



**Escuela Politécnica Superior de
Alicante**

Diseño de aplicaciones web

Taller ASP.net CORE + Vue.js

Contenidos

- App sencilla: gestión de listas
 - CRUD Listas
 - CRUD Tareas
- Uso de Componentes, filtros, computed methods
- Vuex, patrón de almacenamiento global
- Back-end:
 - Almacenamiento en BD
 - Llamadas a la API
- Ejercicios
- Referencias y ampliar información

Inicio de la implementación

- Ahora que ya tenemos el entorno de desarrollo preparado para comenzar a programar desarrollaremos nuestro proyecto de una aplicación sencilla de gestión de listas de tareas.
- Debemos tener en cuenta que trabajaremos realmente con DOS proyectos de implementación:
 - Back-end a partir de un proyecto de ASP.net Core MVC
 - Front-end a partir de un proyecto de Vue.js con vue-CLI
- Además, para desarrollar, tendremos que arrancar ambos proyectos, el de back-end en modo depuración y el de fornt-end en modo 'watch'.

App sencilla: gestión de listas

- Nuestro proyecto consistirá en una aplicación que nos permita generar listas de tareas.
- Cada lista tendrá: id, nombre, color, visibilidad (Sí/No), fecha de creación y una colección (array) de tareas.
- Para cada tarea, almacenaremos: id, texto, fecha creación, fecha límite/caducidad, terminada (Sí/No) y visible (Sí/No)
- Crearemos un CRUD para las listas (listar, detalle, añadir, eliminar y modificar) y por cada lista un CRUD (en formato cabecera/lineas) para sus tareas.
- Usaremos componentes necesarios para el almacenamiento, acceso al back-end, trabajo con fechas, iconografía, etc.

Listado y añadir nuevo

- Con el fin de dotar de las estructuras de datos necesarias, vamos a crear dos modelos en el front-end (dentro de una nueva carpeta llamada 'src/models'), que llamaremos:

TaskList.ts

```
import ItemTask from '@/models/ItemTask';

export default class TaskList {
  id:number=0;
  nombre:string="";
  color:string="#FFFFFF";
  visible:boolean=true;
  fecha:string="";
  tasks:ItemTask[]=[];
}
```

ItemTask.ts

```
export default class ItemTask {
  id:number=0;
  texto:string="";
  fecha:string="";
  caduca:string="";
  terminada:boolean=false;
  visible:boolean=true;
}
```

Listado y añadir nuevo

- En el componente 'MisListas' vamos a ubicar un listado de listas de tareas y toda la estructura necesaria para el resto de operaciones, por tanto vamos a diseñar un layout con los siguientes elementos:
 - Un botón que nos servirá para acceder al formulario de añadir nueva lista.
 - Un 'fieldset' con un 'div' de tipo 'card' (de bootstrap) para cada lista que encontramos (dentro de un bucle 'v-for')
 - En caso de no tener listas, mostraremos un 'div' de tipo 'alert' con un mensaje adecuado.
 - Añadiremos otro 'fieldset' con el formulario de nueva lista (de momento, solo el campo nombre) y dos botones:
 - para realizar la inserción de la nueva lista
 - o para cancelar y volver al listado.

Listado y añadir nuevo

- Primero cambiaremos el nombre de la propiedad 'txt' por 'título' (no olvidar modificar la invocación desde 'Home')
- El código del template quedará:

```
<template>
  <div>
    <h1>{{ titulo }}</h1>
    <button v-if="mode != 'add'" v-on:click="addMode()" class="btn btn-primary">
      {{$t('add.btnAdd')}}</button>

    <div class="container" v-if="mode == 'list'">
      <fieldset>
        <legend>$t('list.titulo')</legend>
        <div class="card" v-for="(list) in lists" v-bind:key="list.id">
          <div class="card-body">{{list.nombre}}</div>
        </div>
        <div v-if="lists.length == 0" class="alert alert-warning">
          {{$t('list.noLists')}}</div>
      </fieldset>
    </div>
```

Listado y añadir nuevo

- El código del template quedará: (...continuación)

```
<div class="container" v-if="mode=='add'>
  <fieldset>
    <legend>{{$t('add.titulo')}}</legend>
    <form>
      <div class="form-group">
        <label for="addListNombre">{{$t('add.lblNombre')}}</label>
        <input v-model=" newList.nombre " type="text"
          class="form-control" id="addListNombre"
          v-bind:placeholder="$t('add.ttpNombre')">
      </div>
    </form>
    <div class="btn-group">
      <button v-on:click="addList()" class="btn btn-primary">
        {{$t('add.btnAddList')}}
      </button>
      <button v-on:click="listMode()" class="btn btn-danger">
        {{$t('app.btnAtras')}}
      </button>
    </div>
  </fieldset>
</div>
</template>
```

