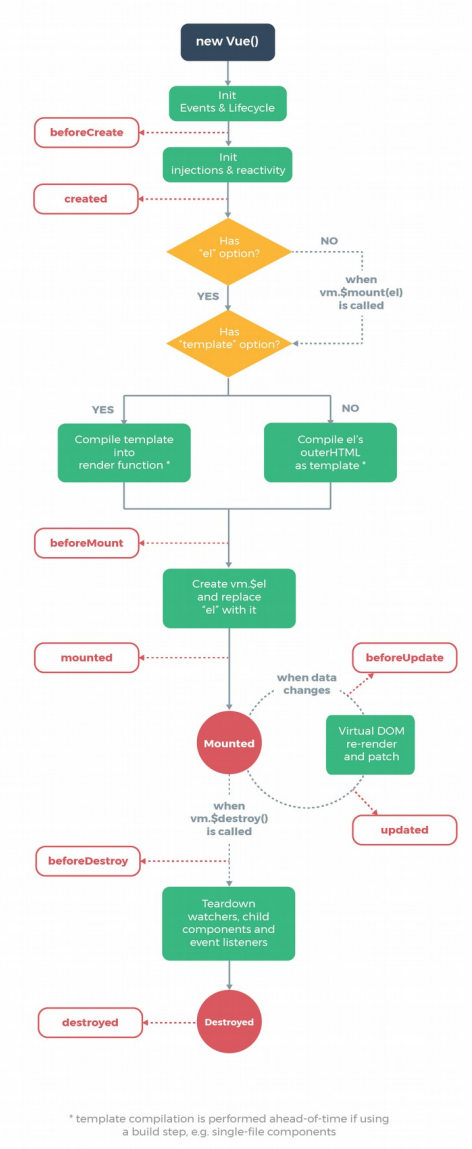
Vue

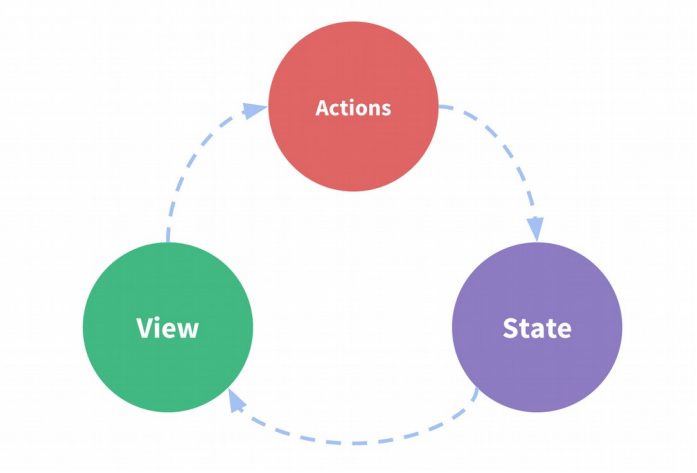
- A partir de la versión 3 de **Vue CLI** se pueden generar proyectos con **Typescript**.
  - *npm install –global @vue/cli*
  - *vue create project-name*
- *vue-class-component*
  - Permite añadir lógica de control en los estados por los que pasa la aplicación durante su ciclo de vida (beforeCreate, created, beforeMount…)
- *npm install –save vue-property-decorator (opcional)*
  - Permite utilizar decoradores para el uso de atributos del componente que estemos definiendo (@Emit, @Component, @Prop, @Watch...)



