

js.web.components

João Pedro Martins Neves

Introdução

Conceito

- É um *framework* escrito em *vanilla* (puro) **JavaScript** com programação orientada a objectos, que pretende criar uma estrutura standard e simplificar diversas operações dinâmicas com o **DOM** numa aplicação web.

Módulos Principais

- Startup
- TemplateCompiler
- Component

Startup

- Onde a aplicação inicia. O ponto de entrada.
- Onde os componentes são registrados para futura compilação.
- Contém também alguma lógica de funcionamento dos componentes, nomeadamente atualizações dinâmicas do **DOM**.

Component

- A aplicação é dividida em componentes/partes lógica individuais.
- Contém informação sobre o próprio componente.
- Contém algum do seu o seu funcionamento.
- Através de uma **Proxy**, está atento a eventos de atualização de propriedades e, se necessário, lança eventos internos.
- É, também, uma classe com o padrão **Observer**.

TemplateCompiler

- A classe que está incumbida de compilar as *templates* e gerar o HTML necessário para o funcionamento aplicação.
- Caso necessário, guarda dados no objeto global Startup.

Utilização

Criação de um Componente

Exemplo:

```
class TestComponent extends Component {  
  Constructor() {  
  
    Super(  
      'app-test', // Nome do Component.  
      '<p> Hello World! </p>', // A template.  
      [''] // CSS.  
    );  
  
    this.title = 'My Website'; // Uma propriedade.  
  }  
}
```

Injeção de um componente na View

Exemplo:

```
<body>
```

```
<app-test&>  
</app-test&>
```

```
</body>
```

(Fazer referencia aos componentes na página HTML desejada)

Inicialização (Startup)

Exemplo:

```
new Startup()
```

```
.addComponent( new HeaderComponent() )
```

```
.addComponent( new NameListComponent() )
```

```
.build();
```

- Deve ser o último *script* a ser executado.
- A instância é guardada como objeto global em **Window**, disponível, por tanto, acedendo a “startup”.
- Guarda os componentes criados e a ser injetados no **DOM**.
- Os componentes são guardados num dicionario, em que a Key é o seu nome.

Funcionamento por Funcionalidade

Renderização de uma propriedade de um Componente

Exemplo:

```
const testTemplate = `  <h1> <_> title </_> </h1>  
</div>  
`;
```

```
class TestComponent extends Component {  
  constructor() {  
    super( 'app-test', testTemplate, ["] ");  
    this.title = 'My Website';  
  }  
}
```

A sintaxe para ir buscar um valor ao componente é:

“<_> (nome-da-propriedade) </_>”

- A classe TemplateCompiler, ao encontrar estes *tokens*, vai buscar o valor da propriedade ao componente e injeta o mesmo diretamente no HTML.

Renderização de de uma propriedade de um Componente, com *data binding* (atualização dinâmica da View)

Exemplo:

```
const testTemplate = `  <p> <_> state.myName </_> </p>  
</div>  
`;
```

```
class TestComponent extends Component {
```

```
  constructor() {  
    super( 'app-test', testTemplate, ["] );
```

```
    this.state = this.createState( {  
      myName: 'João Neves'  
    } );  
  }
```

```
}
```

Exemplo do HTML compilado:

```
<div class="container">  
  <p>  
    <span  
      data-component="app-test"  
      data-binding="myName"  
    >  
      João Neves  
    </span>  
  </p>  
</div>
```

- A sintaxe é:

`<_> state.(nome-da-propriedade) </_>`

- Quando a classe TemplateCompiler encontra o nome “state” dentro de token de propriedade, injeta o respetivo valor dentro de um tag **span** com o nome do componente e propriedade em propriedades do mesmo. Estes são podem ser encontrados no **DOM**, na propriedade **dataset** do Node.
- Assim que haja alguma alteração ao valor da propriedade no componente, este, através de uma Proxy é apanhado.
- De seguida, é encontrado o elemento correto no DOM através do nome do componente e respetiva propriedade.
- O **span** é atualizado com o novo valor.