Listado y añadir nuevo

- La sección de scripts:

```
<script lang="ts">
import { Component, Prop, Vue } from 'vue-property-decorator';
import TaskList from '@/models/TaskList';
@Component
export default class Mislistas extends Vue {
  @Prop() private titulo!: string;
  mode="list";
  lists:TaskList[]=[];
  newList:TaskList=new TaskList();
  addMode():void {
    let nextId= this.lists.length>0?this.lists[this.lists.length-1].id+1:+1;
    this.newList = new TaskList();
    this.newList.id= nextId;
    this.mode="add";
  }
  listMode():void {
    this.mode="list";
  }
  addList():void {
    this.lists.push(this.newList);
    this.mode = 'list';
  }
}
</script>
```

Listado y añadir nuevo

- No olvidar añadir los textos localizados.
- Así quedaría el fichero para los textos en castellano:

```
{  
    "app": {  
        "appName": "Mis listas",  
        "titulo": "Mis listas de tareas",  
        "inicio": "Inicio",  
        "acerca_de": "Acerca de",  
        "btnAtras": "Volver"  
    },  
    "acerca_de": {  
        "titulo": "Sobre esta App",  
        "text": "Aplicación de gestión de mis listas de tareas"  
    },  
    "add": {  
        "titulo": "Añadir nueva lista",  
        "lblNombre": "Nombre: ",  
        "tttNombre": "Nombre de la lista",  
        "btnAdd": "Nueva lista",  
        "btnAddList": "Crear"  
    },  
    "list": {  
        "titulo": "Mis listas",  
        "noLists": "No has creado ninguna lista todavía"  
    }  
}
```

Listado y añadir nuevo

- Vamos a comprobar que el campo nombre tiene contenido antes de añadir la lista a nuestro almacen:
- Para ello añadiremos en el template, una referencia al campo nombre y una caja para mostrar el error.

```
<div class="form-group">
  <label for="addListNombre">{{$t('add.lblNombre')}}</label>
  <input ref=" newList_nombre" v-model=" newList.nombre" type="text"
    class="form-control" id="addListNombre"
    v-bind:placeholder="$t('add.ttpNombre')">
    <span v-if="errList.nombre" class="small text-danger">
      |   |   {{$t('add.errNombre')}}}
    </span>
```

- En los ficheros de textos hay que añadir:
 - "add"."errNombre": "Debe escribir un nombre de la lista."

Listado y añadir nuevo

- Comprobar que el campo nombre tiene contenido:
- En el código, añadiremos un objeto para gestionar los errores del formulario, que inicializaremos a 'false'.
- También haremos un cast al tipo 'HTMLFormElement' de la referencia anterior:

```
export default class Mislistas extends Vue {  
  $refs!: {  
    newList_nombre: HTMLFormElement  
  }  
  @Prop() private titulo!: string;  
  mode="list";  
  lists:TaskList[]=[ ];  
  newList:TaskList=new TaskList();  
  errList= {nombre: false};  
  addMode():void {  
    let nextId= this.lists.length>0?this.lists[this.lists.length-1].id+1:+1;  
  
    this.newList = new TaskList();  
    this.newList.id= nextId;  
    this.mode="add";  
    this.errList.nombre=false;  
  }  
}
```

Listado y añadir nuevo

- Comprobar que el campo nombre tiene contenido:
- También en el código, además modificaremos la función 'addList', para comprobar si el campo nombre tiene valor y actuar en consecuencia:

```
addList():void {
    if (this newList.nombre != "") {
        this.lists.push(this newList);
        this.mode = 'list';
    } else {
        this errList.nombre = true;
        this $refs newList_nombre.focus();
    }
}
```

Añadir más datos a las listas

- Vamos a ampliar la funcionalidad de nuestra aplicación permitiendo gestionar, para una lista dada los siguientes datos adicionales:
 - un color de fondo (campo: color)
 - una fecha de alta (campo: fecha)
 - Un campo para hacerla visible u ocultarla (campo: visible)

Color de la lista

- Para permitir establecer un color de fondo para cada lista:
 - Añadimos un nuevo recurso de texto localizado:
Add.lblColor: "Color de la lista:"
 - Luego modificaremos el formulario de añadir una lista con el nuevo campo:

```
<form>
  <div class="form-group">
    <label for="addListNombre">{{$t('add.lblNombre')}}</label>
    <input ref=" newList_nombre" v-model=" newList.nombre" type="text"
      class="form-control" id="addListNombre"
      v-bind:placeholder="$t('add.ttpNombre')">
    <span v-if="errList.nombre" class="small text-danger">
      {{$t('add.errNombre')}}
    </span>
  </div>
  <div class="form-group">
    <label for="addListColor"> {{$t('add.lblColor')}}</label>
    <input v-model=" newList.color" type="color" class="form-control"
      id="addListColor">
  </div>
</form>
```