Vue Estructura de proyecto

- **./src/main.ts**: fichero de entrada de nuestra app. Lugar dónde importaremos las rutas, el store, otras librerías, hojas de estilo, configuraciones globales…
- **./src/router.ts**: mapea nuestros componentes a rutas del navegador permitiéndo su renderización.
- **./src/classes**: modelo de la aplicación.
- **./src/views**: son componentes que tienen asociados una ruta en el enrutador. Éstos incluyen otros componentes.
- **./src/components/shared**: componentes reutilizables por diferentes vistas.
- **./src/components/others**: componentes no reutilizables que no son vistas.
- **./src/App.vue**: componente principal de nuestra app. Normalmente contiene la etiqueta **<router-view></router-view>** la cual permite cambiar su contenido según a la ruta seleccionada por el usuario.
- **./public/index.html**: lugar dónde **Vue** montará el component principal.
- **./src/store** (opcional): constantes de Vuex para los tipos, módulos en subcarpetas y el store principal.
- **./src/{utils,services}** (opcional): funcionalidades comunes y reutilizables por varios componentes.
- **./src/assets**: hojas de estilo, imágenes, fuentes, librerías.
- **./src/shims-vue.d.ts**: fichero que ayuda a *Typescript* a manejar los tipos de las librerías que importa nuestra app si esta no tiene instalados los typings (@types/library-name).

Vuex

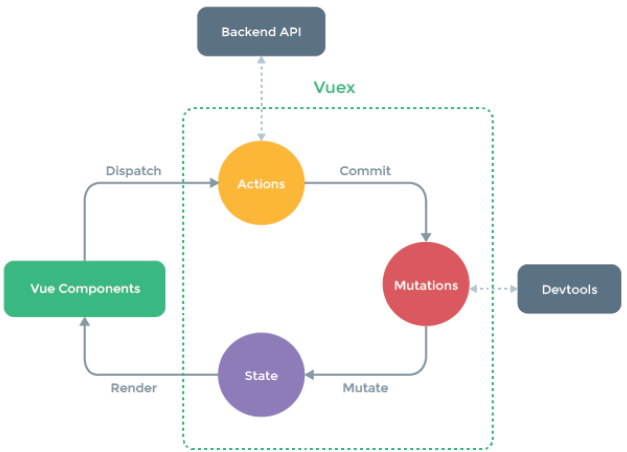


- Cualquier aplicación **SPA** está formada por las siguiente partes: **vista**, **acciones** y **estado**.
- Sin embargo, nos encontramos ante un problema cuando múltiples vistas de nuestra app dependen del mismo estado y desde diferentes puntos se realizan acciones para cambiar ese mismo estado.

Vuex

- **Vuex** se define así mismo como un **State Management Pattern**.

  npm install –save vuex-class
- Inspirado a partir de Flux y Redux.
- Sirve como un *“almacén”* centralizado para todos los componentes de nuestra aplicación.
- Utiliza reglas que garantizan que el estado sólo se puede cambiar de manera predecible y controlada.
- **Vuex** permite **extraer el estado** compartido fuera de los componentes y tratarlo como un *singleton global*.



Vuex Store

- El **store** es la parte principal de una aplicación que usa Vuex.
- Características:
  - Manteniene el **estado** de la app.
  - Reactividad en los componentes.
- Se importa en **main.ts**.

```
// store.ts
export default new Vuex.Store({
  actions: {},
  getters: {},
  mutations: {},
  state: {}
  modules: {},
});

// main.ts
import App from './App.vue';
import router from './router';
import store from './store';
import Vue from 'vue';

new Vue({
  router,
  store,
  render: (h: any) => h(App),
}).$mount('#app');

```

Vuex State

- Es nuestra única **fuente de verdad**.
- Contiene los valores que definen nuestra app en un momento en concreto.
- **Regla fundamental**:
  - los componentes **NO** deben modificar **NUNCA** directamente el estado de la **store**.
  - De esta manera podemos asegurar la integridad de los datos.

```
// En store
export class TodoStoreTypes {
  public static Store: string = 'TodoStore';
  public static Todos: string = 'todos';
}

export class Todo {
  id: number;
  text: string;
  done: boolean;
}

export interface TodoState {
  todos: Todo[];
}

const todosState: TodosState = {
  todos: [],
};

// En componente
<template>
  <div id="MyComp">
    <div v-for="todo in todos">
      <label>{{ todo.text }}</label>
    </div>
  </div>
</template>

<script lang="ts">
  import {Component, Vue} from 'vue-property-decorator';
  import {State} from 'vuex-class'

  @Component({
    name: 'my-comp',
  })
  export const MyComp extends Vue {
    @State(
      TodoStoreTypes.Todos,
      {namespace: TodoStoreTypes.Store}
    ) todos: Todo[];
  }
</script>
```

