UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

DEVELOPMENT OF AN ANGULAR LIBRARY FOR DYNAMIC LOADING OF WEB COMPONENTS

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Jaume Armengol Barahona

In partial fulfilment

of the requirements for the degree in

TELECOMMUNICATION TECHNOLOGIES AND SERVICES ENGINEERING

Advisor: Jose Luis Muñoz Tapia

Barcelona, February 2020

Abstract

This project is based on the development of a library capable of loading components with dynamic content. The library allows users to manage web components easy and comfortable way.

It originated as the study of technologies for web design and development. In this case, the Angular framework has been used, which mainly employs programming languages like Typescript and HTML.

The ultimate goal is that it contains a wide range of different components so that the user, simply by using a JSON file and without the necessary programming knowledge, is able to create a functional web application to his or her liking.

Resum

Aquest projecte es basa en el desenvolupament d'una llibreria capaç de carregar components amb contingut dinàmic. La llibreria permet als usuaris gestionar components web de manera ràpida i còmode.

Es va originar com l'estudi de tecnologies per el disseny i desenvolupament web. En aquest cas s'ha utilitzat el framework Angular, que empra principalment llenguatges de programació com Typescript i HTML.

L'objectiu final es que contingui un gran llista de components diferents per a que l'usuari, simplement utilitzant un arxiu JSON i sense coneixements de programació necessaris, sigui capaç de crear una aplicació web funcional al seu gust.

Resumen

Este proyecto se basa en el desarrollo de una librería capaz de cargar componentes con contenido dinámico. La librería permite a los usuarios gestionar componentes web de forma rápida y cómoda.

Se originó como el estudio de tecnologías para el diseño y desarrollo web. En este caso se ha utilizado el framework Angular, que emplea principalmente lenguajes de programación como Typescript y HTML.

El objetivo final es que contenga un amplio rango de componentes diferentes para que el usuario, simplemente usando un archivo JSON y sin conocimientos de programación necesarios, sea capaz de crear una aplicación web funcional a su gusto.

Acknowledgements

First of all, I would like to thank my family for all the support I have received over the years as the career comes to an end.

I would also like to thank my group of university friends for always being there, always willing to help me in whatever way is necessary, countless times. Thank you.

Finally, I would like to thank Alejandro Perez Ujaque and his everis team for giving me the opportunity to practice with them and make me feel part of their team. And to Jose Luis Muñoz, my supervisor, for helping me in whatever is necessary for the development of TFG and helping me to get this project right.

Revision history and approval record

Revision	Date	Purpose
0	20/12/2019	Document creation
1	22/01/2020	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Jaume Armengol Barahona	jaume1997_2@hotmail.com
Jose Luis Muñoz Tapia	jose.luis.munoz@upc.edu

Written by:		Reviewed and approved by:	
Date	20/12/2019	Date	22/01/2020
Name	Jaume Armengol Barahona	Name	Jose Luis Muñoz Tapia
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables:

1. Introduction

In this project I make a study and a practical development of different functionalities with a frontend framework called angular.

In particular, the functionalities consist in using dynamic rendering of components, creating components in a lazy loaded module and creating a web component with angular elements.

Finally, I apply these functionalities to a web application.

1.1. Objectives

The project main goals are:

- Rendering of Angular components with logic, following a JSON file that indicates the type of component and its configuration parameters
- Use of lazy loaded modules for component loading
- Creation of WebComponents with different functionalities (form with validations, sample of images ...)
- Creation of the library with these functionalities that we later integrate in our spa

A final goal could be to have a SPA that searches for dynamic content (news, events, images, forms…) through a request and be able to render the components that show that information dynamically.

1.2. Requirements and specifications

Project requirements:

- Use and exploit of all the new advantages provided by the latest angular version
- Being able to understand how the web page requested should be and structure it
- Get the configuration parameters from a JSON file
- Create a library that can be used on any computer through its installation

Project specifications:

- Different number of types of web pages that can be configured
- The library must be able to be installed on any computer and download the requested components

1.3. Project background

This is an innovation project that starts from the scratch. What was wanted from the beginning was to investigate the new functionalities and features of Angular 8. See how the updates help to improve the render of dynamic components, the use of lazy loaded modules…

The main project initial ideas are provided by the company supervisor. Although as it is done, the author can contribute new ideas, possible changes and modifications that he considers.

1.4. Utility

The main utility of this library is the creation of a web page quickly and that does not need programming knowledge. The deployment of a web app requires long processes of ideation, development, tests, etc... With this library it is achieved that through the use of the components that are already implemented these processes disappear, and you only have to choose which ones are going to be used and their configuration. Another advantage is the ease to learn how to use it, you just need to know what fields are available for each component and choose which ones we want the web app.

These are the main pros and cons of the library:

Pros	Cons
• Automation of the process of creating a web page • Saving of development, ideation and testing processes • Easy to add new components • Security. Only the input parameters are set, the code is not changed • Easy to correct errors and does not affect the other components	• There is no total freedom to create the desired website • The configuration fields are bounded to those that the library is able to interpret • Must conform with the design created

Table 1.1 - Pros and cons

1.5. Who is it for

The people to whom this library is addressed are those who do not have programming knowledge that is looking for a quick and easy way to create a personalized web page. A professional case could be the example of a worker in the marketing sector who intends to sell an idea, or create a simple website and thus saves contacting the technology department that would make the development, explain the idea you have and take conducted, follow-up meetings and tests, etc...

1.6. Time plan

The time plan followed during this project development is shown in Figure 1.1 Gantt Diagram. The developed work has been split in 4 work packages. A detailed description of each work package and the internal tasks developed can be found on Appendix 1.

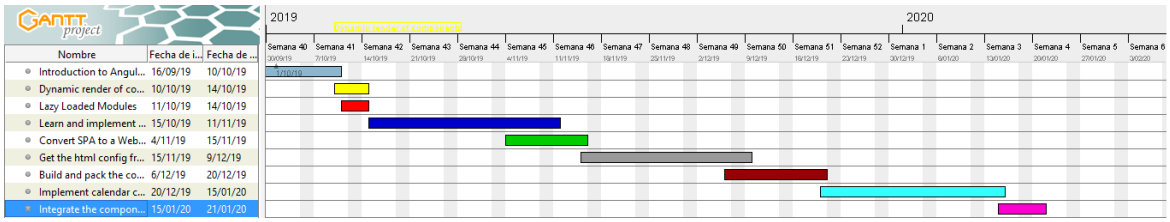


Figure 1.1 - Gantt Diagram

Introduction to Angular 8	16/09/2019 → 10/10/2019
Dynamic render of components	10/10/2019 → 14/10/2019
Lazy Loaded Modules	11/10/2019 → 14/10/2019
Learn and implement Reactive Forms	15/10/2019 → 11/11/2019
Convert SPA to a Web Component	04/11/2019 → 15/11/2019
Get the html config from json file	15/11/2019 → 09/12/2019
Build and pack the components to create a library	06/12/2019 → 20/12/2019
Implement calendar component	20/12/2019 → 15/01/2019
Integrate the component to the library	15/01/2019 → 21/01/2020

1.7. Modifications from initial plan

- The creation of dynamic components in Angular has been ruled out
- Rejected the creation of a directive that distinguishes the showed components
- Add validations to the form parameters
- Not using the changeDetectionOnPush strategy
- Add a new component (Scheduler)

2. State of the art of the technology used or applied in this thesis:

For the development of the project, I used the Angular framework. It is a new and widely used tool in the workplace in the front-end department. For the part of the data input at the external level I have used the JSON format, mainly because it is independent of any programming language, the services that share information by this method, do not need to speak the same language, that is, the sender can be Java and the receiver PHP. Each language has its own library to encode and decode JSON strings.

2.1. Angular framework

Angular is a free and Open Source JavaScript framework, created by Google and designed to facilitate the creation of modern web applications of the SPA type (Single Page Application).

The Angular programming is done using TypeScript, a language that is a superset of JavaScript that adds static typing capabilities. This gives us the advantage of being able to type things like variables, functions, returns, in addition to being able to create Interfaces. TypeScript also gives us the ability to use enumerators, modules, namespaces, decorators and generics. And last but not least is this import system, which we will use daily to atomize and modularize all our code.

The continuation of the more detailed explanation of the Angular technology is in Appendix 2.

2.2. JSON format

JSON (JavaScript Object Notation) is a lightweight data exchange format, which is easy to read and write for programmers and simple to interpret and generate for machines. In fact, it is a standard based on plain text for the exchange of information, so it is used in many systems that require displaying or sending information to be interpreted by other systems. The advantage of JSON, being a format independent of any programming language, is that the services that share information by this method do not need to speak the same language, that is, the sender can be Java and the PHP receiver. Each language has its own library to encode and decode JSON strings. The rest of the explanation is found in Appendix 2

2.3. Visual Studio Code

In the development of this project, the Visual Studio Code text editor has been used to define the files of necessary configuration and to develop the software. For more information see Appendix 2.

2.4. Bitbucket

Bitbucket is a web-based hosting service, for projects that use the Mercurial and Git version control system. It is a free service with an unlimited number of private repositories that are not displayed on profile pages - if a user has only private deposits, the website will give the message "This user has no repositories". The service is written in Python. It is similar to GitHub, which uses Git.

The rest of the explanation is found in Appendix 2.

3. Methodology / project development:

As I said in the introduction, the project is based on the creation of a library capable of rendering components dynamically through the information provided by a json file. The components implemented in this library are:

- Webform
- Random image displayer
- Scheduler
- And one by default in case of introducing a non-implemented component type or an empty file

3.1. Setting up the Local Environment and Workspace

Before starting to develop the project we must take into account all the necessary prerequisites and install the angular environment.

Prerequisites

Angular requires Node.js version 10.9.0 or later

Angular, the Angular CLI, and Angular apps depend on features and functionality provided by libraries that are available as npm packages. To download and install npm packages, you must have an npm package manager.

Once we meet the requirements we must install the CLI:

```
npm install -g @angular/cli
```

Now we have to create the project:

```
ng new init-app
```

The ng new command prompts you for information about features to include in the initial app. The Angular CLI installs the necessary Angular npm packages and other dependencies. The CLI creates a new workspace and a simple Welcome app, ready to run.

To create the components you must use the command:

```
ng generate component component-name
```

To create the library you must use the command:

```
ng generate library app-lib
```

Once we know this, we create our folder of projects where we will create the library.

