Para crear un proyecto empleado este framework que hemos desarrollado debemos realizar los siguientes pasos:

(Comprobar que tenemos instalado nodeJS y typescript)

1. Crear una carpeta nueva y desde la línea de comandos ejecutar:

(Comprobar que tenemos instalado nodeJS: node -v)

npm install typescript

2. Creamos un proyecto:

tsc --init

Y configuramos tsconfig.json con el código que se incluye a continuación:

{

"compilerOptions": {

/\* Basic Options \*/

"target": "es6",

"module": "commonjs",

"outDir": "./dist",

"strict": false,

"resolveJsonModule": true,

"esModuleInterop": true,

"experimentalDecorators": true

},

"include": [

"./src/\*\*/\*.ts",

]

}

3. Creamos la carpeta src para crear los archivos controladores y demás módulos typescript. Además, dentro de esta carpeta deberá encontrarse la carpeta developodoFW que contiene los archivos del framework.

(creamos package.json con npm create)

4. Si deseamos crear aplicaciones con interfaz web instalaremos el plugin que paquetera webpack:

npm install webpack webpack-cli webpack-dev-server

npm install -D ts-loader

5. Configuramos el archivo webpack.config.json con el siguiente código:

const path = require('path');

module.exports = {

entry: path.join(\_\_dirname, '/src/index.ts'),

output: {

filename: 'index.js',

path: \_\_dirname,

libraryTarget: 'var',

library:'developodo'

},

module: {

rules: [

{

test: /\.tsx?$/,

loader: 'ts-loader',

exclude: /node\_modules/,

},

]

},

resolve: {

extensions: [".tsx", ".ts", ".js"]

},

devServer: {

port:8081,

proxy: {

'/items': 'http://localhost:8080'

}

}

};

Nota 1: el nombre con el que exportamos nuestro paquete es developodo.

Nota 2: hemos cambiado al puerto 8081. Es útil si deseamos ejecutar otras aplicaciones en nodeJs en otros puestos, como por ejemplo el servidor RestFul en el puerto 8080.

Nota 3: el campo proxy evita los problemas con los Intercambios de Recursos de Origen Cruzado (CORS). Estas excepciones ocurren cuando se lanzan peticiones a otros dominios, puertos y protocolos distintos. En nuestro caso, tenemos el servicio Rest en otro puerto, para evitar este tipo de problemas realizamos un encaminamiento proxy en la ruta /items que nos redirecciona las peticiones al servicio REST (puerto 8080) sin que el navegador se percate.

6. Creamos el archivo package.json e instalamos las dependencias (si faltase alguna):

npm init -y //CREATE PACKAGE.JSON

npm install

Vamos a package.json y modificamos el archivo para que podamos ejecutar nuestro cliente con el comando npm start (paquetiza automáticamente) :

"scripts": {

"start": "webpack-dev-server --mode development"

},

Para ejecutar nuestro cliente de pruebas:

npm start

De esta forma se crea un archivo index.js en la raíz y se sirve por el puerto 8080. Para poder visualizarlo correctamente, tendríamos que tener un archivo index.html en la raíz con el siguiente código:

<!DOCTYPE html>

<html>

<head>

<meta charset='utf-8'>

<meta http-equiv='X-UA-Compatible' content='IE=edge'>

<title>My RestFull Cliente with FrameWork</title>

<meta name='viewport' content='width=device-width, initial-scale=1'>

<link rel="stylesheet" type="text/css" href="bootstrap.css"/>

</head>

<body>

<app>

<nav class="navbar navbar-light bg-light">

<a class="navbar-brand" href="#">

<img src="https://raw.githubusercontent.com/Developodo/home/master/developodomini.png" width="30" height="30" class="d-inline-block align-top" alt="">

Typescript + Bootstrap (by Developodo)

</a>

</nav>

<!--

It is recommended not to use javascript on HTML files

However, here, there is an example that shows how to connect HTML with our CONTROLLER.

For more information, see export section of index.ts and webpack.config.json

-->

<button type="button" class="btn btn-primary float-left m-1" onclick="developodo.indexpage.refresh()">Refresh</button>

<button type="button" id="myaddbutton" class="btn btn-primary float-right m-1" onclick="developodo.indexpage.addData()">Add Item</button>

<!-- Table Data -->

<table id="data" class="table table-striped">

<thead>

<tr>

<th scope="col">#</th>

<th scope="col">Title</th>

<th scope="col">Descripion</th>

<th scope="col">Actions</th>

</tr>

</thead>

<tbody>

<!-- here observable binding, see our custom framework documentation -->

</tbody>

</table>

<!-- Modal witout Framework tools-->

<div class="modal fade" id="addModal" tabindex="-1" role="dialog" aria-labelledby="addModalLabel" aria-hidden="true">