Vuex Getters

- Devuelven **datos derivados** del estado actual.
- **NO** modifican el estado.
- Pueden recibir parámetros de entrada.

```
// En store
import {GetterTree} from 'vuex';

export class TodoStoreTypes {
  public static Store: string = 'TodoStore';
  public static Todos: string = 'todos';
  public static FinishedTodos: string = 'finishedTodos';
}

const todosGetters: GetterTree<TodosState, any> = {
  [TodoStoreTypes.FinishedTodos](state: TodosState): Todo[] {
    return state.todos.filter(f => f.done);
  },
};

// En componente
<template>
  <div id="MyComp">
    <h1>Finished</h1>
    <div v-for="todo in finishedTodos">
      <label>{{ todo.text }}</label>
    </div>
  </div>
</template>

<script lang="ts">
  import {Component, Vue} from 'vue-property-decorator';
  import {Getter} from 'vuex-class'

  @Component({
    name: 'my-comp',
  })
  export const MyComp extends Vue {
    @Getter(
      StoreTypes.FinishedTodos,
      {namespace: TodoStoreTypes.Store}
    ) finishedTodos: Todo[];
  }
</script>
```

Vuex Mutations

- **ÚNICA** forma de alterar el estado del store.
- Similar a lanzar un evento.
- Cada mutación tiene un **type** y un **handler**.
- **Type** es una cadena que permite identificar univocamente un método.
- El **handler** es una función que recibe como parámetro el *estado actual* (state) y opcionalmente un *payload* (dato).
- Las mutaciones deben ser **síncronas**.
- **NUNCA** debe ser llamada directamente dentro de una función **asíncrona**.
- Se puede llamar directamente desde un componente, pero lo aconsejable es seguir el flujo de **Vuex**.
  - Llamar a una **action** que a su vez llama a un **mutation**.

```
// En store
import {MutationTree} from 'vuex';

export class TodoStoreTypes {
  public static Store: string = 'TodoStore';
  ...
  public static AddTodo: string = 'addTodo';
  public static RemoveTodo: string = 'removeTodo';
  public static UpdateTodos: string = 'updateTodos';
}

const todosMutations: MutationTree<TodosState> = {
  [TodoStoreTypes.AddTodo]:
    (state: TodosState, todo: Todo) => {
      state.todos.push(todo);
    },
  [TodoStoreTypes.RemoveTodo]:
    (state: TodosState, todo: Todo) => {
      const index: number =
        state.todos.findIndex(t => t.id === todo.id);
      if (index > -1) {
        state.todos.splice(index, 1);
      }
    },
  [TodoStoreTypes.UpdateTodos]:
    (state: TodosState, _todos: Todos) => {
      Vue.set(state, 'todos', _todos);
    },
};
```

