradas de forma fracamente acoplada por uma aplicação principal. Cada um dos Micro Frontends deve:

- Ser desenvolvido, testado, implantado e operado de forma independente;
- Comunicar-se com outros serviços por meio de APIs bem definidas;

- Ser especializado em um problema específico.

Desvantagens

A arquitetura de Micro Frontends, embora resolva os problemas citados anteriormente, traz outras dificuldades, tais como:

- Arquitetura mais complexa.
- Dificuldades na estilização, pois CSS é inerentemente global.
- Dificuldade em reutilizar código, visto que as aplicações são independentes.
- Dificuldade na comunicação entre aplicações, que agora não podem passar dados e chamar funções entre si livremente.

Portanto, devemos adotar tal arquitetura apenas quando os problemas da arquitetura monolítica forem, de fato, mais relevantes para o projeto do que as dificuldades impostas por tal arquitetura, e quando for possível subdividir o sistema em subsistemas com baixo acoplamento. Lembre-se que, de forma geral, é melhor um sistema monolítico do que vários sistemas independentes, mas altamente acoplados.

Estratégias de integração de Micro Frontends

Um dos desafios da arquitetura Micro Frontends é como integrar os diferentes subsistemas de forma que, do ponto de vista do usuário, parece uma única aplicação. Para isso, normalmente usamos uma aplicação contêiner que encapsula as demais e dispara a inicialização de cada uma. As diferentes aplicações podem ser ativadas de acordo com a URL, por exemplo.

Integração via iframe

O elemento `<iframe>` pode ser utilizado para tal tarefa, pois ele é capaz de exibir um documento HTML dentro de outro, de forma totalmente

isolada e encapsulada. No exemplo abaixo, alteramos a propriedade src do iframe para exibir o Micro frontend desejado:

```
<html>
<body>
  <h1>Welcome to Feed me!</h1>
  <iframe id="app-container"></iframe>
  <script>    const microFrontendsByRoute = {
    "/": "https://browse.example.com/index.html",
    "/order-food": "https://order.example.com/index.html",
    "/user-profile": "https://profile.example.com/index.html",
  };    const iframe = document.getElementById("app-container");    iframe.src =
microFrontendsByRoute[window.location.pathname];
  </script>
</body>
</html>
```

Esta abordagem tem como vantagem a simplicidade, pois o iframe isola completamente as aplicações. No entanto, ela torna mais difícil implementar roteamento/histórico dentro de cada aplicação, responsividade e comunicação entre aplicações.

Integração via JavaScript

Outra abordagem é carregar cada sistema como um JavaScript independente, e chamar uma função estabelecida que renderiza cada um deles na página.

```
<html>
<body>
  <h1>Welcome to Feed me!</h1>
  <!--
  - Cada módulo disponibiliza uma função no escopo global que o renderiza -->
  <script src="https://browse.example.com/bundle.js"></script>
  <script src="https://order.example.com/bundle.js"></script>
  <script src="https://profile.example.com/bundle.js"></script>
  <div id="app-root"></div>  <script
type="text/javascript">    const
microFrontendsByRoute = {    "/":
renderBrowseRestaurants,
    "/order-food": renderOrderFood,
    "/user-profile": renderUserProfile,
  };
  </script>
```

```
const renderFunction =
microFrontendsByRoute[window.location.pathname];
renderFunction("app-root");
</script>
</body>
</html>
```

Esta abordagem permite grande flexibilidade, pois é possível passar parâmetros de forma mais fácil, bem como compartilhar objetos globais se for necessário. No entanto, as aplicações podem interferir uma na outra de forma indesejável se os devidos cuidados não forem tomados, visto que elas estão no mesmo documento. Por exemplo, podemos ter regras CSS que interferem em elementos indevidos, handlers de eventos que capturam incorretamente eventos disparados por outra aplicação, entre outras situações.

Integração via WebComponents

Como variante da abordagem via JavaScript, podemos definir cada aplicação como um Web Component definido em um arquivo JavaScript.

```
<html>
<body>
  <h1>Welcome to Feed me!</h1>
  <!-- Cada módulo registra um custom element -->
  <script src="https://browse.example.com/bundle.js"></script>
  <script src="https://order.example.com/bundle.js"></script>
  <script src="https://profile.example.com/bundle.js"></script>
  <div id="app-root"></div>

  <script type="text/javascript">    const
webComponentsByRoute = {
  "/": "micro-frontend-browse-restaurants",
  "/order-food": "micro-frontend-order-food",
  "/user-profile": "micro-frontend-user-profile",
};
  const webComponentType = webComponentsByRoute[window.location.pathname];    const root =
document.getElementById("app-root");
  root.appendChild(document.createElement(webComponentType));
</script>
</body>
</html>
```

- Nesta abordagem ainda temos bastante flexibilidade, mas ganhamos maior isolamento, pois os estilos e eventos ficarão contidos dentro do Shadow DOM de cada aplicação. No entanto, o suporte a navegadores mais antigos pode ser um problema, pois a especificação Web Components ainda é relativamente recente

Referências

GEERS, Michael. Micro Frontends: Extending the Microservice ide ato Frontend Development, 2021. Disponível em: <<https://micro-frontends.org/>>. Acesso em: 07 abr. 2022.

JACKSON, Cam. Micro Frontends, 2019. Disponível em: <<https://martinfowler.com/articles/micro-frontends.html>>. Acesso em: 07 abr. 2022.

MOZILLA. MDN Web Docs, 2021. Disponível em: <<https://developer.mozilla.org/>>.

Acesso em: 07 abr. 2022.

SVELTE. Documentação oficial do Svelte, 2021. Disponível em: <<https://svelte.dev/>>. Acesso em: 07 abr. 2022.

W3C. CSS Snapshot, 2020. Disponível em: <<https://www.w3.org/TR/CSS/>>. Acesso em: 07 abr. 2022.

W3C. HTML Standard, 2021. Disponível em: <<https://www.w3.org/html/>>. Acesso em: 07 abr. 2022.