Color de la lista

- Y para terminar modificamos en la plantilla, el código HTML que muestra una lista, añadiendo un estilo CSS a la caja de tipo 'card':

```
<div class="container" v-if="mode=='list'>
  <fieldset>
    <legend>$t('list.titulo')</legend>
    <div class="card" v-for="(list) in lists" v-bind:key="list.id"
        v-bind:style="{ 'background-color': list.color }">
      <div class="card-body">$list.nombre</div>
    </div>
    <div v-if="lists.length==0" class="alert alert-warning">
      $t('list.noLists')
    </div>
  </fieldset>
</div>
```

Fecha de creación de la lista

- Ahora, gestionaremos la fecha de alta de una lista, para ello, instalaremos la librería moment.js en nuestro proyecto:
 - Ejecutaremos desde una consola ubicada en la carpeta de nuestro proyecto:
 - **Npm install vue-moment**
- Debemos incluir la librería en la instancia principal, para ello en 'main.ts' añadiremos:
 - **Vue.use(require('vue-moment'))**
 - Y en todo componente donde usemos la librería, deberemos referenciarla con:
 - **import moment from 'moment';**

Fecha de creación de la lista

- Vamos a modificar los dos modelos, cambiando el tipo de los campos que contengan fechas:
- En ambos ficheros incluiremos la librería moment:
 - **import moment from 'moment';**
- Para 'TaskList.ts' tendremos:
 - **fecha:moment.Moment=moment();**
- Para 'ItemTask.ts', cambiaremos dos campos:
 - **fecha:moment.Moment=moment();**
 - **caduca:moment.Moment=moment();**

Fecha de creación de la lista

- En el método de añadir una nueva lista, pondremos la siguiente linea de código adicional:

```
addList():void {
    if (this newList.nombre != "") {
        this newList.fecha = moment();
        this lists.push(this newList);
        this mode = 'list';
    } else {
        this errList.nombre = true;
        this $refs newList_nombre.focus();
    }
}
```

Fecha de creación de la lista

- Añadiremos un filtro local (solo accesible desde el componente 'MisListas') para formatear las fechas y horas en un formato legible.
- Este tipo de filtros se deben declarar en el decorador @Component.

```
@Component({
  filters: {
    formatDate: function (value: any) {
      //TODO, verificar que hay algo en value!
      return value.format("DD/MM/YYYY hh:mm:ss");
    }
  }
})

export default class Mislistas extends Vue {
  $refs!: {
    newList_nombre: HTMLFormElement
  }
  @Prop() private titulo!: string;
```

Fecha de creación de la lista

- Y, finalmente, mostramos la fecha en la vista o template, dentro del bucle 'v-for':

```
<div class="container" v-if="mode=='list' ">
  <fieldset>
    <legend>$t('list.titulo')</legend>
    <div class="card" v-for="list in lists" v-bind:key="list.id"
         v-bind:style="{ 'background-color': list.color }">
      <div class="card-body">
        <h5>{{list.nombre}}</h5>
        <small>{{list.fecha | formatDate}}</small>
      </div>
    </div>
    <div v-if="lists.length==0" class="alert alert-warning">
      $t('list.noLists')
    </div>
  </fieldset>
</div>
```

Ocultar o mostrar una lista

- Ahora vamos a añadir la funcionalidad que permita usar el campo 'visible' con el fin de ocultar o mostrar nuestras listas.
- Primero de todo, vamos a añadir un plugin al proyecto desde el interface visual: **vue ui**.
- Accedemos a 'complementos → agregar complemento' y buscamos 'font awesome'.
- Seleccionamos 'vue-cli-plugin-fontawesome' y clicamos en instalar.

Ocultar o mostrar una lista

- Instalación del plugin 'vue-cli-plugin-fontawesome':

Agregar un complemento

Buscar Configuración Archivos modificados

fontawe

Complementos

Dependencies

Configuración

Tareas

taller-fe

vue-cli-plugin-fontawesome 0.3.1
vue-cli 3 plugin for fontawesome ↓ 3.4K GregYankovoy Más información

@crazydos/vue-cli-plugin-fontawesome 0.1.1
vue-cli plugin to setup vue-fontawesome ↓ 0 shunnNet Más información

Cancelar

Instalar vue-cli-plugin-fontawesome

- Para comprobar si está instalado correctamente: ver si en 'main.ts' tenemos: import './plugins/fontawesome'

Ocultar o mostrar una lista

- Vamos a añadir dos botones en la vista/template justo al lado del nombre de la lista:

```
<div class="card-body">
  <div class="tituloLista">
    {{list.nombre}}
    <button v-if="list.visible==true" v-on:click="visibleList(index)"
      class="btn btn-sm btn-info" v-bind:title="$t('list.ttpNoVisible')">
      <font-awesome-icon icon="eye-slash"></font-awesome-icon>
    </button>
    <button v-if="list.visible==false" v-on:click="visibleList(index)"
      class="btn btn-sm btn-info" v-bind:title="$t('list.ttpVisible')">
      <font-awesome-icon icon="eye"></font-awesome-icon>
    </button>
  </div>
```

- El estilo 'tituloLista' lo declaramos en la sección 'styles':

```
<style scoped>
  .tituloLista { font-size: 2em; }
</style>
```