<div class="modal-dialog" role="document">

<div class="modal-content">

<div class="modal-header">

<h5 class="modal-title" id="addModalLabel">{{editaddTitle}}</span> item</h5>

<button type="button" class="close" data-dismiss="modal" aria-label="Close">

<span aria-hidden="true">&times;</span>

</button>

</div>

<form id="addForm">

<div class="modal-body">

<div id="addModalContent">

<input type="hidden" name="idItem" id="idItem" value="">

<div class="form-group">

<label for="title">Title</label>

<input type="text" class="form-control" id="title" aria-describedby="emailHelp" placeholder="Enter Title">

</div>

<div class="form-group">

<label for="description">Description</label>

<input type="text" class="form-control" id="description" placeholder="Enter Description">

</div>

</div>

</div>

<div class="modal-footer">

<button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>

<button type="submit" class="btn btn-primary">{{editaddButton}}</button>

</div>

</form>

</div>

</div>

</div>

<!-- Example of double way binding -->

<footer class="col-xs-1 text-center">We have double way binding with title field from modal editing or adding item -> value: <kbd> [{{title}}]</kbd></footer>

</app>

<script src='index.js'></script>

</body>

</html>

7. Para generar el archivo index.js (en la raíz del proyecto) que nos permita portarlo a una web:

npx webpack

Nota 1: si deseamos que se incluyan las hojas de estilo de bootstrap debemos tener en la carpeta raíz el archivo bootstrap.css que se encuentra en la carpeta node\_modules (hay que tener cuidado con las rutas a la hora de importarlo).

Nota 2: para tener disponible todo el ejemplo en funcionamiento deberemos instalar las siguientes librerías:

npm install jquery --save

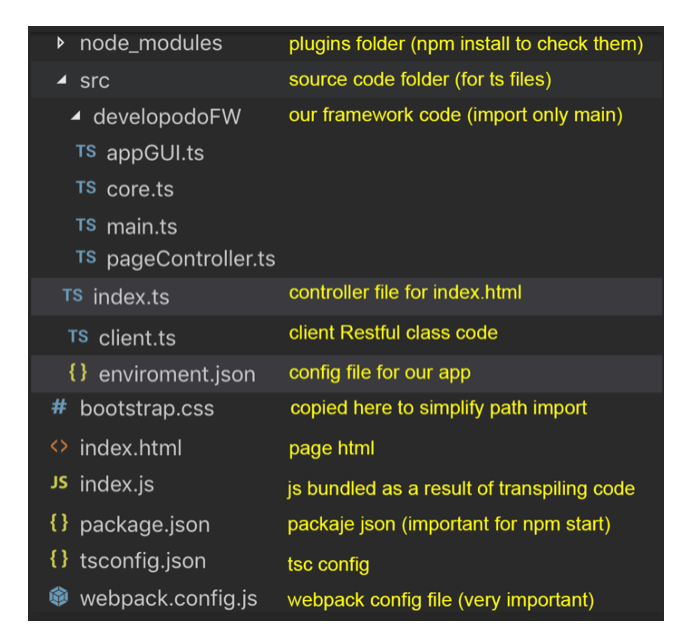
npm install bootstrap --save

npm install popper.js --save

npm install xhr2 //used by client.ts

npm i rxjs

La estructura de archivos debería quedar de la siguiente forma:



#### Código fuente

El código de client.ts sería el mismo que tenemos en el apartado anterior:

/\*\*

\* File: client.ts

\* Description: Client for API RestFul Service (CRUD)

\* @autor: Carlos Serrano Sánchez for Developodo

\*/

import { Observable, of } from "rxjs";

import { ajax } from "rxjs/ajax";

import { map, catchError } from "rxjs/operators";

/\*\*

\* The only class to be exported: clientRestful

\*/

export class clientRestful {

private endPoint: string; //url of service

private XHR2: any; //method to implement XMLHttpRequest on Web and NodeJS

/\*\*

\* @param endP url of api restful, if not present it sets an endpoint by default

\*/

constructor(endP?: string) {

if (endP) {

this.endPoint = endP;

} else {

this.endPoint = "http://localhost/items";

}

//On NodeJS doesn't exist XMLHttpRequest, so if we want to create a client on server side we have to

//provide the xhr2 library: https://www.npmjs.com/package/xhr2

this.XHR2 =

typeof XMLHttpRequest !== "undefined" ? XMLHttpRequest : require("xhr2"); //hack to emulate XMLHttpRequest on NodeJS

}

/\*\*

\* It sends POST request with title and description fields: CREATE

\* @param newItemTitle title to new Item

\* @param newItemDescription description to new Item

\* @returns Observable<any> the response from API

\*/

createItem(

newItemTitle: string,

newItemDescription: string

): Observable<any> {

return ajax({

url: this.endPoint,

createXHR: () => new this.XHR2(),

method: "POST",

crossDomain: true,

hasContent: true,

headers: {

"Content-Type": "application/x-www-form-urlencoded; charset=utf-8" //application/json

},

body: {

title: newItemTitle,

description: newItemDescription

}

}).pipe(

map(response => {

return response.response;

}),

catchError(error => {

return of(error);

})

);

}

/\*\*

\* it calls getItemById without any parameters: reading all items

\*/

getItems() : Observable<any> {

return this.getItemById();

}

/\*\*

\* It sends a GET request to Server to READ items

\* @param id optional parameter to get items.If it is ommited, all items are retrieved.