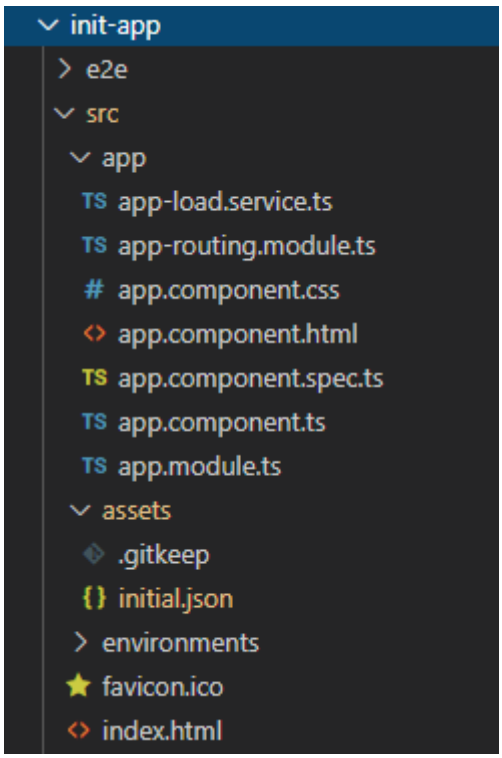


Figure 3.1 - App folder distribution

Inside the library, we will add the components that we want to implement.

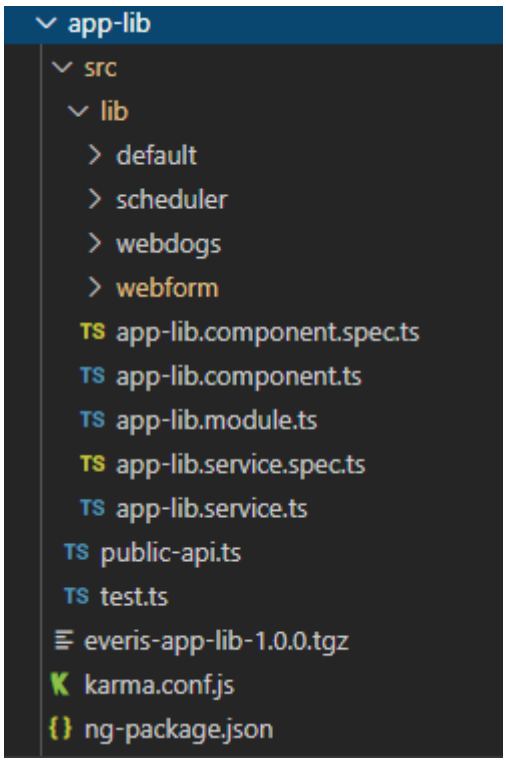


Figure 3.2 - Lib folder distribution

To be able to install the library from another project and to be able to use its components, you have to export them in the *public-api.ts* file.

```
/*
 * Public API Surface of app-lib
 */

export * from './lib/app-lib.service';
export * from './lib/app-lib.component';
export * from './lib/app-lib.module';
export * from './lib/webform/webform.component';
export * from './lib/webdogs/webdogs.component';
export * from './lib/scheduler/scheduler.component';
export * from './lib/default/default.component';

```

Figure 3.3 - public-api.ts file

3.2. App Initializer (local storage)

To obtain the json file with the configuration of our web app we use the Local Storage tool. LocalStorage allows you to store data in the web browser. And that these persist and are available while browsing the web application, until this information is deleted from the browser. To save data in localStorage we will use the following instruction:

```
localStorage.setItem('tutorial', How to use localStorage in Angular');
```

To recover the data stored in the localStorage we will use the following instruction to which we will pass the key of the data we want to recover:

```
this.data = localStorage.getItem('tutorial');
```

In the case of our project, we will set the json file that we will have locally, but the file could be obtained even if it comes from another site, provided it is in the localstorage. When setting, it is convenient to do before the application starts. Since we want to set the json file before the application is initialized, we will use App Initializer. Angular has a hook in its process of initialization called App Initializer. An App Initializer is a special kind of Injection Token— identifier of a dependency— of type Array of Functions that is executed when an application is initialized.

In order for our library to set the json file in the local storage before initializing the app you have to modify the *app-module.ts* file:

```
export function initializeApp1(appLoadService: AppLoadService) {
  return (): Promise<any> => {
    return appLoadService.Init();
  }
}
```

Figure 3.4 - App Initializer function call

```
providers: [
  AppLoadService,
  { provide: APP_INITIALIZER, useFactory: initializeApp1, deps: [AppLoadService], multi: true }
],
```

Figure 3.5 - Providers of app.module.ts

We import the App Initializer module, and call the function that our service will perform in the *app-load.service.ts* file:

```
@Injectable()

export class AppLoadService {

    initial: any[];

    constructor(private http: HttpClient) { }

    Init() {
        return new Promise<void>((resolve, reject) => {
            console.log("AppLoadService.init() called");
            // do your initialization stuff here
            localStorage.clear();
            this.http.get('./assets/initial.json').subscribe(data => {
                this.initial = data as any[];
                localStorage.setItem("initial", JSON.stringify(data));
            });


            setTimeout(() => {
                console.log("AppLoadService finished");
                resolve();
            }, 500);
        });
    }
}
```

Figure 3.6 - App Initializar function

As we can see in the figure, we obtain our local json file through an http client and then set it in the localstorage with the name "initial". If in a timeout of 500 ms you could not perform the action, an error in the app is skipped.

Now that we have the JSON file with the configuration of our web app in the local storage, we must get it in the main activity of our project. Consequently, we can call the library by inputting the configuration so that it can render the necessary components. To do this, in the *app.component.ts* file:

```
10  export class AppComponent implements OnInit {
11    title = 'angularApp';
12    ini: any[];
13
14    constructor(private compFactResolver: ComponentFactoryResolver) {
15      this.ini = JSON.parse(localStorage.getItem("initial"));
16    }
```

Figure 3.7 - Get json file from localstorage

And in the *app.component.html* file:

```
1   <lib-app-lib [initial]="ini"></lib-app-lib>
```

Figure 3.8 - Open library with json input info

This is an example of what we would be saving in the local storage when we set the json file.

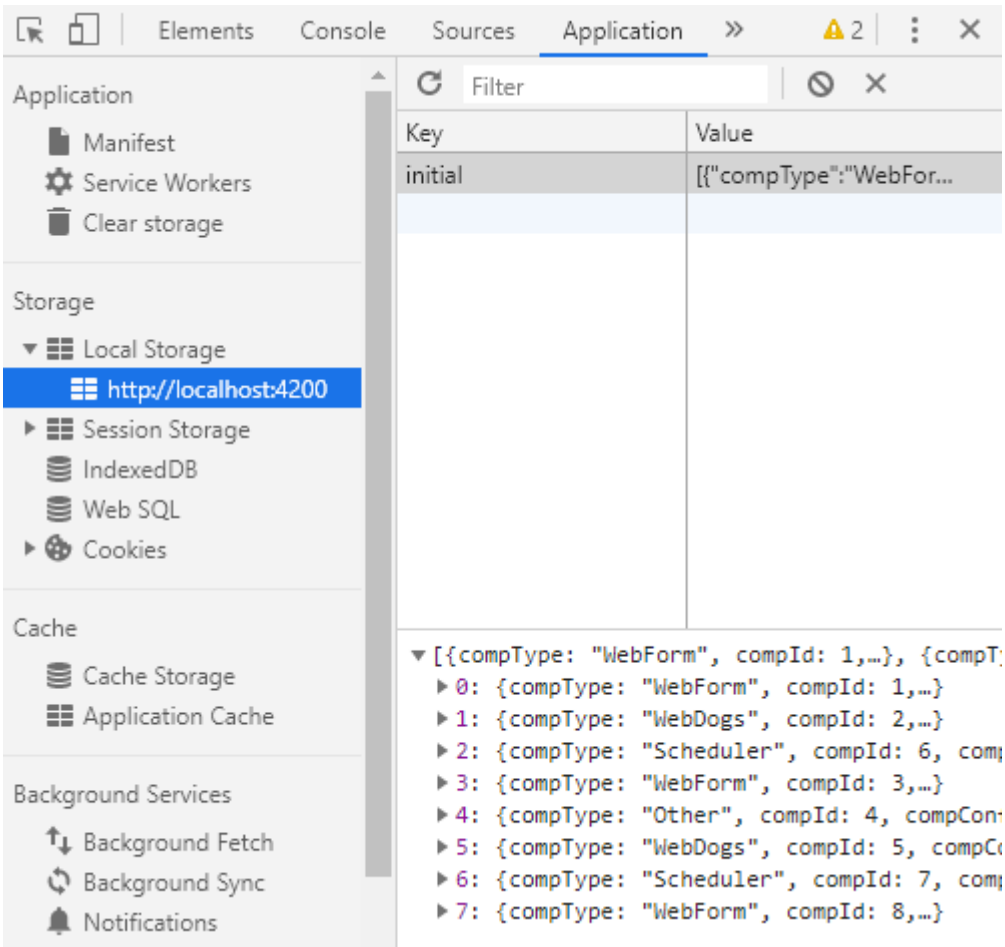


Figure 3.9 - Localstorage information in browser

3.3. Configuration JSON format

To fill the json file with the configuration of the application, you have to follow some rules according to the established sketch. It is necessary to indicate:

- Component ID
- Component Type that we want to render
- Component configuration

This is a simple example of the sketch:

```
{
        "compType": "NameType”,
        "compId": 1,
        "compConfig": [
            {}
        ]
    }
```

3.4. Angular Web Elements

The components that we have implemented in our library are called web components. Web components are a series of web APIs that allow us to create customizable, encapsulated and reusable HTML tags. They can be used on any page and web application without the need to import external JavaScript libraries. In order to use them, our browser must support a series of Web APIs:

- Custom Elements: Allows the creation of new types of elements in the DOM
- Shadow DOM: Define how styles and language are encapsulated
- HTML Imports: Defines the inclusion of HTML in other HTML documents
- HTML Template: Defines how HTML fragments are declared when the page loads, but which can be used later during execution

Angular elements are Angular components packaged as custom elements (also called Web Components), a web standard for defining new HTML elements in an independent way. Custom elements are a Web Platform feature currently supported by Chrome, Firefox, Opera, and Safari, and available in other browsers through polyfills. The @angular/elements package exports a *createCustomElement()* API that provides a bridge from Angular's component interface and change detection functionality to the built-in DOM API.

Custom elements are automatically started when they are added to the DOM and automatically destroyed when they are removed from the DOM. Once a custom element is added to the DOM for any page, it looks and behaves like any other HTML element, no special knowledge of Angular terms is necessary.

- **Easy dynamic content in an angular application.** Transforming a component into a custom element provides an easy path to create dynamic HTML content in your Angular application. HTML content that you add directly to the DOM in an Angular application is usually displayed without Angular processing, unless you define a dynamic component, add your own code to connect the HTML tag to your application data and participate in the detection of changes. With a custom element, all that wiring is done automatically.
- **Content rich applications.** The customized elements allow its suppliers to use sophisticated Angular functionalities without requiring knowledge of it. All you need to tell your content provider is the syntax of your custom item. They don't need to know anything about Angular, or anything about data structures or the implementation of their components.