Ocultar o mostrar una lista

- Añadiremos dos recursos de texto en la sección 'list':
 - ttpNoVisible: 'Ocultar lista'
 - ttpVisible: 'Mostrar lista'
- Añadimos en el código el evento 'visibleList':

```
visibleList(index:number) {
    this.lists[index].visible = (this.lists[index].visible ? false : true);
}
```

Ocultar o mostrar una lista

- Y modificaremos el listado
 - Moviendo el 'v-for' a un elemento 'template'.
 - Añadiendo el contador del bucle 'index' al 'v-for'
 - Y poniendo un v-if a cada lista para ocultar las no visibles.

```
<template v-for="(list, index) in lists">
  <div class="card" v-bind:key="list.id"
    v-bind:style="{ 'background-color': list.color }" v-if="list.visible">
    <div class="card-body">
      <div class="tituloLista">
        {{list.nombre}}
        <button v-if="list.visible==true" v-on:click="visibleList(index)"
          class="btn btn-sm btn-info" v-bind:title="$t('list.ttpNoVisible')">
          <font-awesome-icon icon="eye-slash"></font-awesome-icon>
        </button>
        <button v-if="list.visible==false" v-on:click="visibleList(index)"
          class="btn btn-sm btn-info" v-bind:title="$t('list.ttpVisible')">
          <font-awesome-icon icon="eye"></font-awesome-icon>
        </button>
      </div>
      <small>{{list.fecha | formatDate}}</small>
    </div>
  </div>
</template>
```

Eliminar lista

- Para la operación de eliminar una lista:
 - Añadiremos un nuevo recurso en los ficheros de textos, en unanueva sección 'delete': { 'ttpBtn': 'Eliminar' }
 - Ahora, modificaremos la caja de tipo 'card-body' para añadir un botón de borrar y agrupar los botones existentes.

```
<div class="card-body">
  <div class="tituloLista">
    {{list.nombre}}
    <div class="btn-group">
      <button v-on:click="deleteList(index)" class="btn btn-sm btn-danger" v-bind:title="$t('delete.ttpBtn')">
        <font-awesome-icon icon="trash"></font-awesome-icon>
      </button>
      <button v-if="list.visible==true" v-on:click="visibleList(index)" class="btn btn-sm btn-info" v-bind:title="$t('list.ttpVisibleTrue')">
        <font-awesome-icon icon="eye-slash"></font-awesome-icon>
      </button>
      <button v-if="list.visible==false" v-on:click="visibleList(index)" class="btn btn-sm btn-info" v-bind:title="$t('list.ttpVisibleFalse')">
        <font-awesome-icon icon="eye"></font-awesome-icon>
      </button>
    </div>
  </div>
  <small>{{list.fecha | formatDate}}</small>
</div>
```

Eliminar lista

- En el código del componente, añadimos la función 'deleteList':

```
deleteList(list:number) {  
    this.lists.splice(list, 1);  
}
```

Filtrar listas visibles/ocultas/todas

- Vamos a añadir el patrón de tablas de filtrar datos por una faceta o atributo, en este caso, filtraremos por el campo 'visible' de la lista.
- Añadiremos tres recursos de texto en 'list':
 - optTodas: "Ver todas",
 - optVisibles: "Ver visibles",
 - optNOVisibles: "Ver NO visibles"
- Ademas, necesitaremos una variable nueva en el código: slctVisible, que inicializaremos a 'T' (Ver todas).

Filtrar listas visibles/ocultas/todas

- Modificamos la vista y añadimos un 'select' encima del listado con tres opciones:
 - Ver todas (valor:'T')
 - Ver visibles (valor: 'V')
 - Ver NO visibles (valor: 'N')
- Ademas, la lista (caja tipo card) se mostrará o no según una función de filtrado: getVisibility.

```
<legend>{{$t('list.titulo')}}</legend>
<div class="row justify-content-end">
    <select class="form-control col-md-3" v-model="slctVisible">
        <option value="T">{{$t('list.optTodas')}}</option>
        <option value="V">{{$t('list.optVisibles')}}</option>
        <option value="N">{{$t('list.optNOVisibles')}}</option>
    </select>
</div>
<template v-for="(list, index) in lists">
    <div class="card" v-bind:key="list.id" v-if="getVisibility(index)" v-bind:style="{ 'background-color': list.color }">
```

Filtrar listas visibles/ocultas/todas

- El método 'getVisibility':

```
getVisibility(index:number) {  
    let res = true;  
    switch (this.slctVisible) {  
        case "V":  
            res = this.lists[index].visible;  
            break;  
        case "N":  
            res = !this.lists[index].visible;  
            break;  
        default:  
            res = true;  
            break;  
    }  
    return res;  
}
```

Ejercicios

- A partir del proyecto realizado durante el taller:
 - Permitir que sea el propio usuario al añadir una lista, quien aporte la fecha de creación, usad un DatePicker.
 - Añadir la posibilidad de poder ordenar las listas por su nombre o su fecha de creación
 - Añadir la posibilidad de modificar una lista mediante un botón de modificar:
 - que muestre un formulario similar al de añadir, con los datos de la lista a editar
 - que al confirmar la modificación vuelva al listado con los datos cambiados
 - Validar los datos antes de realizar el cambio y mostrar mensajes de error si hiciera falta.

