06

Entendendo a estrutura

Transcrição

Sem precisarmos fazer nada, o comando `npm run dev` levantou um servidor http totalmente já configurado para que possamos acessar nossa aplicação, e mais, automaticamente abriu o navegador padrão do sistema operacional que carregou o arquivo `alurapic/index.html`. Mordomia assim só na casa da minha mãe!

Falando em mordomia, veremos que esse servidor voltado para o ambiente de desenvolvimento faz muito mais do que imaginamos, mas primeiro vamos entender a estrutura do projeto que foi criada, começando pelo conteúdo do arquivo `alurapic/index.html`:

```
<!-- alurapic/index.html -->

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>alurapic</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="/dist/build.js"></script>
  </body>
</html>
```

Espere um pouco! Essa página apenas importa um script e possui uma div vazia. Como é possível que ela seja a página que estamos visualizando no navegador? E se eu disser para vocês que o conteúdo exibido no navegador, inclusive todos os arquivos da pasta `alurapic/src` foram transformados no script `/dist/build.js`? Quer dizer que nossa página se transformou em um script? Sim!

Babel, Webpack e geração do bundle

Com a revelação que acabei de declarar, acredito que vocês estejam ansiosos para ver o conteúdo de `/dist/build.js`, mas sinto informa-los que esse arquivo não existe fisicamente, ainda. O conteúdo da pasta `alurapic/src` foi transformando em memória no arquivo `build.js`, por isso o arquivo não existe. Isso se dá assim para acelerar o tempo de desenvolvimento permitindo que o desenvolvedor veja o quanto antes o resultado de suas alterações no projeto. Aprendemos a gerar esse arquivo ainda neste curso, para que possamos distribuir nossa aplicação.

Voltando ao nosso arquivo `build.js`, eu disse que ele é o resultado da transformação dos arquivos da pasta `alurapic/src`, mas quem realiza essa transformação? Qual sua finalidade? Primeiramente, ocorrem duas transformações cada uma com ferramentas diferentes.

A primeira transforma através do processo de transcompilação todo o código escrito usando ES2015 para ES5 garantindo maior compatibilidade da nossa aplicação, mais notadamente em browsers desatualizados de smartphones. A segunda se encarrega de gerar um bundle para que seja carregado pelo navegador.

Para realizar as transformações anteriores que acabei de citar, o Vue CLI utiliza respectivamente Babel (https://cursos.alura.com.br/course/javascript-es6-orientacao-a-objetos-parte-3/task/20294) e WebPack

(https://webpack.github.io/). Por fim, vale ressaltar que Webpack vai mais além do que simplesmente criar um bundle, mas para início de conversa o que sabemos é suficiente para podermos continuar.

Agora que você já entendeu o motivo e como o `build.js` é gerado, vamos deixá-lo de lado e focar a pasta `alurapic/src` para entender o papel de cada arquivo que fará parte do bundle. Aliás, qual deles equivale ao conteúdo que é exibido em nosso navegador?

Componentes declarados em Single File Templates

O arquivo que corresponde à página que estamos vendo no navegador é o `alurapic/src/App.vue` :

```
<!-- src/App.vue -->

<template>
  <div id="app">
    <img src="./assets/logo.png">
    <h1>{{ msg }}</h1>
    <h2>Essential Links</h2>
    <ul>
      <li><a href="https://vuejs.org" target="_blank">Core Docs</a></li>
      <li><a href="https://forum.vuejs.org" target="_blank">Forum</a></li>
      <li><a href="https://gitter.im/vuejs/vue" target="_blank">Gitter Chat</a></li>
      <li><a href="https://twitter.com/vuejs" target="_blank">Twitter</a></li>
    </ul>
    <h2>Ecosystem</h2>
    <ul>
      <li><a href="http://router.vuejs.org/" target="_blank">vue-router</a></li>
      <li><a href="http://vuex.vuejs.org/" target="_blank">vuex</a></li>
      <li><a href="http://vue-loader.vuejs.org/" target="_blank">vue-loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-vue" target="_blank">awesome-vue</a></li>
    </ul>
  </div>
</template>

<script>
export default {
  name: 'app',
  data () {
    return {
      msg: 'Welcome to Your Vue.js App'
    }
  }
}
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
```

```
h1, h2 {
  font-weight: normal;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  display: inline-block;
  margin: 0 10px;
}

a {
  color: #42b983;
}
</style>
```

No entanto, ele não é uma página, mas um **Single file template** (template de único arquivo) que equivale a um **módulo** que declara um **componente**. Muito coisa para um arquivo só, não? O que precisamos entender aqui é que o arquivo sendo um módulo, se quisermos usar o componente que ele declara precisamos importá-lo em outros módulos da aplicação que queira utilizá-lo.

Pense em um módulo como uma caixa preta que pode ter diversas funcionalidades e só aquelas que forem explicitamente exportadas podem ser utilizadas em outros módulos. Tanto isso é verdade que dentro da tag `<script>` de `App.vue` há a instrução `export default` permitindo que nosso componente seja importado por outros módulos. Aliás, falando em componente, que nada mais são do que um objeto que possui sua apresentação, dado e comportamento. É por isso que nosso componente é definido através de três grandes blocos: template (apresentação), script (comportamento e dados) e style (o estilo da apresentação)

Sabemos que esse arquivo será transformado automaticamente para algo que seja compreendido pelo navegador, mas como é feita a ligação desse componente com `index.html` , uma vez que o componente é exibido assim que nossa página é carregada. É isso que veremos.