Angular provides the *createCustomElement()* function to convert an angular component, along with its dependencies, into a custom element. The function collects the observable properties of the component, along with the angular functionality that the browser needs to create and destroy instances, and to detect and respond to changes.

The *customElements.define()* function is used to register the configured constructor and its custom element label associated with the browser. When the browser finds the tag for the registered item, it uses the constructor to create a custom item instance.

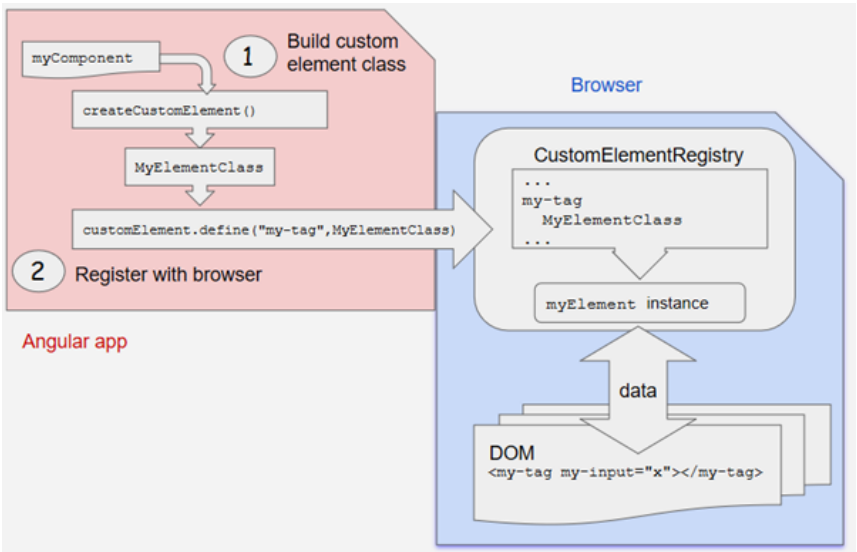


Figure 3.10 - Create Custom Element

The browsers that support the use of web components are:

BROWSER	CUSTOM ELEMENT SUPPORT
CHROME	Supported natively
SAFARI	Supported natively
FIREFOX	Supported natively as of version 63
OPERA	Supported natively
MICROSOFT EDGE	In process

Table 3.1 - Browsers supporting Angular Elements

In order to use the web components in our project, it is necessary to add and import the dependency of angular elements, with the following command:

`ng add @angular/elements`

Once we have added the dependency, we must create the web components, using the *createCustomElement()* method, in the *app.lib.module.ts* file:

```
import { AppLibComponent } from './app-lib.component';
import { WebformComponent } from './webform/webform.component';
import { WebdogsComponent } from './webdogs/webdogs.component';
import { DefaultComponent } from './default/default.component';
import { SchedulerComponent } from './scheduler/scheduler.component';


@NgModule({
  declarations: [
    AppLibComponent,
    WebformComponent,
    WebdogsComponent,
    DefaultComponent,
    SchedulerComponent,
  ],
  exports: [
    AppLibComponent,
    WebformComponent,
    WebdogsComponent,
    DefaultComponent,
    SchedulerComponent
  ],
  schemas: [
    CUSTOM_ELEMENTS_SCHEMA
  ],
  providers: [ AgendaService, DayService, WeekService, WorkWeekService, MonthService ],
  entryComponents: [
    WebformComponent,
    WebdogsComponent,
    DefaultComponent,
    SchedulerComponent
  ]
})
```

Figure 3.11 - Configuration of app-lib.module.ts

As we can see in the figure, we must import the components that we want to create as web components, declare them and add them to the array of entry components, since this component is not part of a parent component and is a component that must be part of the bootstrapping of the application.

```
export class AppLibModule {

  constructor(private injector: Injector) {

    customElements.define('ng-webform', createCustomElement(WebformComponent, { injector }));
    customElements.define('ng-webdogs', createCustomElement(WebdogsComponent, { injector }));
    customElements.define('ng-scheduler', createCustomElement(SchedulerComponent, { injector }));
    customElements.define('ng-default', createCustomElement(DefaultComponent, { injector }));

  }

  ngDoBootstrap() { }
```

Figure 3.12 - Definition of Custom Elements

In this part we see how we use the function *customElements().define*, which creates a web component with the HTML tag with which later, when used, it will be referenced.

3.5. Library Components

In the *app-lib.component.ts* file we have as input the configuration of the web app, which we have called *initial*, which will tell us which components we have implemented we want to use, and the respective fields of each one. In case the *initial* file is empty, our application will render the component by default. Otherwise, we will go through the entire file to see how many components we should load and what type they are, according to their *compType* parameter in the json file. For each type of component, we will pass as input the configuration to be able to render them with the necessary information. If the *compType* does not correspond to any of the components implemented in the library, the default component will be loaded.

```
@Component({
  selector: 'lib-app-lib',
  template: `

    <div *ngIf="initial.length === 0">
      <ng-default></ng-default>
    </div>
    <div *ngFor="let comp of initial" [ngSwitch]="comp.compType">

      <ng-webform [form]="comp.compConfig" *ngSwitchCase="'WebForm'"></ng-webform>
      <ng-webdogs [dogsInfo]="comp.compConfig" *ngSwitchCase="'WebDogs'"></ng-webdogs>
      <ng-scheduler [eventInfo]="comp.compConfig" *ngSwitchCase="'Scheduler'"></ng-scheduler>
      <ng-default *ngSwitchDefault></ng-default>

    </div>

  `,
  styles: []
})
export class AppLibComponent implements OnInit {

  @Input() initial;
```

Figure 3.13 - Component rendering according to the compType

Next we will explain the development of each of the components of our library.

3.5.1. Webform Component

The webform component, as the name implies, is a reactive form, created from the fields indicated in the configuration parameter called *rForm*. It is created with the information passed as input to the *app-lib.component.ts* file seen previously. The fields created in the form and that can be used as we want are:

1) Email
2) Password
3) Text
4) Phone number
5) Date
6) Time
7) Week
8) Option
9) Radio button
10) Checkbox
11) Button

The **email** refers to an input field in which only one string with an email format is allowed, that is, with the @ separator and expressing the domain (.com, .es, etc.)

The **password** refers to an input field that indicates a hidden string that corresponds to the email key. The user who writes it will not be able to see it on the screen, like any password, and it must have a minimum of 8 characters to be accepted. Otherwise, an error would appear indicating the length requirement.

The **text** field is an input that we can use, whatever data we want to know about the user, be it name, surname, gender, profession... In the case of first name, a minimum of 2 characters, if not, shows an error message.

The **phone number** is an input field that collects the value of the telephone number of the user who completes the form. Like any phone number, it must have 9 values and the first one must be 6, to be correct.

The **date** field is an entry that can refer to any day you want to ask, such as the date of birth, expiration date, date of obtaining the driving license ... If required, a minimum and a maximum date can be added. That will make the calendar we see to choose the date, is limited to specific values.

The **time** entry field lets us choose a specific time, which corresponds to what is requested in the form, whether it is the time of delivery of something, meeting time, current time, whatever.

The **week** entry field is similar to the date field but instead of choosing a specific day, an entire week of the year is selected.

The **option** corresponds to an entry field in which we can choose between several options, previously indicated, such as the case of nationality, profession, brand... We can put as many options as we want. In this case at the time of choosing, a list with all the options is broken down.

The **radio button** refers to a field similar to the option, but in this we can see all the options with a selection button next to it. We can only choose one of the buttons, each time we select one, the one that was previously selected will not be.

The **checkbox** is a field that we can select or not, it is like marking an option in case it is correct or we want to mark it as achieved or done. It is independent of the other component checkboxes. We can select or unselect it by clicking on it.

The **button** is a field that refers to a function if it is clicked. In the case of this component, it refers to the *clear()* function that deletes all other fields contained in the form.

All fields described above are required, except checkboxes. It is necessary to fill them in order to submit. If any of the required fields is empty, the form submission cannot be completed. If we click the submit button with one or more empty fields, an error message will appear saying that all the fields are required, and another one below each one that has not been filled. In addition, error messages corresponding to the validations of each field may also appear, such as the email format, password length, phone number...

In the *webform.component.ts* file is where we perform all this configuration. We keep in the *form* variable the information that corresponds to the fields that we are going to show in our form. To do this, the reactive form and a series of flags must be created so that the component does not appear on the screen before it is created correctly. At the moment it detects the data input, it goes through the *form* variable and adds the fields it contains to the reactive form. All these fields that you add make them required (except the checkbox) and then depending on the type of field, add some or other validations.

```
9   export class WebformComponent implements OnInit {
10
11    rForm = new FormGroup({ });
12    submitted = false;
13    countries: [];
14    flag = false;
15
16    @Input() form;
17
18    constructor() {
19    }
20
21    ngOnInit(){
22    }
23    ngOnChanges() {
24      this.flag = true;
25      for (let i = 0; i < this.form.length; i++) {
26        this.rForm.addControl(this.form[i].name, new FormControl(this.form[i].value));
27        if (this.form[i].type !== 'checkbox') {
28          this.rForm.controls[this.form[i].name].setValidators(Validators.required);
29        }
30        if (this.form[i].type === 'email') {
31          this.rForm.controls[this.form[i].name].setValidators(Validators.email);
32        } else if (this.form[i].name === 'Firstname') {
33          this.rForm.controls[this.form[i].name].setValidators([Validators.minLength(this.form[i].minlength),
34        } else if (this.form[i].type === 'password') {
35          this.rForm.controls[this.form[i].name].setValidators([Validators.minLength(this.form[i].minlength)]
36        } else if (this.form[i].type === 'phonenumber') {
37          this.rForm.controls[this.form[i].name].setValidators([Validators.minLength(this.form[i].minlength),
38        } else if (this.form[i].name === 'Country') {
39          this.countries = this.form[i].countries.split(', ');
40        }
```

Figure 3.14 - Saving config params in a reactive form

Some of the validations that we add to the form fields are already implemented in the Angular language, such as required or maximum or minimum, but not all. In that case we can create them the way we want. An example of this is the phone number requirement. We have to create a validation so that an error message appears in case it doesn't start with 6.

```
45    phoneNumberValidator(control: AbstractControl) {
46      if (!control.value.startsWith('6') && control.value.length !== 0) {
47        return { validNumber: true };
48      }
49      return null;
50    }
```

Figure 3.15 - Custom validator creation phoneNumberValidator

In addition to the fields that we can choose for our form, there are some fixed buttons that will always appear, there are more or less fields. These are the submit button, which refers to the sending of data (although in this case it shows the entered values per console) and the update params button, which modifies the values of the variables we want (in this case, the time and name).

```
52    onSubmit() {
53      this.submitted = true;
54      if (this.rForm.status === 'VALID') {
55        console.log(this.rForm.value);
56      } else {
57        console.log('All files are required. Invalid form');
58      }
59    }
60
61    updateParams() {
62      this.rForm.patchValue({
63        Firstname: 'Sergio',
64        Time: '14:08',
65      });
66    }
```

Figure 3.16 - OnSubmit and updateParams functions

The clearButton, although optional, should normally be on all forms. Call the *clear()* function, which deletes the values of all existing fields and in the case of radio buttons, deselect the button that was selected when we clicked on another.

```
68    clear(name: string, type: string) {
69      if (name === 'ClearButton') {
70        for (let params of this.form) {
71          if (params.name !== 'ClearButton') {
72            this.rForm.controls[params.name].setValue('');
73          }
74        }
75
76      } else if (type === 'radio') {
77        for (let params of this.form) {
78          if (params.type === 'radio' && params.name !== name) {
79            this.rForm.controls[params.name].setValue('false');
80          }
81        }
82
83      }
84    }
```

Figure 3.17 - ClearButton functionality

In the *webform.component.html* file is where we perform the view of the component. Before talking about it, we must mention Angular material.