\*/

getItemById(id?: any): Observable<any> {

return ajax({

url: id ? this.endPoint + "?id=" + id : this.endPoint,

createXHR: () => new this.XHR2(),

crossDomain: true,

async: true,

headers: {

"Content-Type": "application/x-www-form-urlencoded; charset=utf-8" //application/json

},

method: "GET"

}).pipe(

map(response => {

return response.response; //it maps the output, getting the response field of the object

}),

catchError(error => {

return of(error); //it captures the error and return an observable (according to function definition) of error message

})

);

}

/\*\*

\* Similar to getItemById but filtering by title field

\* @param title title parameter to filter items

\*/

getItemByTitle(title: any): Observable<any> {

return ajax({

url: this.endPoint + "?title=" + title,

createXHR: () => new this.XHR2(),

crossDomain: true,

method: "GET"

}).pipe(

map(response => {

return response.response;

}),

catchError(error => {

return of(error);

})

);

}

/\*\*

\* It sends a PUT request to UPDATE the item

\* @param id id of item to be updated

\* @param newItemTitle new title to update selected item

\* @param newItemDescription new description to update selected item

\*/

updateItem(

id: any,

newItemTitle: string,

newItemDescription: string

): Observable<any> {

return ajax({

url:

this.endPoint +

"/" +

id +

"?title=" +

newItemTitle +

"&description=" +

newItemDescription,

createXHR: () => new this.XHR2(),

method: "PUT",

crossDomain: true,

headers: {

"Content-Type": "application/x-www-form-urlencoded; charset=utf-8" //application/json

}

}).pipe(

map(response => {

return response.response;

}),

catchError(error => {

return of(error);

})

);

}

/\*\*

\* It sends a DELETE request to DELETE the selected item

\* @param id id of item to be removed

\*/

removeItem(id: any): Observable<any> {

return ajax({

url: this.endPoint + "/" + id,

createXHR: () => new this.XHR2(),

crossDomain: true,

async: true,

method: "DELETE",

headers: {

"Content-Type": "application/x-www-form-urlencoded; charset=utf-8" //application/json

}

}).pipe(

map(response => {

return response.response;

}),

catchError(error => {

return of(error);

})

);

}

}

A continuación, se incluyen el resto de código de los archivos ts:

appGUI.ts

/\*\*

\* Package: developodoFW, framework implemented in typescript to develop frontend web app

\* File: appGUI.ts

\* Description: it defines some utilities to craate web components

\* @autor: Carlos Serrano Sánchez for Developodo

\* @see pageController.ts

\* @see core.ts

\*/

import {$} from './core';

/\*\*

\* this class is instanciated in core class: App, so every app has all this methods available with sintax: this.gui.method()

\*/

export class AppGUI {

/\*\*

\* showLoading creates an on top div semitransparent with a spinner

\*/

showLoading() {

this.hideLoading();

let template = `<div class="develop-loading d-flex justify-content-center align-self-center"

style="width:100%;height:100%;position:absolute;top:0;left:0;z-index:9999;background-color: rgba(255,255,255,0.5);padding-top:45%">

<div class="spinner-border text-primary" role="status">

<span class="sr-only">Loading...</span>

</div>

</div>`;

$("body").append(template);

}

/\*\*

\* hideLoading hides the loading div

\* @see showLoading

\*/

hideLoading() {

$(".develop-loading").remove();

}

/\*\*

\* showToast creates a toast box on left bottom corner (if there are more than one, the new one is stacked bellow the older ones).

\* Toast are hidden automatically after 4 seconds.

