

Curso Técnico Superior Profissional em Tecnologias e Programação de Sistemas de Informação

Unidade Curricular: Desenvolvimento Web – Front-End

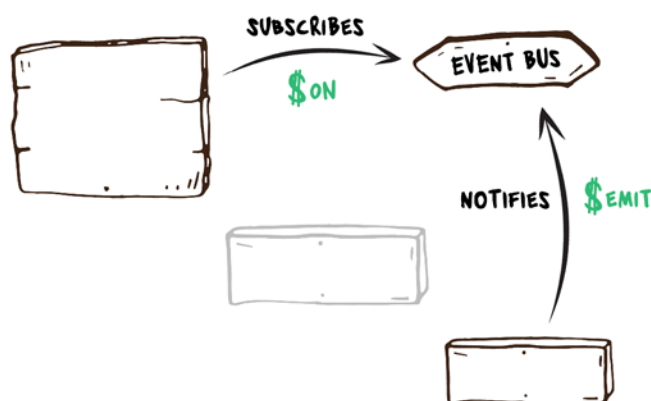
1º Ano/2º Semestre

Docente: Marco Miguel Olival Olim

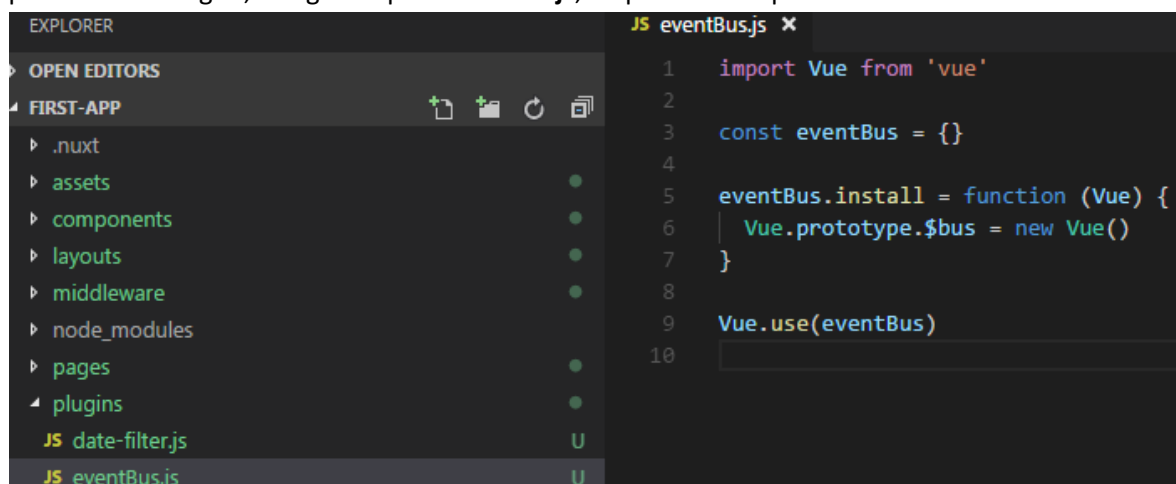
Data 01/06/2018

ESTE EXERCÍCIO ABORDA DIFERENTES MÉTODOS DE COMUNICAÇÃO ENTRE COMPONENTES

Após a conclusão da ficha 21 ficamos a conhecer o processo de comunicação entre dois componentes, ou seja, com **\$emit** para enviar dados e **props** para consumir dados pelo componente. No entanto, se diversos componentes necessitarem dos mesmos dados este processo aumenta de complexidade por termos de identificar as diversas fontes e destino dos dados, principalmente num processo de refactoring. Um dos padrões recomendados para solucionar este problema consiste em gravar estes dados num único repositório, o qual se torna a fonte de toda a informação e ao qual qualquer componente pode aceder para obter os dados que necessita. A este padrão de desenho de *software* designamos por **EVENT BUS**.