Angular Material

Angular Material is a style library (like Bootstrap) based on the design guide Material Design, made by the Angular team to integrate seamlessly with Angular. To use it, we must add the dependency to our project with the following command:

```
ng add @angular/material
```

The ng add command will install Angular Material, the Component Dev Kit (CDK), Angular Animations and ask you the following questions to determine which features to include:

1. Choose a prebuilt theme name, or "custom" for a custom theme:
   You can choose from prebuilt material design themes or set up an extensible custom theme.
2. Set up HammerJS for gesture recognition:
   HammerJS provides gesture recognition capabilities required by some components (mat-slide-toggle, mat-slider, matToolTip).
3. Set up browser animations for Angular Material:
   Importing the BrowserAnimationsModule into your application enables Angular's animation system. Declining this will disable most of Angular Material's animations.

The ng add command will additionally perform the following configurations:

- Add project dependencies to *package.json*
- Add the Roboto font to your *index.html*
- Add the Material Design icon font to your *index.html*
- Add a few global CSS styles to:
  - Remove margins from body
  - Set height: 100% on html and body
  - Set Roboto as the default application font

Now that we know Angular Material, we can see the development of the component view.

```
<div *ngIf="flag">
  <mat-card>
    <mat-card-header>
      <mat-card-title>WebForm</mat-card-title>
    </mat-card-header>
    <br>
    <mat-card-content>
      <form [formGroup]="rForm" (ngSubmit)="onSubmit()">
        <div *ngFor="let param of form">
          <div *ngIf="param.type != 'option' && param.type != 'checkbox' && param.type != 'radio' && param.
            <mat-form-field>
              <mat-label>{{param.name}}</mat-label>
              <input matInput [formControlName]="param.name" [type]="param.type" [(ngModel)]="param.value"
            </mat-form-field>
          </div>
          <div *ngIf="param.type == 'checkbox'">
            <mat-checkbox [formControlName]="param.name" [checked]="param.checked">{{param.name}}</mat-chec
          </div>
          <div *ngIf="param.type == 'radio'">
            <mat-radio-group [formControlName]="param.name">
              <mat-radio-button [value]="param.checked" (click)="clear(param.name, param.type)">{{param.nam
            </mat-radio-group>
          </div>
          <div *ngIf="param.type == 'option'">
            <mat-form-field>
              <mat-select [formControlName]="param.name" required>
                <div *ngFor="let country of countries">
                  <mat-option [value]="country" >{{country}}</mat-option>
                </div>
              </mat-select>
              <mat-label>Country</mat-label>
            </mat-form-field>
          </div>
          <div *ngIf="param.type == 'button'">
            <button mat-raised-button [type]="param.type" color="primary" (click)="clear(param.name, param.
          </div>

          <div *ngIf="submitted && rForm.get(param.name).errors">
            <div class="requiredMsg" *ngIf="rForm.get(param.name).hasError('required')">{{param.name}} is r
            <div class="requiredMsg" *ngIf="rForm.get(param.name).hasError('email')">Format is not correct
            <div class="requiredMsg" *ngIf="rForm.get(param.name).hasError('minlength')">Min of {{param.min
            <div class="requiredMsg" *ngIf="rForm.get(param.name).hasError('maxlength')">Max of {{param.max
            <div class="requiredMsg" *ngIf="rForm.get(param.name).errors.validNumber">Phone number format i
          </div>

          <br>
          </div>
        </form>
      </mat-card-content>
```

Figure 3.18 - Html file of webform component

First of all, we have a flag that does not display the page until you have added all the fields to the form, which is activated in the *webform.component.ts* file that we have seen previously. Then we create the layout where we will show the form, with its title in the header. For the body, we go through the *form* variable that contains all the fields that the user has entered, and depending on the type, we will create one content or another. All have associated the name, type and value of the field in the form, and some in addition to that have the minimum or maximum associated, whether checked or not, the function you call in the case of the button, etc. Finally, we can see in the figure the messages that will appear on the screen in case there is an error in the requirements at the time of submitting. Depending on the type of error, one message or another is displayed.

```
51      <div *ngIf="submitted && rForm.status === 'INVALID'" class="requiredMsg">All fields are required. Check
52        does not meet the specifications </div>
53
54      <br>
55
56      <p>Form Status: {{ rForm.status }}</p>
57
58      <br>
59      <button mat-stroked-button type="submit" (click)="onSubmit()">Submit</button>
60      <button mat-stroked-button (click)="updateParams()">Update Params</button>
61      <br>
62
63      <div *ngIf="submitted && rForm.valid">
64        <h2>Personal Information</h2>
65        <p>{{rForm.value | json}}</p>
66      </div>
67    </mat-card>
68
69  </div>
```

Figure 3.19 - Common part of all forms

At the bottom of the page, a message will appear telling us if the current status of the form is valid or not, depending on whether all the requirements are met. In case they are not met and the submit button is clicked, an error message will appear indicating that all the fields are necessary, in addition to the one that already appears below the fields that do not meet the requirements. There are also two buttons that will always appear on all forms, the Submit and the UpdateParams. Submit prints the form data in console, and allows the data to be viewed in json format at the bottom of the page. UpdateParams changes the name and time, if any, of the form, whether written or not.

3.5.2. Image Displayer Component

This component shows on the screen an image that comes from an API and a message that we choose. It is created with the information passed as input to the *app-lib.component.ts* file seen previously. The input fields that we can configure are:

1) API
2) Message
3) Image Height
4) Image Width
5) Font API

The **API** is the web address from which we will receive a random image, which will then be the one displayed on the page. In the API there is a huge image bank and every time we make a request it will return a different image.

In the **message** we can write what we want. It is a simple phrase that appears on the image.

We can also set the image size. To do this you must indicate the **height** and **width** of the image in pixels.

In the **font API** field we can indicate the typeface we want for our message.

In the *webdogs.component.ts* file is where we perform all this configuration. We receive the configuration information through an input and save it in the *dogsInfo* variable. At the moment it receives the data, it saves them in their respective variables and activates the flag, so that it shows the image once it has the necessary data.

```
export class WebdogsComponent implements OnInit {

  @Input() dogsInfo;
  api;
  message;
  imgheight;
  imgwidth;
  fontapi;
  img;
  flag = false;

  constructor(private http: HttpClient) { }

  ngOnInit() {
  }

  ngOnChanges() {
    this.api = this.dogsInfo[0].api;
    this.message = this.dogsInfo[0].message;
    this.imgheight = this.dogsInfo[0].height;
    this.imgwidth = this.dogsInfo[0].width;
    this.fontapi = this.dogsInfo[0].fontapi;
    this.getImage();
    this.flag = true;
  }
```

Figure 3.20 - Saving config params in local variables

To obtain the image, we use the *getImage()* function, which uses a variable of type *httpClient* to obtain the API response URL, with the random image address.

```
  getImage() {
    fetch(this.api).then((response) => {
      this.img = response.url;
      console.log(this.img);
    })
  }
```

Figure 3.21 - Getting the image URL from the API

The development of the view is very simple. First we must refer to the *font api* to be able to use the typeface that we have entered in the configuration. Then we will use the flag to only show the component when all the necessary fields have been saved to display the image. We will also add a refresh button, so that each time it is clicked it changes the image that is displayed, although they all come from the same API. Finally, we show the image with the size specified in the configuration.

```
<link href="https://fonts.googleapis.com/css?family=Nanum+Pen+Script" rel="stylesheet">
<div class="component_container" *ngIf="flag">
  <h1>{{message}}</h1>
  <button (click)="getImage()">Refresh</button>
  <br><br>
  <img src="{{img}}" *ngIf="img" [width]="imgwidth" [height]="imgheight"/>
</div>
```

Figure 3.22 - Html file of image displayer component

3.5.3. Scheduler Component

This component shows a calendar in which appear the different tasks and events that we have entered in the json configuration file. It is created with the information passed as input to the *app-lib.component.ts* file seen previously. The input fields that we can configure are:

1) Id
2) Subject
3) Start time
4) End time

The **Id** is simply an identifier to refer to a task or event.

The **subject** field is a brief description of the event or task that we want to appear on the calendar.

StartTime and **endTime** are the fields that correspond to the start and end of the event. They are written in the following format:

StartTime/EndTime: “Year, Month, Day, Hour, Minute”

In the *scheduler.component.ts* file is where we perform all this configuration. We receive the configuration information through an input and save it in the *eventInfo* variable. We create the *eventShow* variable, which is what we will pass as input to the view to show the calendar, and the flag that will allow you to see the page when all the events have been collected.