\* @param msg the message to be shown on toast component

\*/

showToast(msg: string) {

if (!$(".toast-container").length) {

let container = `<div class="toast-container" aria-live="polite" aria-atomic="true" style="min-height: 200px;">

<div style="position: absolute; bottom: 0; right: 0;">

<!-- Then put toasts within -->

</div>

</div> `;

$("app").append(container);

}

let toastCode = Math.floor(Math.random() \* 1000000) + 1;

let template = `<div class="toast toast${toastCode}" role="alert" aria-live="assertive" aria-atomic="true">

<div class="toast-header">

<img src="https://raw.githubusercontent.com/Developodo/home/master/developodomini.png" class="rounded mr-2" alt="" style="width:20px">

<strong class="mr-auto">Developodo</strong>

<button type="button" class="ml-2 mb-1 close" data-dismiss="toast" aria-label="Close">

<span aria-hidden="true">&times;</span>

</button>

</div>

<div class="toast-body">

${msg}

</div>

</div>`;

$(".toast-container > div").append(template);

$(".toast").toast({

autohide: true,

delay: 4000

});

setTimeout(() => {

$(".toast" + toastCode).remove();

}, 4500); //0.5 seconds more to show de hiding animation and then toast is auto-destroyed

$(".toast").toast("show");

}

/\*\*

\* showModal creates a window modal on top of the page with the following parameters

\* @param title Title for the modal window

\* @param content HTML code to be inserted in the modal content div

\* @param positiveNameButton Message shown on the accept button

\* @param callbackPositive Function to be called when accept button is pressed

\*/

showModal(

title: string,

content: string,

positiveNameButton: string,

callbackPositive: any

) {

let modalCode = Math.floor(Math.random() \* 1000000) + 1;

let template: string = `<div id="modal${modalCode}" class="modal fade" tabindex="-1" role="dialog" aria-hidden="true">

<div class="modal-dialog" role="document">

<div class="modal-content">

<div class="modal-header">

<h5 class="modal-title">${title}</h5>

<button type="button" class="close" data-dismiss="modal" aria-label="Close">

<span aria-hidden="true">&times;</span>

</button>

</div>

<div class="modal-body">

${content}

</div>

<div class="modal-footer">

<button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>

<button type="button" class="btn btn-danger" id="modalButton${modalCode}">${positiveNameButton}</button>

</div>

</div>

</div>

</div>`;

$("app").append(template);

$("#modalButton" + modalCode).click(() => {

callbackPositive();

$("#modal" + modalCode).modal("hide");

});

$("#modal" + modalCode).modal("show");

}

}

core.ts

/\*\*

\* Package: developodoFW, framework implemented in typescript to develop frontend web app

\* File: core.ts

\* Description: Import all dependecies: jquery, popper and bootstrap. This is the file that exported them and the defines

\* the core class App and the decorator functions.

\* @autor: Carlos Serrano Sánchez for Developodo

\* @see pageController.ts

\* @see appGUI.ts

\*/

import $ = require("jquery"); //npm install jquery --save

import "popper.js"; //npm install popper.js --save

import "bootstrap"; //npm install bootstrap --save

import {AppGUI} from './appGUI';

/\*\*

\* propertyBinding changes the value of the one HTML element's property when the variable assigned has new a new value.

\* @param field id of the HTML element

\* @param property name of the property to connect the variable assigned

\*/

function propertyBinding(field: string,property:string) {

return function(target: any, propertyKey: string) {

let value = target[propertyKey]; //read de value of this variable

Object.defineProperty(target, propertyKey, {

//changing descriptor

get: () => {

return value;

},

set: newValue => {

value = newValue;

$(field).attr(property,value);

}

});

};

}

/\*\*

\* twoWayBinding connect the value of an input field to a typescript variable bidirectionally

\* @param field id of the HTML element connected to

\*/

function twoWayBinding(field: string) {

App.bindOneWay(field); //it allows onewaybinding too

return function(target: any, propertyKey: string) {

let value = target[propertyKey]; //read de value of this variable

Object.defineProperty(target, propertyKey, {

get: () => {

return value;

},

set: newValue => {

value = newValue;

if ($(".tmpl-" + propertyKey)) $(".tmpl-" + propertyKey).html(value);

$("#" + propertyKey).unbind("change"); //avoid infinite loop on next command

$("#" + propertyKey).val(value);

$("#" + field).change(x => {

value = x.target.value;

if ($(".tmpl-" + propertyKey)) $(".tmpl-" + propertyKey).html(value);

});

}

});

};

}

/\*\*

\* App core class -> onewaybinding and GUI tools

\*/

class App{

gui:AppGUI; //All controller class can access to gui in this way -> this.gui.showLoading(); , for example

p: any;

static executed: boolean = false;

constructor(){

this.gui=new AppGUI();

}

binding(p: any) {

this.p = p;

if (!App.executed) { //if programmers define two (or more) Controller class (mistake) avoid executing binding method more than one time

this.bindAllOneWay();

App.executed = true;

}

}

//This method is used by Decorators, programmers shouldn't use it

static bindOneWay(field: string,container?:string) {

container=(container?container:"app");

let web = $(container).html();

var replace = "{{(" + field + ")}}";

var rex = new RegExp(replace, "g");

web = web.replace(rex, '<span class="tmpl-$1"></span>');

$(container).html(web);