O VueJS já vem com este mecanismo preparado, mas no NUXT necessita que seja implementado. Começamos por criar um Plug-in, designado por **eventBus.js**, na pasta correspondente do NUXT



```
1 import Vue from 'vue'
2
3 const eventBus = {}
4
5 eventBus.install = function (Vue) {
6   Vue.prototype.$bus = new Vue()
7 }
8
9 Vue.use(eventBus)
10
```

Registamos o plugin no **nuxt.config.js** e assim o Event Bus já fica pronto a ser utilizado

```
EXPLORER JS eventBus.js JS nuxt.config.js x
OPEN EDITORS
FIRST-APP
  .nuxt
  assets
  components
  layouts
  middleware
  node_modules
  pages
  plugins
    JS date-filter.js
    JS eventBus.js
    README.md
  static
  store
  .editorconfig
  .gitignore
  JS nuxt.config.js
  package-lock.json
  package.json
  README.md
17 { rel: 'icon', type: 'image/x-icon', href: '/favicon.ico' }
18 ]
19 },
20
21 /*
22 ** Customize the progress-bar color
23 */
24 loading: { color: '#3B8070' },
25
26 /*
27 ** Global CSS
28 */
29 css: [
30 ],
31
32 /*
33 ** Plugins to load before mounting the App
34 */
35 plugins: [
36   '~plugins/date-filter.js',
37   '~plugins/eventBus.js'
38 ],
39
40 /*
```

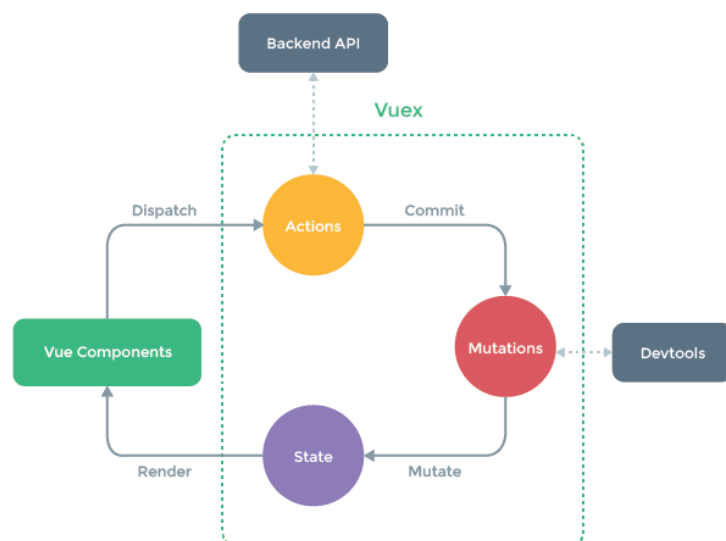
Quando necessitarmos de guardar dados usamos: **this.\$bus.\$emit('nome-do-evento', 'dados')**

```
37 methods:{
38   carregaCarrinho(artigo){
39     this.carrinhoCompras.unshift({...artigo, dataCompra: new Date()});
40     this.ultimaCompra = this.carrinhoCompras[0].dataCompra;
41     this.$bus.$emit('CARREGA_CARRINHO', artigo)
42   },
}
```

Para acedermos aos dados no Event Bus usamos: **this.\$bus.\$on('nome-do-evento', 'dados')**