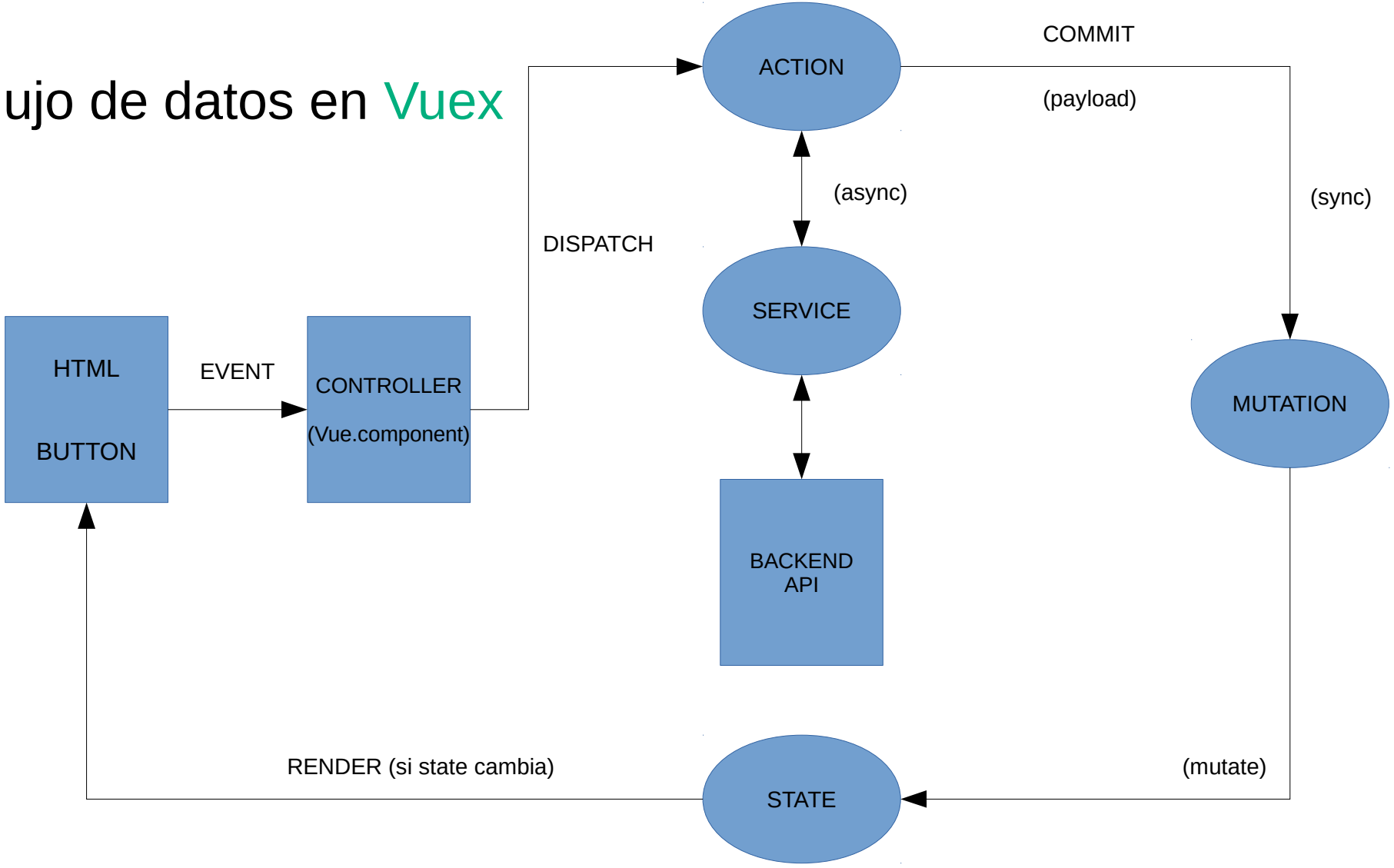
Vuex Actions

- Se utilizan (mayormente) para realizar **cambios** de forma **asíncrona** aunque también sincronamente.
- Reciben un objeto **context** que expone los mismos métodos y propiedades que la **store**.
- No modifican directamente el estado, sino emiten mutaciones.
  - **context.commit('mutationType', payload?)**

```
// En store
import {ActionTree} from 'vuex';
import Api from '../services/api';

export class TodoStoreTypes {
  public static Store: string = 'TodoStore';
  ...
  public static GetTodos: string = 'getTodos';
  public static RemoveTodo: string = 'removeTodo';
  public static UpdateTodos: string = 'updateTodos';
}

const todosActions: ActionTree<TodoState, any> = {
  [TodoStoreTypes.GetTodos]: (context) => {
    Api.getAllTodos().then(response => {
      const todos: Todo[] = response.data;
      context.commit(TodoStoreTypes.UpdateTodos, todos);
    });
  },
  [TodoStoreTypes.RemoveTodo]: (context, todo: Todo) => {
    Api.removeTodo(todo).then(response => {
      const todo: Todo = response.data;
      context.commit(TodoStoreTypes.RemoveTodo, todo);
    });
  },
};
```

```vue
// En componente
<template>
  <div id="MyComp">
    <div v-for="todo in todos">
      <label>{{ todo.text }}</label>
      <i v-bind:class="[todo.done ? 'fas fa-check' : 'fas fa-times']"></i>
      <button @click="removeTodo(todo)" title="Remove todo">
        <i class="fa fa-trash-alt"></i>
      </button>
    </div>
  </div>
</template>

<script lang="ts">
  import {Component, Vue} from 'vue-property-decorator';
  import {Action, State} from 'vuex-class'

  @Component({
    name: 'my-comp',
  })
  export const MyComp extends Vue {
    @State(TodoStoreTypes.Todos,{namespace: TodoStoreTypes.Store})
      public todos: Todo[];
    @Action(TodoStoreTypes.GetTodos, {namespace: TodoStoreTypes.Store})
      public getTodos!: () => void;
    @Action(TodoStoreTypes.RemoveTodo, {namespace: TodoStoreTypes.Store})
      public removeTodo!: (todo: Todo) => void;

    mounted() {
      this.getTodos();
    }

  }
</script>