Vuex: patrón de almacenamiento global

- Vamos a usar Vuex para gestionar las listas de nuestro aplicativo y así tenerlas en un almacen de ámbito global.
- Como ya instalamos 'vuex' al crear el proyecto, no tenemos que hacer ninguna configuración adicional.
- De hecho, tenemos una carpeta llamada 'src/store' y dentro de ella un fichero 'index.ts' con una definición básica o esqueleto del 'store'.
- Además en el fichero 'main.ts', se puede ver como se ha importado el módulo de 'store' y como se incluye al declarar la instancia principal de Vue.

Vuex: patrón de almacenamiento global

- En el fichero 'src/store/index.ts', en la sección 'state', vamos a añadir un array de objetos 'TaskList', y el import necesario.
- Comprobaremos que tenemos todas las secciones necesarias:
 - State
 - Getters
 - Mutations
 - Actions
 - modules

```
import Vue from 'vue'
import Vuex from 'vuex'

import TaskList from "@/models/TaskList";

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
    listas: Array<TaskList>()
  },
  getters: {
  },
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})
```

Vuex: patrón de almacenamiento global

- La sección de 'getters' contendrá una función por cada una de las operaciones de lectura que hacemos contra el almacen de datos desde el componente 'MisListas': getNextId y getVisible, además de dos getters nuevos: getAll y getCount

```
getters: {
    getAll(state) {
        return state.listas;
    },
    getCount(state) {
        return state.listas.length;
    },
    getNextId(state) {
        return state.listas.length>0?state.listas[state.listas.length-1].id+1:1;
    },
    getVisible: (state) => (index: number) => {
        return state.listas[index].visible;
    }
},
```

Vuex: patrón de almacenamiento global

- En la sección 'mutations' añadiremos las operaciones de modificación del 'store' que teníamos en nuestra aplicación:
 - Añadir lista: add
 - Eliminar lista: del
 - Cambiar visibilidad: setVisible

```
mutations: {
  add(state, lista: TaskList) {
    state.listas.push(lista);
  },
  del(state, index: number) {
    state.listas.splice(index, 1);
  },
  setVisible(state, index:number) {
    state.listas[index].visible = (state.listas[index].visible ? false : true);
  }
},
```

Vuex: patrón de almacenamiento global

- Ahora vamos a adaptar el componente 'MisListas' para que use el 'store' de Vuex en lugar de la variable local 'listas'.
- En el template, modificaremos la colección de datos del v-for por una llamada a 'getAll()'
- El mensaje de 'no hay listas' usará una función: 'getCount'

```
<template v-for="(list, index) in getAll()">
  <div class="card" v-bind:key="list.id" v-if="getVisibility(index)"
    v-bind:style="{ 'background-color': list.color }">
    ...
  </div>
</template>
<div v-if="getCount()==0" class="alert alert-warning">
  {{ $t('list.noLists') }}
</div>
```

Vuex: patrón de almacenamiento global

- En el código del componente realizaremos varios cambios.
- Quitamos la variable local 'listas'
- Definimos dos métodos nuevos: getAll y getCount.

```
export default class Mislistas extends Vue {  
    $refs!: {  
        newList_nombre: HTMLFormElement  
    }  
    @Prop() private titulo!: string;  
    mode="list";  
    //VUEX: lists:TaskList[]=[];  
    newList:TaskList=new TaskList();  
    errList= {nombre: false};  
    slctVisible='T';  
    getAll() {  
        return this.$store.getters.getAll;  
    }  
    getCount() {  
        return this.$store.getters.getCount;  
    }  
}
```

Vuex: patrón de almacenamiento global

- Cambios en el código del componente.
- La función 'addMode' usará el getter 'getNextId':

```
addMode():void {  
    //VUEX: let nextId= this.lists.length>0?this.lists[this.lists.length-1].id+1:+1;  
    let nextId= this.$store.getters.getNextId;  
    this newList = new TaskList();
```

- La función 'AddList' invocará al mutador 'add':

```
addList():void {  
    if (this newList.nombre != "") {  
        this newList.fecha = moment();  
        //VUEX: this.lists.push(this newList);  
        this.$store.commit('add', this newList);  
        this.mode = 'list';
```