}

/\*\*

\* bindAllOneWay creates the connection between mustache sintax and variables (of controller class and initializated)

\* @param container optional, it defines the html container to analize, app by default

\*/

bindAllOneWay(container?:string) {

container=(container?container:"app");

let web = $(container).html();

for (let item of Object.getOwnPropertyNames(this.p)) {

if (

typeof this.p[item] === "string" ||

typeof this.p[item] === "number" ||

typeof this.p[item] === "boolean"

) {

var replace = "{{(" + item + ")}}";

var rex = new RegExp(replace, "g");

web = web.replace(rex, '<span class="tmpl-$1"></span>');

}

}

$(container).html(web);

for (let item of Object.getOwnPropertyNames(this.p)) {

if (

typeof this.p[item] === "string" ||

typeof this.p[item] === "number" ||

typeof this.p[item] === "boolean"

) {

Object.defineProperty(this.p, item, {

//changing descriptor

set: newValue => {

$(".tmpl-" + item).html(newValue);

}

});

}

}

}

}

export { App, twoWayBinding,propertyBinding, $ };

pageController.ts

/\*\*

\* Package: developodoFW, framework implemented in typescript to develop frontend web app

\* File: pageController.ts

\* Description: it defines the Controller abstract class that must be imported and implemented by all pages of the app

\* How to use it:

\* The page controller must import main.ts file and implement this class.

\* The page controller contruct must have the two first lines:

\* constructor() {

\* super();

\* this.init();

\* }

\* The page controller must implement the method

\* OnInit() {} that executes its command after page is loaded

\* OnUnload(){} that executes its commands before leaving the page

\* The page controller can use binding tools:

\* -OneWayBinding: controller variables are directly shown on GUI. If they chage, GUI changes. GUI and

\* Controller are connected unidirectionally: C -> G

\* A) How to define them:

\* In Controller class must be declared and inicializated as properties of class.

\* class myController extends pageController{

\* myOneWayVariable:string='Hello World';

\* ...

\* constructor(){

\* ...

\* In View File (HTML) must be attached using mustache syntax:

\* <div>{{myOneWayVariable}}</div>

\* B) How to use them:

\* Just change the value of the variable and it the new value will be rendered on GUI.

\* -DoubleWayBindng: controller variables are linked with input fields of GUI. If the variable changes,

\* then input value changes. If an user change the value of the input, directly the

\* controller variable has the new value. GUI and Controller are connected bidirectionally: C <-> G

\* A) How to define them:

\* In Controller class must be declared and inicializated as properties of class. (see OneWayBinding).

\* But the @twoWayBinding decorator must be used. This decorator receives a parameter: the id of the input

\* field connected to the variable.

\* class myController extends pageController{

\* @twoWayBinding('idOfMyInput')

\* myTwoWayVariable:string='Hello World';

\* ...

\* constructor(){

\* ...

\* In View File (HTML) must exist and input field with the selected id:

\* <input type="text" id="idOfMyInput" value="">

\* It can be attached using mustache syntax, as the OneWayBinding variabels:

\* {{myTwoWayVariable:}}

\* B) How to use them:

\* Just change the value of the variable and the new value will be rendered on GUI.

\* Just change the input value and the variable value will change.

\* -PropertyBinding: controller variables can be connected to properties of elements, such as class, disabled, id, name, style...

\* If, the variable changes, the property changes. GUI and Controller are connected unidirectionally: C -> G

\* A) How to define them:

\* In Controller class must be declared and inicializated as properties of class. (see TwoWayBinding).

\* But the @propertyBinding decorator must be used. This decorator receives two parameter: the id of html element

\* and the name of the parameter to be binded connected to the variable.

\* class myController extends pageController{

\* @propertyBinding('idOfMyInput','disabled')

\* mypropertyVariable:boolean=true;

\* ...

\* constructor(){

\* ...

\* In View File (HTML) must exist and input field with the selected id:

\* <input type="text" id="idOfMyInput" value="">

\* B) How to use them:

\* Just change the value of the variable and the new value will be applied to the property of the selected element.

\* For example: this.mypropertyVariable=false makes the input becomes enabled

\* ----

\* HOW TO CREATE A NEW PAGE

\* 1) Copy de framework folder (developodoFW) and install the dependencies:

\* npm install jquery --save npm install popper.js --save npm install bootstrap --save

\* npm install webpack webpack-cli webpack-dev-server —save-dev npm install -D ts-loader

\* 2) tsc init -> and config tsconfig.json

\* npm init -y //create package.json npm install (to make sure all modules and libraries are installed)

\* Config webpack.config.json (npm start -> to simulate or npx webpack -> to build the bundle js)

\* @see tsconfig.json, package.json and webpack.config.json examples.

\* 3) Create a view file: html and in the body add the app tag. If you want to bind something must be inside this tag.