Renderização de valores dentro de um *array*, através de uma *tag for loop*

Exemplo:

```
const nameListTemplate = `  <ul>  
    <_for let="person of names">  
      <li> <_> person </_> </li>  
    </_for>  
  </ul>  
</div>  
`;
```

```
class NameListComponent extends  
Component {  
  
  constructor() {  
    super( 'app-nameList', nameListTemplate,  
[""]);  
  
    this.names = ['John Doe', 'Oliver Hoe',  
'Fiona Silva'];  
  
  }  
  
}
```

- Ao encontrar o *token* “**_for**”, a classe **TemplateCompiler** retira o bloco de **HTML** a ser repetido de dentro do *tag*, e vai buscar os valores do *array* ao componente a que a *template* pertence.
- Por cada valor, esta repete o bloco de código, podendo ainda injetar valores presentes no componente na View.

Renderização de valores dentro de um *array*, através de uma *tag for loop*, com *data binding*

Exemplo:

```
const todoListTemplate = `

- <_for let="todoItem of state.todoItems">  
<li> <_> todoItem </_> </li>  
</_for>  
</ul>  
</div>`;


```

```
class TodoListComponent extends Component {
```

```
  constructor() {  
    super( 'app-todoList', todoListTemplate, ["] );
```

```
    this.state = {  
      todoItems: this.createState( ['Study', 'Learn  
design patterns', 'Learn data structures'],  
'todoItems' )  
    };
```

```
  }
```

```
}
```

Exemplo do HTML compilado:

HTML:

```
<div class="container">  
  <ul>  
    <span  
      data-component="app-test"  
      data-binding="myName">  
      <li> Study </li>  
      (...)  
    </span>  
  </ul>  
</div>
```

TEMPLATE:

```
<template  
  data-component="app-nameList"  
  data-binding="todoItems"  
  data-token="for">  
  <_for let="todoItem of state.todoItems">  
    <li> <_> todoItem </_> </li>  
  </_for>  
</template>
```

- Comporta-se da mesma forma que o anterior, porém, depois do **TemplateCompiler** compilar a primeira versão do **HTML**:
 - Cria um **span** que alberga o bloco compilado dentro do **FOR** de forma a marcar a posição do elemento no **DOM**.
 - Cria também uma **tag template** com o bloco não compilado dentro da mesma, para no final da compilação de todos os componentes ser injetado no **DOM**.
- De seguida, a classe **Startup** injeta todas as *templates* que foram guardadas pelo **TemplateCompiler** na mesma, no final do **DOM**.
- Por fim, adiciona a um dicionário presente no respetivo componente, composto pelo nome da propriedade como *Key* e como valor uma função *callback* contendo a lógica de atualização da **View**, a qual é chamada pelo componente quando existe uma alteração nessa propriedade, encontrando e por sua vez compilando o conteúdo da **template** anteriormente injetada no **DOM**.

Chamar um método do componente depois de eventos do DOM

Exemplo:

```
const testTemplate = `

```
 <button (click)="addItem()"
 class="btn btn-success" type="button">
 Add Item
 </button>
```



```
</div>
`;
```



```
class testComponent extends Component {
```



```
 constructor() {
 super('app-todoList', testTemplate, []);
 }
```



```
 addItem(e) {
 console.log('Someone called me! \n', e);
 }
```



```
}
```



- Ao encontrar o token de eventos – (nomeEvento) – a classe TemplateCompiler guarda no componente um objeto informações:
  - Nome do evento (e.g: “click”).
  - Nome do método a ser chamado.
  - Um identificador único.
- O compilador adiciona uma propriedade dataset ao elemento original com o identificador único e o nome do método a chamar.  
  
E.g: data-eventmethodcall="K0na51" data-eventmethodname="addItem"
- Depois da compilação do HTML inicial, a Startup adiciona um eventListener ao evento desejado, no elemento que é identificado no objeto criado anteriormente, com um callback que chama esse método.


```

Obrigado!