Vuex: patrón de almacenamiento global

- Cambios en el código del componente.
- También cambiarán 'deleteList', 'visibleList' y 'getVisibility':

```
deleteList(list:number) {
    //VUEX: this.lists.splice(list, 1);
    this.$store.commit('del', list);
}

visibleList(index:number) {
    //VUEX: this.lists[index].visible = (this.lists[index].visible ? false : true);
    this.$store.commit('setVisible', index);
}

getVisibility(index:number) {
    let res = true;
    switch (this.slctVisible) {
        case "V":
            //VUEX: res = this.lists[index].visible;
            res = this.$store.getters.getVisible(index);
            break;
        case "N":
            //VUEX: res = !this.lists[index].visible;
            res = !this.$store.getters.getVisible(index);
            break;
        default:
            res = true;
            break;
    }
}
```

Almacenamiento en BD

- Vamos a crear una BD para almacenar los datos de nuestra aplicación y a usar Ingeniería Inversa y para conectarnos a la misma:
 - Crearemos una BD en SQL-Server Express con dos tablas:
 - TaskLists (id PK, nombre, fecha, visible, color)
 - ItemTasks (id PK, texto, fecha, caduca, visible, terminada, lista FK→TaskLists.id)
 - Crearemos un login para acceder al SQL-Server
 - Añadiremos unos datos de ejemplo (varias listas con sus tareas)
- Podeis obtener un script de creación de esta BD en el moodle.

Almacenamiento en BD

- Tal y como hicimos en el anterior taller (de ASP.net CORE) tenemos que instalar varios paquetes necesarios:
 - 'Microsoft.EntityFrameworkCore.Tools', versión >5.0
 - 'Microsoft.EntityFrameworkCore.SqlServer', versión >5.0
- Y usamos la herramienta 'Scaffold-DbContext' para crear los modelos basados en las tablas de nuestra BD:

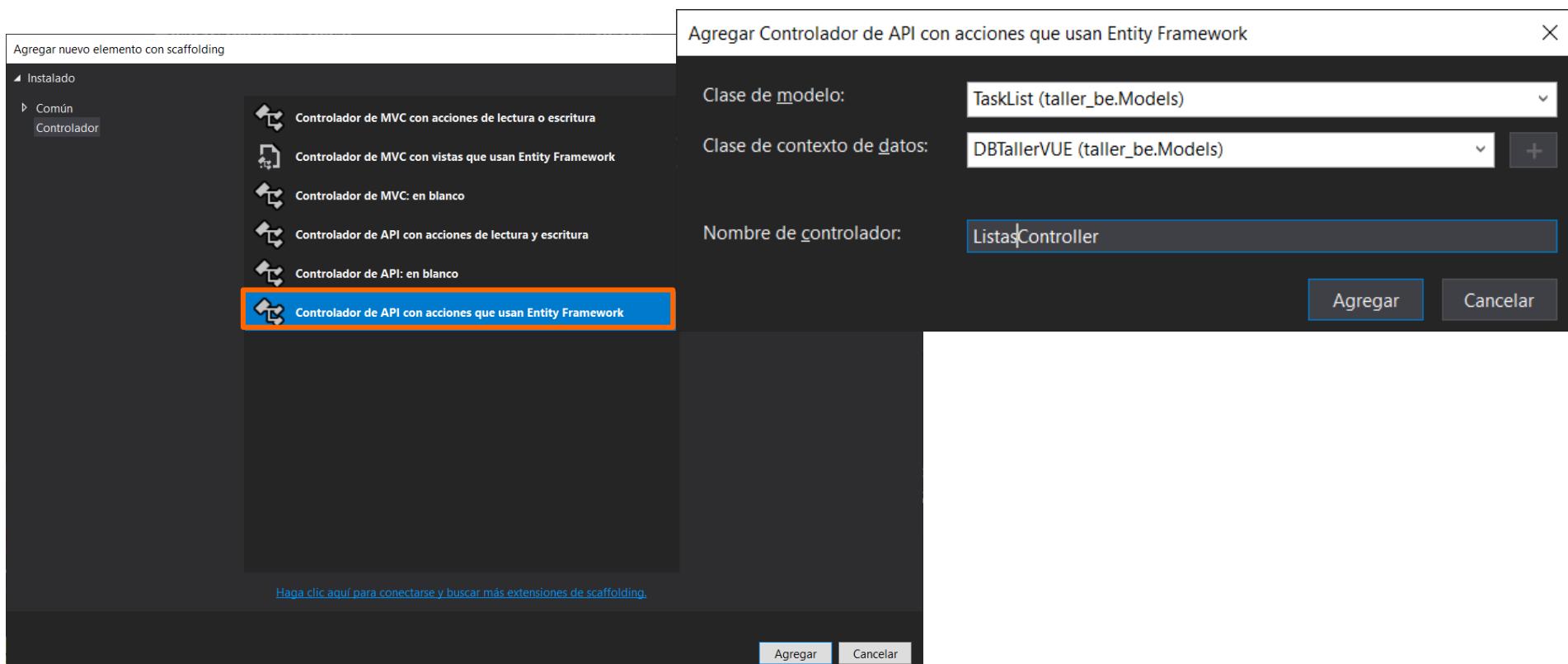
```
Scaffold-DbContext "Server=(local)\sqlexpress;  
Database=NUESTRABD; Integrated Security=False;  
Persist Security Info=False; User ID=USUARIO;  
Password=#####"
```

```
Microsoft.EntityFrameworkCore.SqlServer  
-Tables TaskLists, ItemTasks -OutputDir Models  
-Context DBTallerVUE
```

- Hay que añadir el contexto en 'ConfigureServices' del fichero 'Startup.cs':
 - `services.AddDbContext<Models.BDTallerCore>();`

Llamadas a la API

- Ahora crearemos un controlador API con scaffolding que incluya acciones del Entity Framework.
- Basado en el modelo 'TaskList', usando el contexto 'DBTallerVUE' llamado 'APIListasController'.