\* <body>

\* <app>

\* <!-- here your html code -->

\* </app>

\* </app>

\* <!-- include the bundle js build by webpack -->

\* <script src='index.js'></script>

\* 4) Create a controller file: index.ts:

\* Import main file of framework in this way:

\* import { pageController, twoWayBinding,propertyBinding, $ } from "./developodoFW/main";

\* Create a controller class as it is explained on top of this file.

\* class IndexPage extends pageController { --- }

\* Create an instance of the controller class

\* let indexpage = new IndexPage();

\* If you want to bind the methods of your controller class in the html code, just export the class (last lines):

\* export = {

\* indexpage

\* };

\* Now you can access from HTML: <button onclick="developodo.indexpage.myMethod()"></button>

\* But before, you have to configure webpack.config.json library:

\* output: {

\* filename: 'index.js',

\* path: \_\_dirname,

\* library:'developodo' //<-change it if you want to

\* },

\* 5) To test your page execute: npm start (if tsconfig.json and webpackage.config.json are corrected)

\* http://localhost:8080

\* 6) To build your bundle file JS: index.js, execute: npx webpack

\*

\* @autor: Carlos Serrano Sánchez for Developodo

\* @see core.ts

\* @see appGUI.ts

\*/

import {App,$,propertyBinding,twoWayBinding} from './core';

/\*\*

\* Main class of controller, exteds the core class App for binding tools

\*/

abstract class pageController extends App {

constructor() {

super();

//Config the OnUnload method to be executed on leaving the page

$(window).on("unload", () => {

this.OnUnload();

});

}

//This method is called in the constructor of controller page

init() {

this.binding(this); //analize DOM to inject onewaybinding code

this.OnInit(); //Execute the OnInit method at the beginning

}

abstract OnInit();

abstract OnUnload();

}

export { pageController, propertyBinding,twoWayBinding,$ };

main.ts

//BYPASS FILE to simplify importing process in page controllers

//MAIN FILE <- IMPORT ONLY THIS TO CREATE AN APP

export {pageController, propertyBinding,twoWayBinding,$ } from './pageController';

enviroment.json

{

"description":"This files is used to set configurable parameters. It's imported in page controllers",

"endPoint":"http://localhost:8081/items/"

}

Nota: vamos a emplear como servidor rest el desarrollado en NodeJS de los apartados anteriores. Se despliega en el puerto 8080, pero vamos a considerarlo como puerto 8081 para que no de problemas de seguiridad (CORS) y redireccionar la petición con el proxy de webpack.

index.ts

/\*\*

\* File: index.ts

\* Description: it creates the controller class from indexPage (index.html) and executes it

\* @autor: Carlos Serrano Sánchez for Developodo

\* @see developodoFW

\*/

/\*\*

\* Importing core frameworks tools:

\* pageController -> all binding utilities and oninit and unload methods

\* twoWayBinding -> decorator for two way binding

\* propertyBinding -> decorator for properties binding

\* $ -> jQuery tools

\*/

import { pageController, twoWayBinding,propertyBinding, $ } from "./developodoFW/main";

/\*\*

\* Importing config file as enviroment object

\*/

import \* as enviroment from './enviroment.json';

/\*\*

\* Importing client for Restful Service

\*/

import { clientRestful } from "./client";

/\*\*

\* Importing Subject for example purpose

\*/

import { Subject } from "rxjs";

/\*\*

\* item interface represents our data model.

\* All data objects in database fit with this interface

\*/

interface item {

id: number;

title: string;

description: string;

}

/\*\*

\* Every page has a file (this) with a class with the name {namePage}Page,

\* IndexPage in this case, that must extend the core class pageController.

\* This class is the main controller of our GUI(View): index.html in this case.

\*/

class IndexPage extends pageController {

/\*\*

\* Every one way binding must be declared as attribute of the controller class

\* and must be initialized (IMPORTANT). That allows us to use mustache code to

\* insert this variables into View, for example <button>{{editaddButton}}</button>

\* becomes <button>Add</button> in this default case.

\*/

editaddButton: string = "Add";

editaddTitle: string = "Adding";

/\*\*

\* If you want to make a double way binding you just have to use de Decorator

\* @twoWayBinding and the parameter: id of the input field of GUI. Only it is allowed on

\* input fields (IMPORTANT), so in the HTML must exist and input with the id name

\* defined, for exable title.

\* After the decorator, the variable binded must be declared and initialized (IMPORTANT).

\* This variable can be rendered on View using mustache sintax

\*/

@twoWayBinding("title")

title: string = "";

@twoWayBinding("description")

description: string = "";

@twoWayBinding("idItem")

idItem: string = "";

/\*\*

\* If you want to make a property binding you just have to use de Decorator

\* @propertyBindig and the parameters: id of the field of GUI and property to bind.