To save all the tasks in a list, we go through the *eventInfo* variable and store the events in an auxiliary variable. In this variable they are saved with the necessary format to be able to show them, with their Id, their subject, their start time and their end time. We will add this auxiliary variable to *eventShow*, so that the scheduler can understand the data.

```
export class SchedulerComponent implements OnInit {

  @Input() eventInfo;
  flag = false;

  constructor() { }

  public eventShow: EventSettingsModel = {
    dataSource: []
  };
  public dataAux: Object[] = [];

  ngOnInit() {
  }

  ngOnChanges() {
    for(let i = 0; i < this.eventInfo.length; i++) {
      this.dataAux.push({
        Id: this.eventInfo[i].Id,
        Subject: this.eventInfo[i].Subject,
        StartTime: new Date(this.eventInfo[i].StartTime.split(', ')[0], this.eventInfo[i].StartTime.split('
        EndTime: new Date(this.eventInfo[i].EndTime.split(', ')[0], this.eventInfo[i].EndTime.split(', ')[1
    });
    }

    this.eventShow.dataSource = this.dataAux;
    this.flag = true;
  }

}
```

Figure 3.23 - Saving config params in an aux variable

For the development of the component view we have used a Syncfusion tool. The scheduler used in this project is created from the `ej2-angular-schedule` package. To use it, you have to install the package dependency with this command:

`npm install @syncfusion/ej2-angular-schedule –save`

Once installed, the *scheduleModule* must be imported into *app-lib.module.ts* together with the following providers: *AgendaService, DayService, WeekService, WorkWeekService, MonthService* (can be seen in figure 3.11).

You also have to add the CSS reference link inside the index.html file:

```
<link href="https://cdn.syncfusion.com/ej2/material.css" rel="stylesheet" />
```

And now that we have followed this process, we can define the code in the *scheduler.component.html* file that will show the calendar with all the tasks and events that we pass as input.

```
<div *ngIf="flag">
    <ejs-schedule [eventSettings]="eventShow"></ejs-schedule>
</div>
```

Figure 3.24 - Passing the info as input to create the scheduler

3.5.4. Default Component

The default component is a simple page that displays the error message:

“You have not entered any component or your component type is not registered”

This component has virtually no configuration, it only shows that message. It appears in two possible cases:

- The *initial* file that comes from the localstorage with the web page configuration is empty, does not contain any components
- The type of component indicated in the *initial* file is not implemented in the library

3.6. Pack the library

To make our library possible to install and use in another project we must package it. To do this you have to build the project and use the following command:

`npm pack` on dist/app-lib

The dist folder is created after build the project. Now we can give the library a name, which we will use to import all its components from other projects where we want to use it.

```
npm init -scope=@libname
```

This command will create a .tgz file with the name of the library, which will be the one we will use from another project to install it and thus be able to import all the modules.

4. Results

The results of this TFG project are detailed below. The final result obtained is a library that dynamically renders and loads the components that it has implemented, the webform, the image displayer and the scheduler.

4.1. Webform Component

In this section we will see a real example of how this component works. How it reacts to different input configurations and to the validations of the form fields.

A case in which the *initial.json* file has all the fields that we can configure in the form and all filled in would be:

```
{
    "compType": "WebForm",
    "compId": 1,
    "compConfig": [
        {
            "type": "email",
            "name": "Email",
            "value": "jaume@mail.com",
            "placeholder": "Your email here"
        },
        {
            "type": "password",
            "name": "Password",
            "value": "admin123",
            "placeholder": "Your password here",
            "minlength": "8"
        },
        {
            "type": "firstname",
            "name": "Firstname",
            "value": "Jaume",
            "placeholder": "Your Firstname here",
            "maxlength": "15",
            "minlength": "2"
        },
        {
            "type": "lastname",
            "name": "Lastname",
            "value": "Armengol",
            "placeholder": "Your Lastname here"
        },
```

```
        {
            "type": "phonenumber",
            "name": "PhoneNumber",
            "value": "678901234",
            "placeholder": "Your phone number here",
            "maxlength": "9",
            "minlength": "9"
        },
        {
            "type": "date",
            "name": "Birthdate",
            "value": "1997-07-07",
            "placeholder": "yyyy/dd/mm",
            "min": "1979-12-31"
        },
        {
            "type": "time",
            "name": "Time",
            "value": "16:20",
            "placeholder": "00:00"
        },
        {
            "type": "week",
            "name": "Week",
            "value": "2013-W13",
            "placeholder": "Week here"
        },
```

```
    {
        "type": "option",
        "name": "Country",
        "value": "Spain",
        "placeholder": "Your country here",
        "countries": ", France, Germany, Italy, Spain, USA"
    },
    {
        "type": "radio",
        "name": "Accept",
        "value": "true",
        "placeholder": "",
        "checked": "true"
    },
```

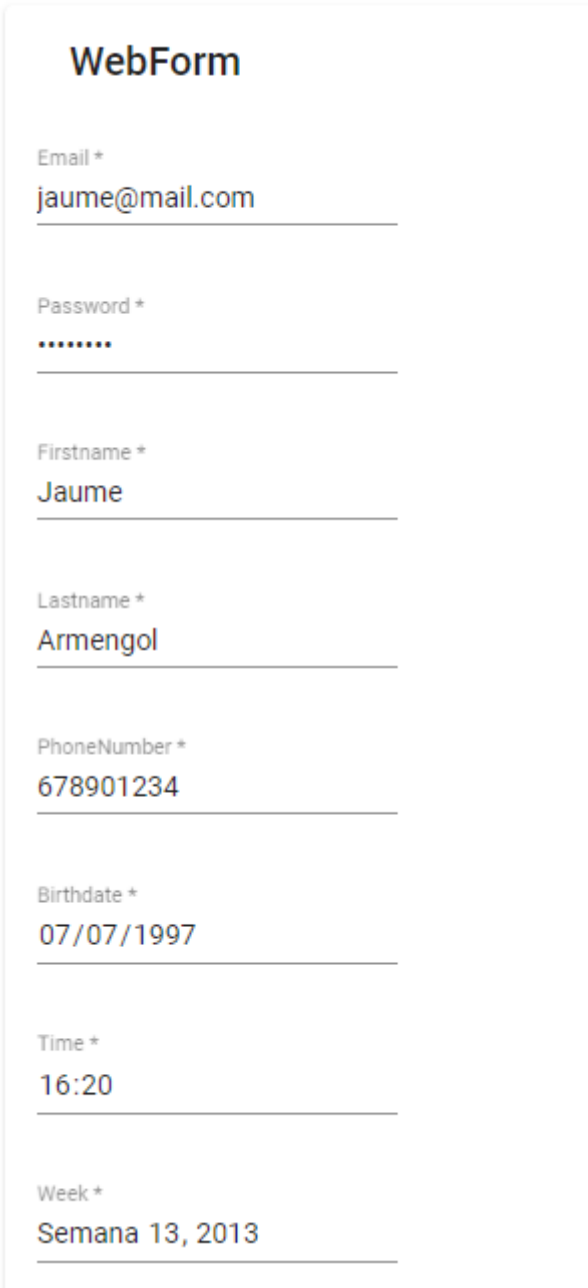
```
        {
            "type": "radio",
            "name": "Deny",
            "value": "false",
            "placeholder": "",
            "checked": "true"

        },
        {
            "type": "radio",
            "name": "NS/NC",
            "value": "false",
            "placeholder": "",
            "checked": "true"

        },
        {
            "type": "checkbox",
            "name": "Done",
            "value": "true",
            "placeholder": "",
            "checked": ""
        },
        {
            "type": "checkbox",
            "name": "Want",
            "value": "",
            "placeholder": "",
            "checked": ""
        },
        {
            "type": "button",
            "name": "ClearButton",
            "value": "x",
            "placeholder": "GO"
        }
    ]
},
```


Figure 4.1 - Webform json file example (all the fields)

And with this configuration, the result of the website would be:



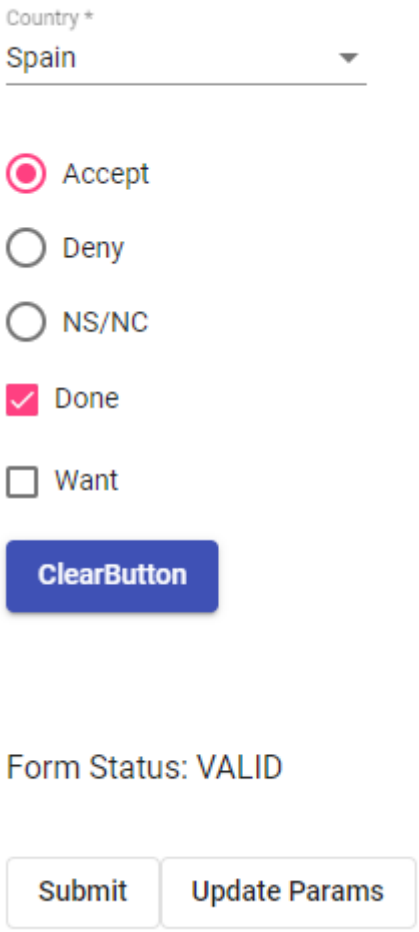


Figure 4.2 - Webform view in browser

As we can see, all the fields that we have entered in the component configuration appear in the json file. All appear with their default value. We can also see that the password field is hidden, and the selection buttons are clicked or not according to what we have specified. In the case of radio buttons, if we want one to be clicked, we must set the value parameter to true, if not false. We must remember that of the existing radio buttons there can only be one selected. In the case of checkboxes, if we want them to be selected, we must set the value of checked to true, but if we do not want it, we must leave it blank. The parameters Week and Birthdate, being of type date, we can modify them using a small calendar that appears. The Country field is of type options, and its value can be changed using the list of countries that appear when we click on the parameter.

If we click the submit button, as in this case the form format is valid, the user data will appear in json format at the bottom of the page.

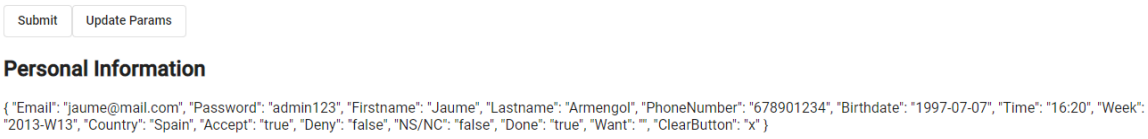


Figure 4.3 - User data view

If the format is not valid, clicking on the submit button will not display the data, but an error message will appear.