Llamadas a la API

- El nuevo controlador 'APIListasController' tiene todas las funciones de un CRUD típico:

```
[Route("api/[controller]")]
[ApiController]
1 referencia
public class APIListasController : ControllerBase
{
    private readonly DBTallerVUE _context;

    0 referencias
    public APIListasController(DBTallerVUE context)...

    // GET: api/APIListas
    [HttpGet]
    0 referencias
    public async Task<ActionResult<IEnumerable<TaskList>>> GetTaskLists()...

    // GET: api/APIListas/5
    [HttpGet("{id}")]
    0 referencias
    public async Task<ActionResult<TaskList>> GetTaskList(decimal id)...
```

Llamadas a la API

- Continuación...:

```
// PUT: api/APIListas/5
// To protect from overposting attacks, enable the specific properties you want to
// more details, see https://go.microsoft.com/fwlink/?LinkId=2123754.
[HttpPut("{id}")]
0 referencias
public async Task<IActionResult> PutTaskList(decimal id, TaskList taskList)...
```



```
// POST: api/APIListas
// To protect from overposting attacks, enable the specific properties you want to
// more details, see https://go.microsoft.com/fwlink/?LinkId=2123754.
[HttpPost]
0 referencias
public async Task<ActionResult<TaskList>> PostTaskList(TaskList taskList)...
```



```
// DELETE: api/APIListas/5
[HttpDelete("{id}")]
0 referencias
public async Task<ActionResult<TaskList>> DeleteTaskList(decimal id)...
```



```
1 referencia
private bool TaskListExists(decimal id)...
```

Llamadas a la API

- Primera operación del CRUD: listado.
- Modificaremos el servicio añadiendo una nueva llamada GET que llamaremos 'getRaw':

```
async getRaw(ruta:string, id: string="") {  
    let dato="";  
    let resp = new APIResponse();  
    if (id!="") {  
        dato = "/" + id;  
    }  
    await http.get(  
        ruta + dato  
    ).then((respuesta) => {  
        resp.status=APIStatus.OK;  
        resp.respuesta=respuesta.data;  
    }).catch((error) => {  
        resp.status=APIStatus.ERR;  
        resp.error = error.toString();  
    });  
    return resp;  
}
```

Llamadas a la API

- Primera operación del CRUD: listado.
- Modificaremos el 'store' de Vuex (store/index.ts):
 - Creamos una **acción**: 'getLists', que accederá al back-end, concretamente al end-point: 'api/APIListas' con el método GET.

```
actions: {
  getLists({ commit }) {
    serviceAPI.getRaw("APIListas").then(r=> {
      console.log(r);
      if (r.status==APIStatus.OK) {
        commit('setLists', r.respuesta)
      } else {
        this.state.error=r.error;
      }
    }).catch(e=> {
      this.state.error=e;
   ));
  }
}
```

Llamadas a la API

- Primera operación del CRUD: listado.
- Modificaremos el 'store' de Vuex (store/index.ts):
 - Añadimos un nuevo campo en el 'store': error, un string que inicializaremos a cadena vacía.
 - Y un getter para obtener el posible mensaje de error:

```
    ...
  getError(state) {
    return state.error;
  }
}
```

- Añadimos una 'mutation' que cargará las listas obtenidas en la acción anterior en el 'state' del almacenamiento.

```
  mutations: {
    setLists(state, lists) {
      state.listas = lists;
    },
  }
```

Llamadas a la API

- Primera operación del CRUD: listado.
- En el componente 'MisListas', añadiremos el método 'mounted' para obtener las listas de la BD nada más cargar la aplicación
- Tambien dos métodos para gestionar errores.

```
mounted() {
  this.$store.dispatch("getLists");
}
hayError() {
  return this.$store.getters.getError!="";
}
getError() {
  return this.$store.getters.getError;
}
```

Llamadas a la API

- Siguiente operación: Añadir lista
- Para esta acción, necesitamos adaptar un poco el controlador del back-end.
- Añadimos el atributo 'FromBody' al argumento.

```
// POST: api/APIListas
// To protect from overposting attacks, enable the specific properties you want to bind to, for
// more details, see https://go.microsoft.com/fwlink/?LinkId=2123754.
[HttpPost]
0 referencias
public async Task<ActionResult<TaskList>> PostTaskList([FromBody]TaskList taskList)
{
    _context.TaskLists.Add(taskList);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetTaskList", new { id = taskList.Id }, taskList);
}
```