\* In this case, the submit button inside modal

\*/

@propertyBinding(".modal button[type=submit]","disabled")

buttonAcceptModalDisabled:boolean=false;

/\*\*

\* Other variables for controller

\* myService: instance of the client Restful Service (CRUD)

\* data: an observable over the list of items. It check if there is

\* changes to update the table on GUI.

\*/

myService: clientRestful = null;

data: Subject<item[]> = new Subject<item[]>();

/\*\*

\* Constructor has always two instructions:

\* a call to super (pageController) is needed in Typescrip.

\* the call to init() method to start core tools (binding data)

\*/

constructor() {

super();

this.init();

}

/\*\*

\* OnInit method is called automatically after contructor.

\* Put here everything to be executed on load page

\*/

OnInit() {

//Instance of Client with the endPoint URL as parameter (defined in config file)

this.myService = new clientRestful(enviroment.endPoint);

//Managing changes on data to update de table on GUI

this.data.subscribe(d => {

this.fillTable(d);

});

//Load data from RestFull Service and update data variable, so table is

//automatically updated on GUI

this.refresh();

}

/\*\*

\* OnUnLoad method is called automatically on UnLoad the page.

\* Put here everything to be executed before exiting

\*/

OnUnload(){

//Unsubscribe the data observable

this.data.unsubscribe();

}

//All the rest of methods of this CONTROLLER

/\*\*

\* addData shows an custom modal (see HTML file) with some

\* doublewaybinding variables: title, description and idItem and other

\* onewaybinding variables: editaddButton and editaddTitle

\*/

addData(): void {

//let's empty input fields of modal form by doublewaybinding

//see alternative mode without doublewaybinding

this.title = ""; //$("#title").val("");

this.description = ""; //$("#description").val("");

this.idItem = ""; //$("#idItem").val("");

//Lets enable accept button by propertyBinding

this.buttonAcceptModalDisabled=false; //$("#addModalButton").prop("disabled", false);

//As this form is used for edit and add, we cancel previus action on submit

$("#addForm").unbind("submit");

//Set forom action to add Item

$("#addForm").submit(() => {

//it disabled accept button to avoid double click and strage behaviors

this.buttonAcceptModalDisabled=true; //$("#addModalButton").prop("disabled", true);

//Some doublewaybinding maginc

let t2 = this.title; //$("#title").val();

let d2 = this.description; //$("#description").val();

if (t2 != "" && d2 != "") {

this.addDataConfirmed(t2, d2); //it uses client of service

} else {

this.buttonAcceptModalDisabled=false; //$("#addModalButton").prop("disabled", false);

alert("Fill all fields");

}

event.preventDefault();

});

//Some onewayBinding magic to set the GUI messages

this.editaddButton = "Add";

this.editaddTitle = "Adding";

//it shows the modal

$("#addModal").modal("show");

}

/\*\*

\* addDataConfirmed is called from accept button of modal shown by addData function.

\* it uses the client to connect to API RestFul and add an item

\* Then, it updates the GUI

\* @param t title of item to edit (received from modal: editData function)

\* @param d description of item to edit (received from modal: editData function)

\*/

addDataConfirmed(t: string, d: string):void {

//it uses the loading component offered by our custom framework gui tools

this.gui.showLoading();

//it uses the createItem method of the client service than send and POST request to add an item

this.myService.createItem(t, d).subscribe(r => {

//it hides the modal

$("#addModal").modal("hide");

//it reloads the GUI

this.refresh();

//it uses the toast utility implemented by our custom framework (see documentation)

this.gui.showToast("A new Item added ("+t+")");

});

}

/\*\*

\* editData shows an custom modal (see HTML file) with some

\* doublewaybinding variables: title, description and idItem and other

\* onewaybinding variables: editaddButton and editaddTitle

\* @param i id of item to edit (read from data-id)

\* @param t title of item to edit (read from data-title)

\* @param d description of item to edit (read from data-description)

\*/

editData(i: string, t: string, d: string): void {

//let's populate data to input fields of modal form by doublewaybinding

//see alternative mode without doublewaybinding

this.title = t; //$("#title").val(t);

this.description = d; //$("#description").val(d);

this.idItem = i; //$("#idItem").val(i);

//Lets enable accept button by propertyBinding

this.buttonAcceptModalDisabled=false; //$("#addModalButton").prop("disabled", false);

//As this form is used for edit and add, we cancel previus action on submit

$("#addForm").unbind("submit");

//Set forom action to edit Item

$("#addForm").submit(() => {

//it disabled accept button to avoid double click and strage behaviors

this.buttonAcceptModalDisabled=true; //$("#addModalButton").prop("disabled", true);

//Some doublewaybinding magic

let t2 = this.title; //let t2=$("#title").val(t);

let d2 = this.description; //let d2=$("#description").val(t);

if (t2 != "" && d2 != "") {

this.editDataConfirmed(i, t2, d2); //it uses client of service

} else {

alert("Fill all fields");

this.buttonAcceptModalDisabled=false; //$("#addModalButton").prop("disabled", false);;

}

event.preventDefault();

});

//Some onewayBinding magic to set the GUI messages

this.editaddButton = "Edit";

this.editaddTitle = "Editing";

//it shows the modal

$("#addModal").modal("show");

}

/\*\*

\* editDataConfirmed is called from accept button of modal shown by editData function.