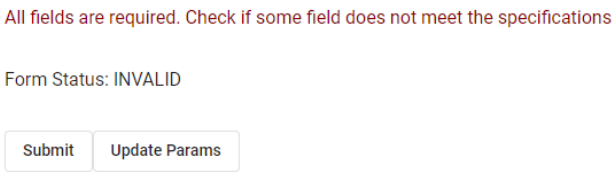
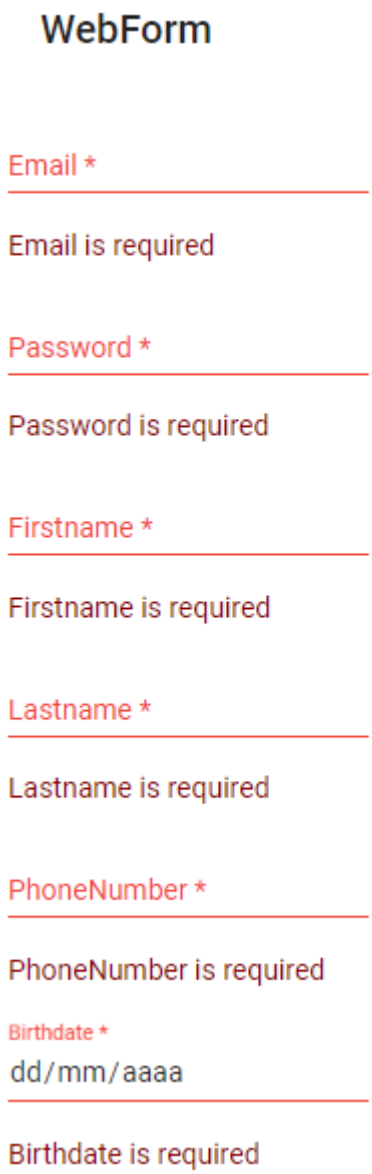


Figure 4.4 - Error message for invalid form

If we now use the "ClearButton" button, we will erase the value of all the fields in the form and the different error messages of each field will appear.



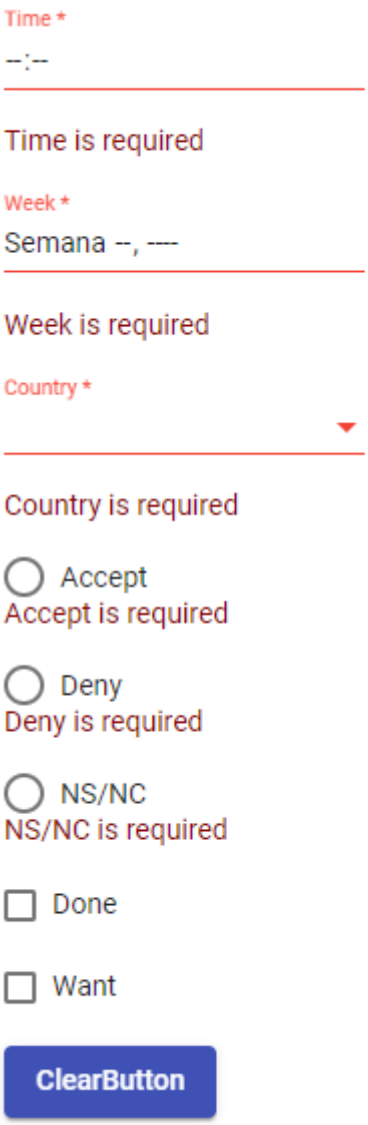


Figure 4.5 - Error messages from empty fields in the form

As the fields are filled in, the error messages will disappear. Having deleted all the values, the error messages that appear correspond to the requirement of required, which indicates whether the field is mandatory or not. Now we are going to see the errors in the fields that have some more requirements.

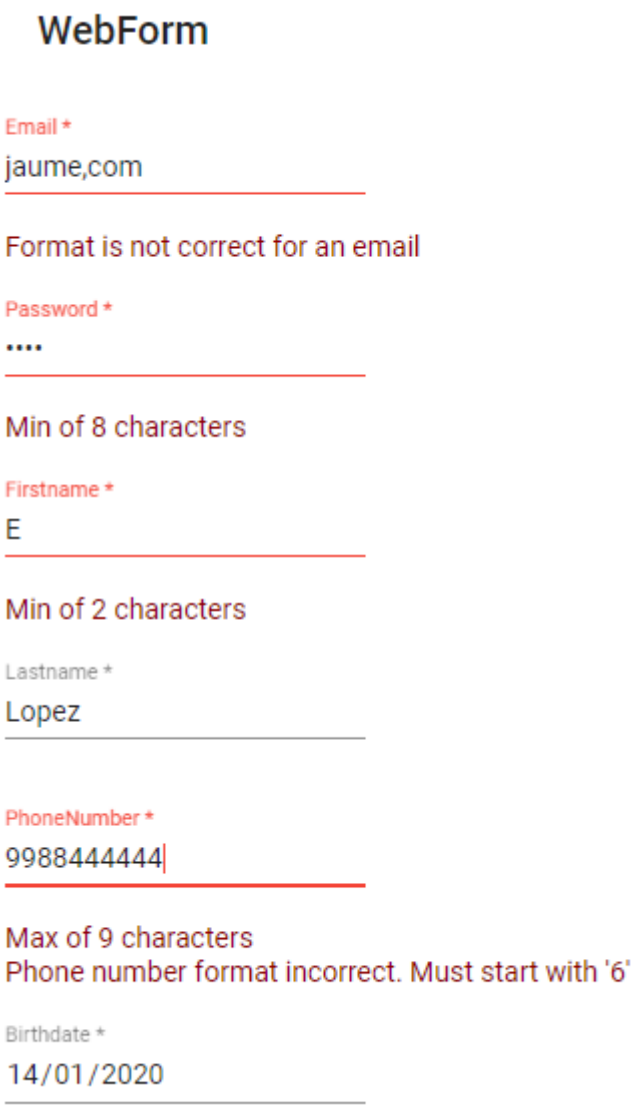


Figure 4.6 - Error messages from fields with custom validations

The form is still invalid, therefore we cannot submit, even if the fields are not empty. This is due to the requirements of some fields that are not met. We can see the case of the firstname and the password that asks us for a minimum number of characters; the input format of the email field, which has to be of the style: example@dom.com; the phone number must have 9 digits and start with 6; All errors are explained in the message.

Once these errors are corrected, the form format will be valid again and we can see the user data in json format at the bottom of the page.

It is not necessary to use all possible fields of the component. A simpler example with fewer configured webform fields and without setting a default value would be:

```
{
    "compType": "WebForm",
    "compId": 8,
    "compConfig": [
        {
            "type": "email",
            "name": "Email",
            "value": "",
            "placeholder": "Your email here"
        },
        {
            "type": "password",
            "name": "Password",
            "value": "",
            "placeholder": "Your password here",
            "minlength": "8"
        },
        {
            "type": "button",
            "name": "ClearButton",
            "value": "x",
            "placeholder": "GO"
        },
        {
            "type": "date",
            "name": "Birthdate",
            "value": "",
            "placeholder": "yyyy-dd-mm",
            "min": "1979-12-31"
        },
        {
            "type": "checkbox",
            "name": "Done",
            "value": "",
            "placeholder": "",
            "checked": ""
        }
    ]
}
```

Figure 4.7 - Webform json file example (not all the fields)

And with this configuration, the result of the website would be:

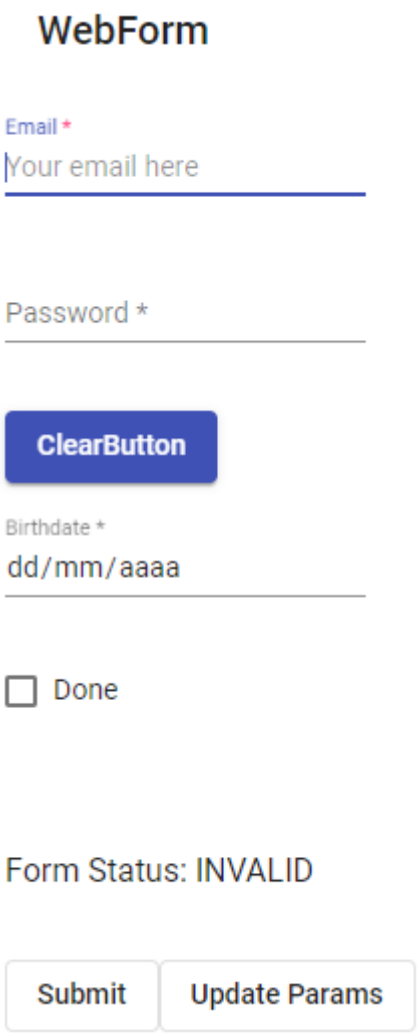


Figure 4.8 - Webform view in browser (with this json file config)

As we can see, there are empty fields, so it indicates that the format is invalid. But the error messages do not appear because we have not clicked the submit button yet. If we do it with the empty fields, the error messages will appear as in the previous case, if we do it with the fields filled in, we will be able to see the user data also as in the case explained before.

4.2. Image Displayer Component

In the case of this component, the input parameters are less and more limited. Let's see an example of how the image displayer component works. To do this, the *initial.json* file should have the following configuration:

```
{
    "compType": "WebDogs",
    "compId": 2,
    "compConfig": [
        {
        "api": "https://source.unsplash.com/category/nature",
        "message": "Pic of the Day",
        "height": "100",
        "width": "100",
        "fontapi": "https://fonts.googleapis.com/css?family=Nanum+Pen+Script"
        }
    ]
},
```

Figure 4.9 - Image displayer json file example

As we can see, we must indicate the type of component (webdogs corresponds to the image displayer component), an identifier and its configuration. We have chosen an image of size 100x100, which comes from the API https://source.unsplash.com/category/nature, with the message "Pic of the day" and the font of the address indicated by the *font api* parameter. The result on the screen is:

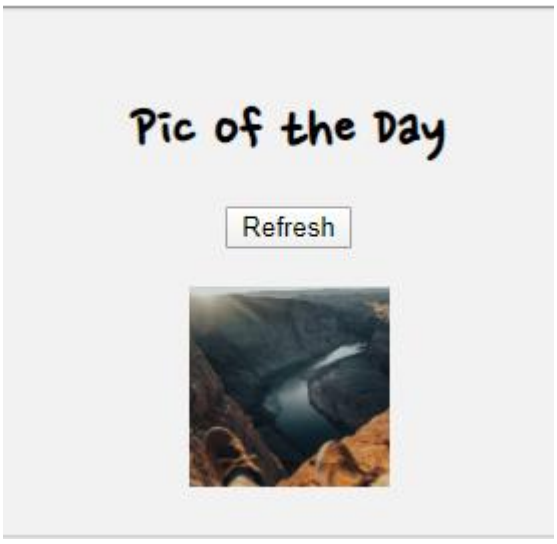


Figure 4.10 - Image displayer view in browser

We can use the refresh button and it will automatically show us another random image.