Llamadas a la API

- Siguiente operación: Añadir lista
- Ahora añadimos una nueva función en el servicio de Llamadas a la API para invocar un post.

```
async post(ruta:string, dato: any) {  
    let resp = new APIResponse();  
  
    await http.post(  
        ruta, dato  
    ).then((respuesta) => {  
        resp.status=APIStatus.OK;  
        resp.respuesta=respuesta.data;  
    }).catch((error) => {  
        resp.status=APIStatus.ERR;  
        resp.error = error.toString();  
    });  
    return resp;  
}
```

Llamadas a la API

- Siguiente operación: Añadir lista
- Modificaremos el 'store' para añadir una nueva acción, 'addList'. Adaptaremos los datos para que no den problemas en el back-end.

```
addList({ commit }, lista) {
    lista.fecha = lista.fecha.format("YYYY-MM-DDTHH:mm:ss");
    lista.visible = lista.visible?"S":"N";
    serviceAPI.post("APIListas", lista).then(r=> {
        if (r.status==APIStatus.OK) {
            commit('add', r.respuesta)
        } else {
            this.state.error=r.error;
        }
    }).catch(e=> {
        this.state.error=e;
    });
}
```

Llamadas a la API

- Siguiente operación: Añadir lista
- En el componente 'MisListas', modificaremos el método 'addMode', eliminando algunas asignaciones:

```
addMode():void {
    //VUEX: let nextId= this.lists.length>0?this.lists[this.lists.length-1].id+1:+1;
    //API: let nextId= this.$store.getters.getNextId;
    this newList = new TaskList();
    //API: this.newList.id= nextId;
    this.mode="add";
    this.errList.nombre=false;
}
```

Llamadas a la API

- Siguiente operación: Añadir lista
- En el componente 'MisListas', modificaremos el método 'addList' para invocar a la acción anterior:

```
addList():void {
    if (this newList.nombre != "") {
        this newList.fecha = moment();
        //VUEX: this.lists.push(this newList);
        //API: this.$store.commit('add', this newList);
        this.$store.dispatch("addList", this newList);
        this.mode = 'list';
    } else {
        this errList.nombre = true;
        this.$refs newList_nombre.focus();
    }
}
```

Llamadas a la API

- Gestión de errores: para que los mensajes de error no se confundan entre las diferentes llamadas, vamos a añadir una serie de cambios en el componente 'MisListas'.
- En el código del componente declararemos dos variables nuevas y modificaremos 'mounted' y 'addList'.

```
errGet = "";
errAdd = "";
mounted() {
  this.$store.dispatch("getLists");
  this.errGet = "";
  if (this.hayError()) {
    this.errGet = this.getError();
  }
}
```

```
addList():void {
  if (this newList.nombre != "") {
    this newList.fecha = moment();
    //VUEX: this.lists.push(this newList);
    //API: this.$store.commit('add', this newList);
    this.$store.dispatch("addList", this newList);
    this.errAdd = "";
    if (this.hayError()) {
      this.errAdd = this.getError();
    } else {
      this.mode = 'list';
    }
  } else {
    this.errList.nombre = true;
    this.$refs newList_nombre.focus();
  }
}
```

Llamadas a la API

- Gestión de errores: en el template también modificaremos los mensajes de error y las condiciones para que se muestren:

```
<div class="alert alert-danger" v-if="errGet!=''>Error: {{errGet}}</div>

<div class="container" v-if="errGet=='' && mode=='list'">
  <fieldset>
    ...
  </fieldset>
</div>

<div class="container" v-if="errGet=='' && mode=='add'">
  <fieldset>
    <legend>$t('add.titulo')</legend>
    <form>
      ...
      <div class="alert alert-danger" v-if="errAdd!=''>Error: {{errAdd}}</div>
    </fieldset>
  </div>
</div>
```

Ejercicios

- A partir del proyecto ampliado con el store de Vuex y el acceso a la BD:
 - Añadir las funcionalidades que faltan de eliminar lista y cambiar visibilidad para que operen contra la BD mediante el back-end.
 - Añadir el código necesario para que la opción de modificar se gestione desde el store global de Vuex.
 - Añadir, además el end-point necesario para que esta funcionalidad se vea reflejada en la base de datos mediante un servicio del back-end.
- Cread un nuevo componente 'Lista.vue' que encapsule la lógica de negocio y de datos de una lista individual y modificar el componente 'MisListas' para que lo use.

Referencias y más información

- Vue: <https://vuejs.org/>
- Vue-CLI:
 - <https://cli.vuejs.org/guide/>
 - <https://vuejs.org/v2/guide/typescript.html>
- Componentes y plugins
 - <https://www.npmjs.com/package/vue-moment>
 - <https://www.npmjs.com/package/vue-cli-plugin-fontawesome>