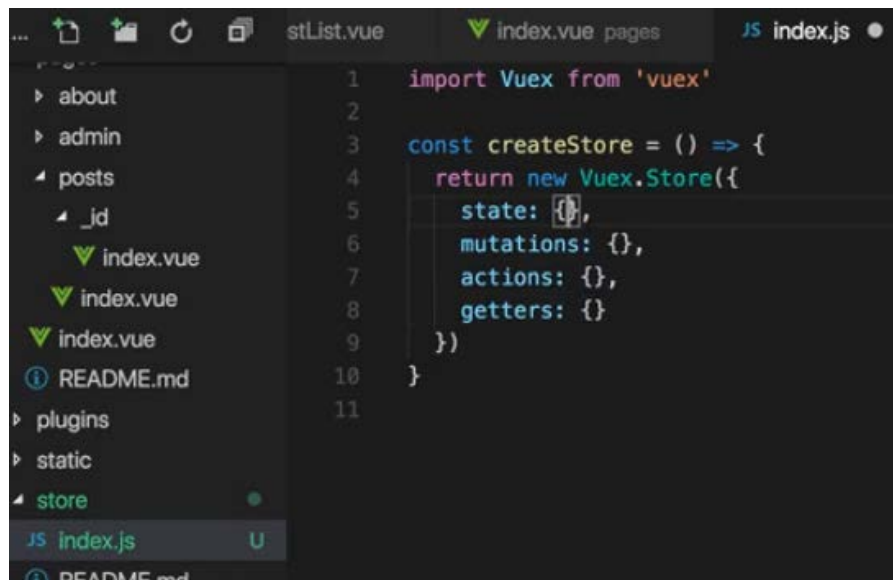
```
36 },
37 created () {
38   this.$bus.$on('CARREGA_CARRINHO', (data) => { this.artigo = data } )
39 },
40 methods:{
```

Refira-se que o EventBus é funcional entre componentes na mesma página, mas entre páginas (**pages**) o NUXT faz o reset aos Plug-ins e perdem-se os dados, devendo-se por isso usar o mecanismo integrado no NUXT: a **VUEX store**. Este padrão de desenho, semelhante ao REDUX, é baseado no Flux usado no Facebook



Cofinanciado por:

Como o VUEX já vem integrado no NUXT não precisamos de instalar, configurar ou importar nada. Basta criar o ficheiro **index.js** na pasta **store** para começar a parametrizar a store na nossa aplicação



```
1 import Vuex from 'vuex'
2
3 const createStore = () => {
4   return new Vuex.Store({
5     state: {},
6     mutations: {},
7     actions: {},
8     getters: {}
9   })
10 }
11
```

Basicamente os nossos dados são guardados em **state**. O acesso a **state** é um processo igual ao EventBus. Para alterarmos o state usamos as mutations, que é também onde colocamos o nosso Business Logic.

```
import Vuex from 'vuex'

const createStore = () => {
  return new Vuex.Store({
    state: {
      carrinhoCompras: []
    },
    mutations: {
      setCarrinho(state, artigos) {
        state.carrinhoCompras = artigos
      },
    },
    actions: {},
    getters: {},
  })
}

export default createStore
```

Quando necessitarmos de guardar dados usamos o **commit**:

```
40
41   methods: {
42     carregaCarrinho(artigo) {
43       this.carrinhoCompras.unshift({...artigo, dataCompra: new Date()});
44       this.ultimaCompra = this.carrinhoCompras[0].dataCompra;
45       //this.$bus.$emit('CARREGA_CARRINHO', artigo)
46       this.$store.commit('setCarrinho', this.carrinhoCompras);
47     },
48   },
```

Cofinanciado por:



Para acedermos aos dados acedemos diretamente ao **state**:

```
37   created () {
38     //this.$bus.$on('CARREGA_CARRINHO', (data) => { this.artigo = data } )
39     this.carrinhoCompras = this.$store.state.carrinhoCompras
40   },
41   methods:{
```

Por vezes necessitamos que os dados nos componentes se atualizem automaticamente quando houver alguma alteração da propriedade no **state**. Nesta circunstância temos de implementar o getters porque só estes é que são **reactivos**:

```
13   actions: {},
14   getters:{
15     carrinhoCompras(state) {
16       return state.carrinhoCompras
17     }
18   },
19 }
```

Para, tipicamente, acedermos a um getter:

```
41   computed: {
42     carrinhoCompras() {
43       return this.$store.getters.carrinhoCompras
44     }
45   },
46   methods:{
```

No caso das **actions**, acedemos da mesma forma que as mutations mas usamos **dispatch** em vez de commit. A diferença entre mutations e actions é que o primeiro funciona sincronamente e o segundo assincronamente (por isso é indicado para chamadas de http). No entanto, está previsto pela equipa de desenvolvimento do VUEX juntar actions e mutations numa única operação.