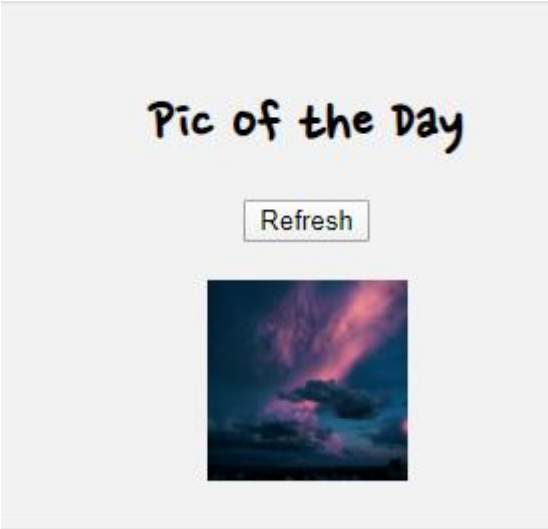


Figure 4.11 - Image displayer view in browser (refreshed)

And we can also change the size, the API and the message, changing the configuration parameters of the json file.

```
{
    "compType": "WebDogs",
    "compId": 5,
    "compConfig": [
        {
        "api": "https://source.unsplash.com/collection/881002",
        "message": "Pic of the Month",
        "height": "300",
        "width": "300",
        "fontapi": "https://fonts.googleapis.com/css?family=Nanum+Pen+Script"
        }
    ]
},
```

Figure 4.12 - Image displayer json file example (changed config)

And the result would be:

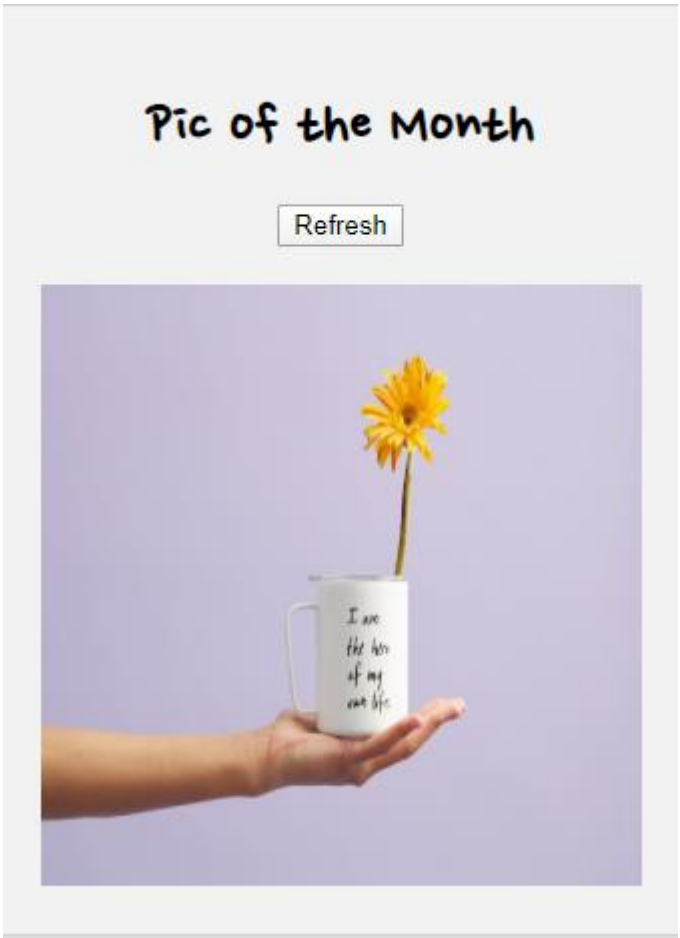


Figure 4.13 - Image displayer view in browser with new example

4.3. Scheduler Component

In the case of the scheduler, as already mentioned, we can indicate in the *initial.json* file the different events that we want to appear in our calendar. A configuration example would be:


```
{
    "compType": "Scheduler",
    "compId": 6,
    "compConfig": [
        {
            "Id": 1,
            "Subject": "Board Meeting",
            "StartTime": "2019, 10, 27, 9, 0",
            "EndTime": "2019, 10, 27, 11, 0"
        },
        {
            "Id": 2,
            "Subject": "Training session",
            "StartTime": "2019, 10, 27, 15, 0",
            "EndTime": "2019, 10, 27, 17, 0"
        },
        {
            "Id": 3,
            "Subject": "Sprint planning with team members",
            "StartTime": "2019, 10, 29, 9, 30",
            "EndTime": "2019, 10, 29, 11, 0"
        },
        {
            "Id": 4,
            "Subject": "Stack Point with supervisor",
            "StartTime": "2019, 10, 20, 11, 0",
            "EndTime": "2019, 10, 20, 14, 0"
        }
    ]
},
```


Figure 4.14 - Scheduler json file example

As we can see, we had 4 events:

Board Meeting – 27/10/2019 *09:00* → 27/10/2019 *11:00*

Training session – 27/10/2019 *15:00* → 27/10/2019 *17:00*

Sprint planning with team members – 29/10/2019 *09:30* → 29/10/2019 *11:00*

Stack point with supervisor – 20/10/2019 *11:00* → 20/10/2019 *14:00*

The result on the calendar would be:

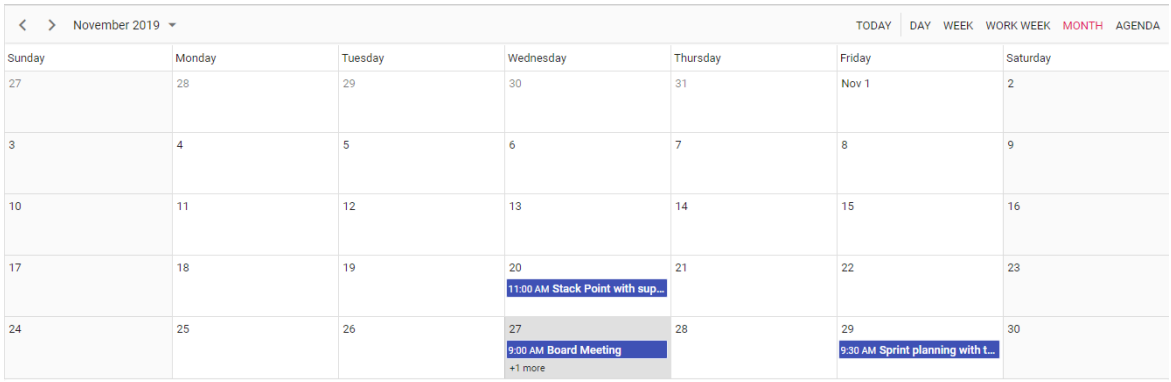


Figure 4.15 - Scheduler view in browser

For the visualization of the tasks there are four types of view:

1. Day
2. Week
3. Month
4. Work Week

The Work Week mode is the same as Week but only from Monday to Friday, and the Day mode is the same as Week but just seeing the schedule of that day. The previous example corresponds to the month view. We can also see the calendar in week mode.

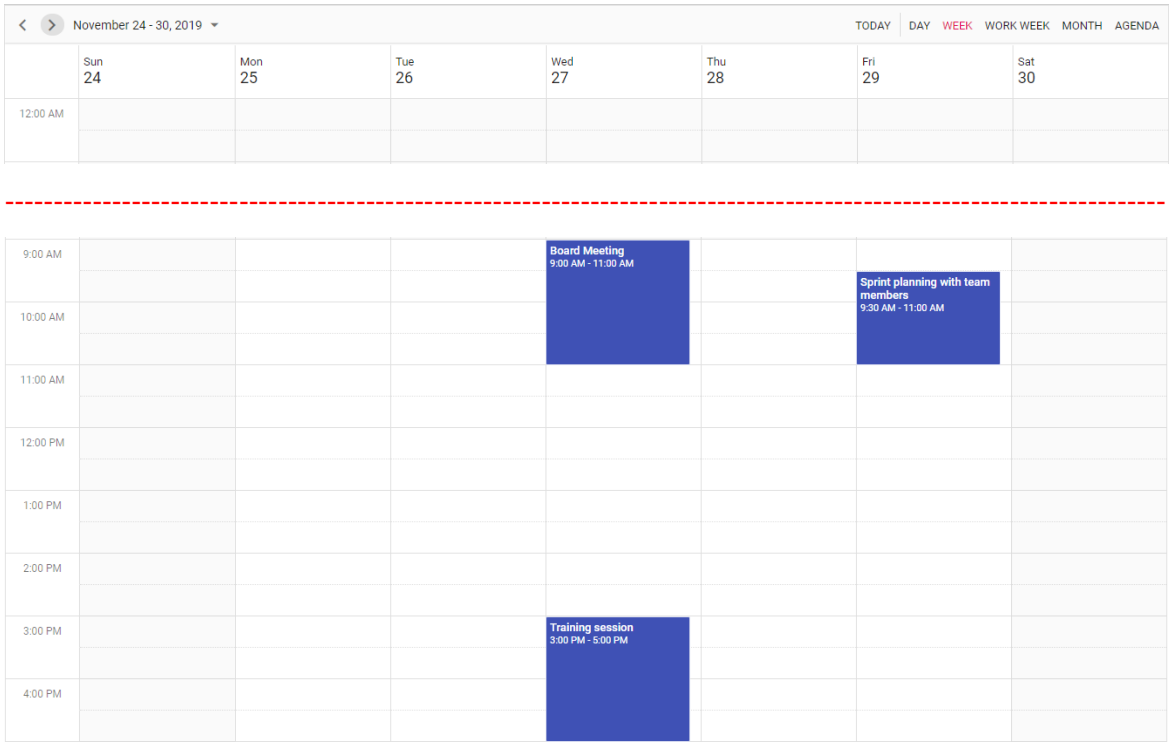


Figure 4.16 - Scheduler view with Week mode (part 1)

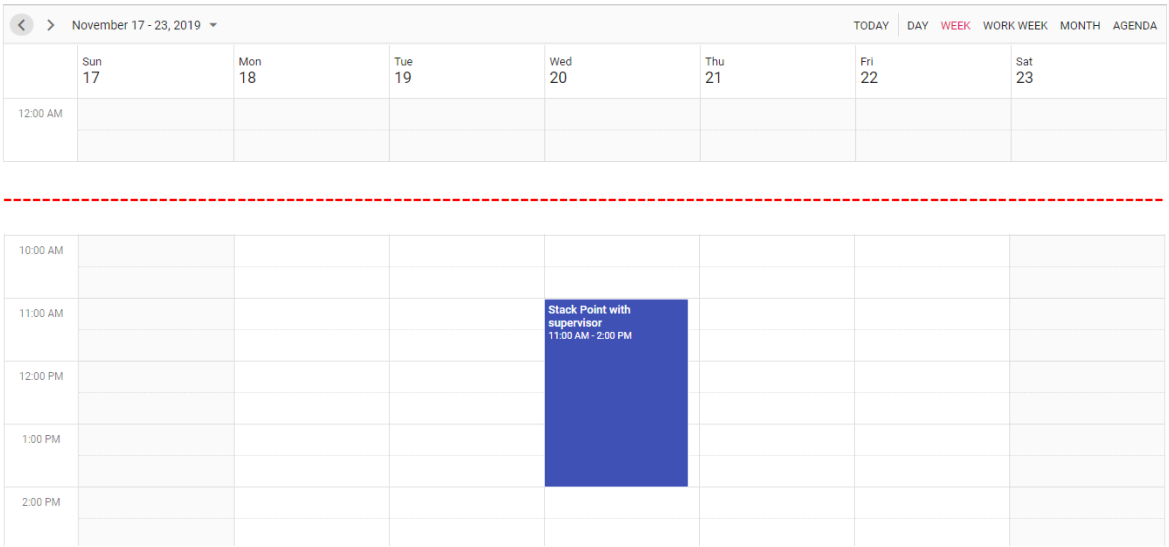


Figure 4.17 - Scheduler view with Week mode (part 2)

If we wish, we can add events when we have already created the scheduler, directly in the calendar. Just click on the time we want task to elapse, fill in the fields of the pop-up window that appears and create the task.