```

Flujo de datos en Vuex



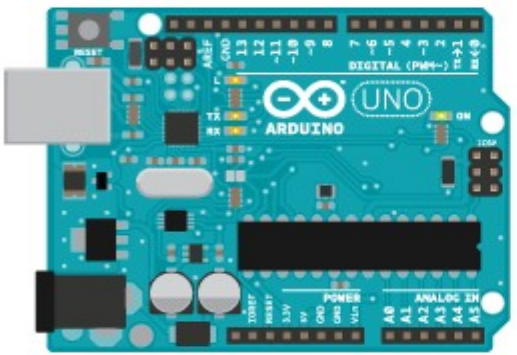
Vuex Modules

- Nuestra app puede crecer y crecer, así que gestionar todos los datos desde una misma **Store** no es lo más recomendable...
- Vuex proporciona **módulos** donde cada uno de ellos contiene su propio *store*, *mutations*, *actions*, *getters* y módulos anidados.
- Por defecto, si no se indica, las mutaciones, acciones y getters se registran globalmente, de esta manera varios módulos podrían reaccionar a la vez ante la misma acción / mutación / getter, es por ello que:
  - El atributo **namespaced** con valor **true** registrará el módulo a un único espacio de nombres.
- Un módulo puede disparar un *action / mutation / getter* de otro módulo diferente.
  - **context.dispatch('type/moduleType', payload, {root: true})**

```
export const TodosStore: Module<TodoState, any> = {
  actions: todosActions,
  getters: todosGetters,
  mutations: todosMutations,
  namespaced: true,
  state: todosState,
};
```

```
import {TodosStore} from '@/store/todos.store';

export default new Vuex.Store({
  actions: {},
  getters: {},
  mutations: {},
  state: {}
  modules: {
    TodosStore,
  },
});
```

Sí, mucho Vue + Vuex…
¿y si lo vemos en un ejemplo real?…



```
{
  "action": "light",
}
```