\* it uses the client to connect to API RestFul and edit the item

\* Then, it updates the GUI

\* @param i id of item to edit (received from modal: editData function)

\* @param t title of item to edit (received from modal: editData function)

\* @param d description of item to edit (received from modal: editData function)

\*/

editDataConfirmed(i: string, t: string, d: string):void {

//it uses the loading component offered by our custom framework gui tools

this.gui.showLoading();

//it uses the updateItem method of the client service than send and PUT request to update an item

this.myService.updateItem(i, t, d).subscribe(r => {

//it hides the modal

$("#addModal").modal("hide");

//it reloads the GUI

this.refresh();

//it uses the toast utility implemented by our custom framework (see documentation)

this.gui.showToast("Item edited ("+t+")");

});

}

/\*\*

\* removeData shows a modal created by GUI framework tool (see documentation) with some

\* @param i id of item to remove (read from data-id)

\* @param t title of item to remove (read from data-title)

\*/

removeData(i: string, t: string): void {

//it creates a modal with parameters: title, content, function executed by accept button

this.gui.showModal("Removing Item",

"<h1>" + t + "</h1>",

"Remove it?",

()=>{

this.removeDataConfirmed(i);

} );

}

/\*\*

\* removeDataConfirmed is called by the accept button of modal shown by removeData function

\* @param i id of item to be removed

\*/

removeDataConfirmed(i: string):void {

//it uses client service removeItem function to send a DELETE request to remove an item from database

this.myService.removeItem(i).subscribe(r => {

//it removes directly from DOM, so we don't need to refresh data table

$("#data" + i).remove();

//it uses GUI tools for UX

this.gui.showToast("Item removed");

});

}

/\*\*

\* refresh method show loading component and call getItems method from service cliente module.

\* When the observable (return) gets data or is completed, the input is injected in the data observable to be

\* captured and managed. Then, it hides loading component.

\*/

refresh():void {

this.gui.showLoading();

this.myService.getItems().subscribe(d => {

this.data.next(d); //send data received from server to data observable

this.gui.hideLoading();

});

}

/\*\*

\*

\* @param d data received from data observable from server: array of items

\*/

fillTable(d:item[]):void{

//it empties the data table: remove all the children (html elements)

$("#data tbody").empty();

//if theres is something to be shown

if (d.length > 0) {

d.forEach(row => {

let tr =`<tr id="data${row.id}">

<th scope="row">${row.id}</th>

<td>${row.title}</td>

<td>${row.description}</td>

<td>

<button type="button" class="btn btn-info btn-edit" data-id="${row.id}"

data-title="${row.title}" data-description="${row.description}">E</button>

<button type="button" class="btn btn-danger btn-remove" data-id="${row.id}"

data-title="${row.title}">D</button>

</td>

</tr>`;

$("#data tbody").append(tr);

});

//there shouldn't be any button with event listener, because we just emptied the data table,

//but just in any case, it removes all event listeners from all buttons.

$(".btn-remove").unbind("click");

$(".btn-edit").unbind("click");

//it adds event (click) listener to add and edit buttons

$(".btn-remove").click(x => {

this.removeData($(x.target).data("id"), $(x.target).data("title"));

});

$(".btn-edit").click(x => {

this.editData(

$(x.target).data("id"),

$(x.target).data("title"),

$(x.target).data("description")

);

});

}

}

}

//This is the most important action: we create our controller instance

let indexpage = new IndexPage();

//this code is only for demostration purposes

indexpage.gui.showToast("Welcome to our experimental framework");

indexpage.gui.showToast("Only for educational purposes");

setTimeout(()=>{

indexpage.gui.showToast("It could be much better :)");

},2000)

//this code is only for demostration purposes

$('#addModal').on('hidden.bs.modal', ()=> {

indexpage.title=""; //doublewaybinding beauty. Some code to show how it works. See the footer

})

//IMPORTANT: we export our controller in order to make it available form GUI. See webpack.config.json. Sew how index.html uses it.

export = {

indexpage

};

index.html (ya visto anteriormente)

index.js (generado al paquetizar con npm start o nix webpack)

package.json (generado automaticamente con el código visto anteriormente)

tsconfig.json (ya visto anteriormente)

webpack.config.json (ya visto anteriormente)