Puerto serie
/dev/ttyACM0

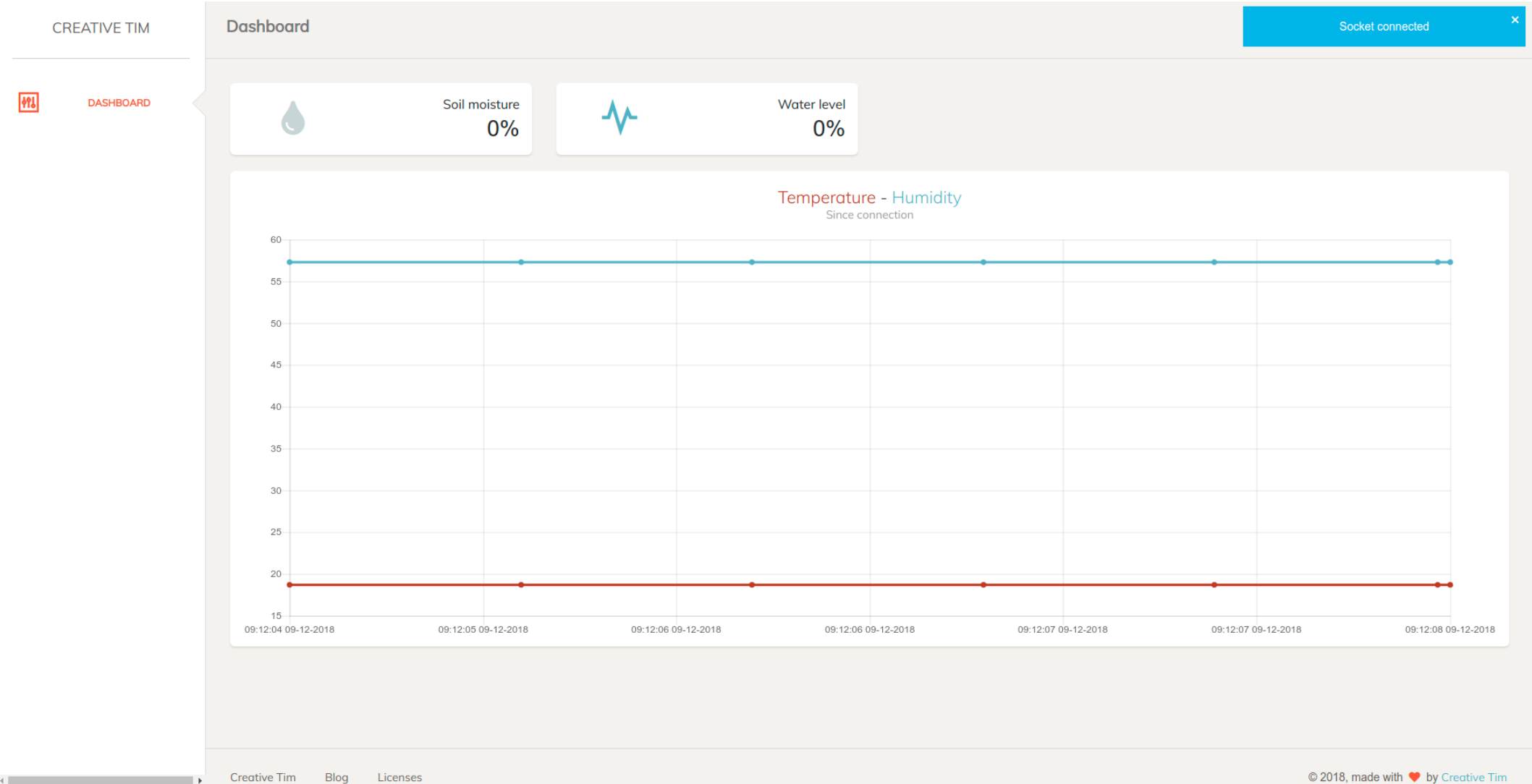
nest
puerto 3000

Websocket
puerto 3001

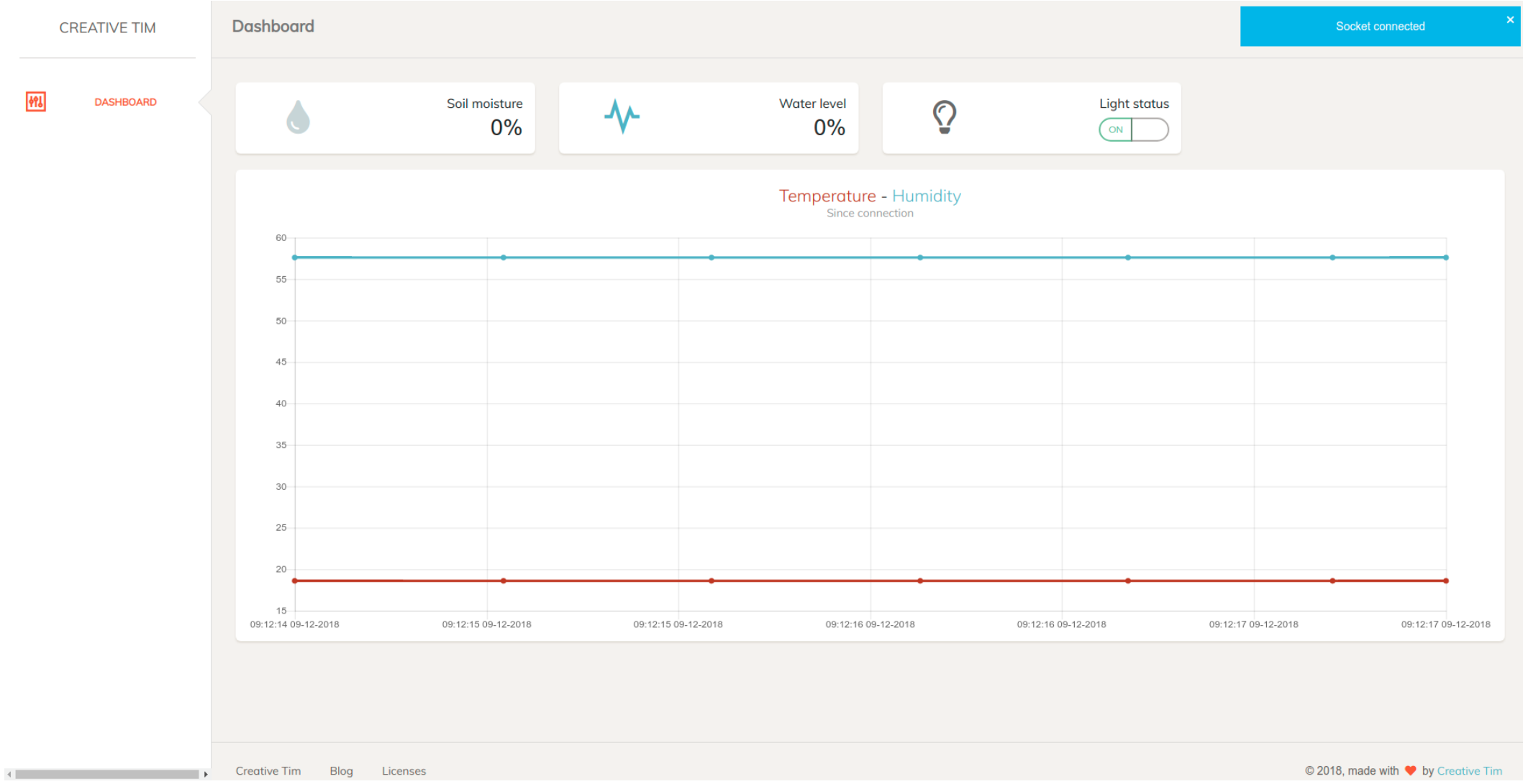
Puerto 8080

```
{
  "humidity": 50,
  "temperature": 19.5,
  "light": false,
  "higrometer": 0,
  "waterLevel": 3.6,
}
```

Frontend con Vue - I



Frontend con Vue - II



Crear **componente** y conectarlo con la **Store**:

- Dentro del directorio **src/components/** crear un fichero **.vue**.
- A partir de la plantilla original, seleccionar el componente que nos interese, inspeccionamos el elemento, copiamos el código html y lo pegamos en la sección **<template>** del fichero **.vue**.
- Cambiar el apartado **script** del componente de la siguiente manera:

```
<script lang="ts">
  import {Component, Vue} from 'vue-property-decorator';

  @Component({
    name: 'light-state',
  })
  export default class LightState extends Vue {

  }
</script>
```

- Añadir la lógica de control dentro de la clase y adaptar el código html copiado anteriormente a nuestra necesidad:
  - Añadir un **getter** en la store para consultar el estado del led.
  - Añadir un **action** en la store para cambiar el estado del led.
  - Añadir el **getter** y el **action** en el componente para mapear los métodos del store con el componente.
  - Añadir la lógica de control en el template para que represente el estado del led.
  - Importar el componente en la vista.
  - Probar (y cruzar los dedos)...

https://github.com/xino1010/vue-nest-arduino