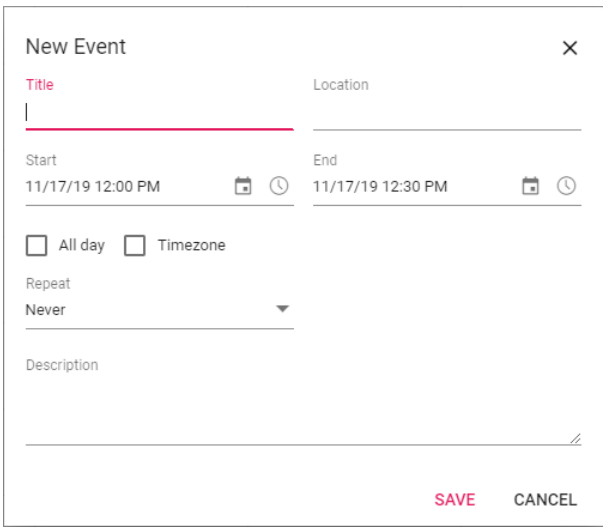


Figure 4.18 - New events configuration panel

We can also add, delete or modify the tasks once they are written in the calendar.

4.4. Default Component

The default component is only shown in case the *initial.json* configuration file is empty or the type of component that we specify in the file is not implemented in the library. An example of a json file where we indicate a different type of component to the three implemented so far (webform, webdogs or scheduler) would be this:

```
{
    "compType": "Other",
    "compId": 4,
    "compConfig": [
        {
        "clock": "16:20",
        "message": "Hello",
        "Div": "EUR"
        }
    ]
},
```

Figure 4.19 - Error component type json file example

And the result on the screen would be the next:

You have not entered any component or your component type is not registered

Figure 4.20 - Default component view in browser

5. Budget

This project is based on software development, no prototype or hardware has been used to create the library.

The software used is free, so nothing has been invested in licenses. All software used and implemented has been made with free or open source tools, so no cost related to this has been considered in the budget.

Therefore, the budget will only take into account the costs of the computers that have been used for the development of the project and personal salaries according to the time spent.

5.1. Equipment

For the development of the project no virtual payment tool has been used. The repository used (Bitbucket) is free and the management and code development programs (Angular, Visual Studio Code, etc.) as well.

What is necessary for the creation of the library is a computer. In this case we have used a personal laptop, with its basic accessories: mouse, charger and network cable. The laptop is a DELL model Latitude E5550. The average price in the market is 650€. The Ethernet cable could cost around 6€ and the mouse about 30€.

Equipment	**Price**	**Quantity**	**Total**
Laptop Dell E5550	650.00 €	1	650.00 €
Mouse	30.00 €	1	30.00 €
Ethernet cable	6.00 €	2	12.00 €
		Total	*692.00 €*

Table 5.1 - Equipment cost

5.2. Staff

The table includes the salary per hour corresponding to each worker, as well as the total worked hours and the final salary.

I count myself as a junior engineer. The project is valued at 18 ECTS credits, each one with 30 hours. To calculate the salary, we assume that a junior engineer can collect around 9€/h.

$$Total\ Hours\ TFG\ project\colon\ 18\ ECTS\ \times \frac{30\ hours}{ECTS} = 540\ h$$

Staff	*Salary/hour*	*Total Hours*	*Cost*
Junior Engineer	9.00 €	540	4,860.00 €

Table 5.2 - Personal salaries

5.3. Total

Concept	*Cost*
Equipment	692.00 €
Staff	4,860.00 €
Total	***5,552.00 €***

Table 5.3 - Total cost of the project

The total cost of the project has needed an investment of **5,552.00 €**.

6. Conclusions and future development:

First of all, we have to look back at the objective I had at the beginning, and determine whether I accomplished it or not. The main goal was to create an app-library that render dynamic components and learn how to use the Angular framework. Although there are lots of improvements that can be done, I consider that I have successfully accomplished the objective.

The components that have been implemented I think are very useful at a professional level and widely used in general. I think this project can be a good tool for many people and there are many use cases. It is also an application that can be in continuous growth; that you could add new components to the library without modifying or altering the existing ones, or modify some of them without affecting the rest. Regarding the implemented components, I think they could include more fields but those that can be configured are already enough to load a functional web with a minimum of basic content.

On a personal level, I think this TFG has helped me to understand and manage a project in a similar way to a professional project. The way to structure times, break down work into tasks, documentation, etc. I also think that thanks to this TFG I have learned a new programming language and how a framework widely used at a business level works, such as Angular.

As for future developments, as already mentioned, it is a library of components, we could implement as many components as we want, whatever we can think of or there is a need in the market. There is also an idea that came up in the company at the beginning of this project, which would be to join it with a project that transforms the hand-drawn wireframes into digital design files and front-end code. That is, we make a drawing of a web page in a notebook or on a touch screen, through a program with machine learning we identify the fields that appear in the image and we save the information in a json file. That json file could become the one we pass as an entry to our library, but there is still a long way to go in the case of this future development.

Bibliography:

[1] “Angular”. Google, 2019. [Online] Available: https://angular.io/

[2] Stephen Fluin. “Why Developers and Companies Choose Angular”. Medium, 2017. [Online] Available: https://medium.com/angular-japan-user-group/why-developers-and-companies-choose-angular-4c9ba6098e1c

[3] “Las 5 principales ventajas de usar Angular para crear aplicaciones web”. Campus MVP, 2017. [Online] Available: https://www.campusmvp.es/recursos/post/las-5-principales-ventajas-de-usar-angular-para-crear-aplicaciones-web.aspx

[4] Jordi Torres. “Las 7 razones para utilizar Angular en tus proyectos de desarrollo de web”. Offing, 2017. [Online] Available: https://offing.es/las-7-razones-para-utilizar-angular-4-en-tus-proyectos-de-desarrollo-web/

[5] Arturo Barrera. “JSON: ¿Qué es y para qué sirve?”. Next-u, 2015. [Online] Available: https://www.nextu.com/blog/que-es-json/

[6] Victor Garibay. “JSON marcando tendencias”. Medium, 2016. [Online] Available: https://medium.com/@victor.garibayy/qu%C3%A9-es-y-para-qu%C3%A9-sirve-json-be05fe02e67d

[7] Nicolás Avila. “Web Components con Angular Elements”. Medium, 2018. [Online] Available: https://medium.com/angular-chile/web-components-con-angular-elements-9bc6efd1265f

[8] Sandy Veliz. “Angular + Material Design | Instalación Angular Material”. Medium, 2019. [Online] Available: https://medium.com/@sandy.e.veliz/angular-material-design-instalaci%C3%B3n-angular-material-790caca5677b

[9] Prashank Jauhari. “Creating forms in angular 2 using json schema at run time”. Oodles Technologies, 2017. [Online] Available: https://www.oodlestechnologies.com/blogs/Creating-forms-in-angular-2-using-json-schema-at-run-time/

[10] Kevin Kreuzer. “The ultimate guide to set up your Angular library project”. Medium, 2019. [Online] Available: https://medium.com/angular-in-depth/the-ultimate-guide-to-set-up-your-angular-library-project-399d95b63500

[11] “How to get started easily with Syncfusion Angular 6 Scheduler?”. Syncfusion, 2018. [Online] Available: https://www.syncfusion.com/kb/9720/how-to-get-started-easily-with-syncfusion-angular-6-scheduler

[12] Esther Vaati. “Validación de formulario Angular con formularios reactivos y controlados por plantillas”. Envato tuts+, 2018. [Online] Available: https://code.tutsplus.com/es/tutorials/angular-form-validation-with-reactive-and-template-driven-forms--cms-32131

[13] Web StackOverflow. [Online] Available: https://stackoverflow.com/

[14] “Creating Custom Validators in Angular 7|8|9 Reactive Forms”. PositronX, 2019. [Online] Available: https://www.positronx.io/custom-validators-angular-7-reactive-forms/

[15] “Introduction to localStorage and sessionStorage”. Alligator, 2018. [Online] Available: https://alligator.io/js/introduction-localstorage-sessionstorage/

Appendix 1

Work Packages:

<table>
<tr><td colspan="2">Project: Introduction to Angular 8</td><td colspan="2">WP ref: (WP1)</td></tr>
<tr><td colspan="2">Major constituent: Software</td><td colspan="2">Sheet 1 of 1</td></tr>
<tr><td colspan="2" rowspan="2">Short description:

The main focus of this activity is to understand all the basic concepts and how angular 8 works.</td><td colspan="2">Planned start date 16/09/2019
Planned end date: 14/10/2019</td></tr>
<tr><td colspan="2">Start event: 16/09/2019
End event: 14/10/2019</td></tr>
<tr><td>Internal task T1: How to render dynamic components

Internal task T2: Create and build lazy loaded modules

Internal task T3: Investigate about web components advantages</td><td>Deliverables:</td><td>Dates:</td></tr>
</table>

Table A1. 1 - Work Package 1

<table>
<tr><td colspan="2">Project: Learn and implement Reactive Forms</td><td colspan="2">WP ref: (WP2)</td></tr>
<tr><td colspan="2">Major constituent: Software</td><td colspan="2">Sheet 1 of 1</td></tr>
<tr><td colspan="2" rowspan="2">Short description:

This work package will focus on learning the functionalities that we will use later to realize the final objective. The web format that we will create from the json file will be based mainly on reactive forms and web components.</td><td colspan="2">Planned start date 15/10/2019
Planned end date: 15/11/2019</td></tr>
<tr><td colspan="2">Start event: 15/10/2019
End event: 15/11/2019</td></tr>
<tr><td>Internal task T1: Build a simple reactive form with some fields like name, last name, mail, phone number…</td><td>Deliverables:</td><td>Dates:</td></tr>
</table>

Internal task T2: Create a web component that shows a random image or similar.		

Table A1. 2 - Work Package 2

Project: Get the html config from json file	**WP ref:** (WP3)	
Major constituent: Software	Sheet 1 of 1	
Short description: In this part of the project we will try that the configuration parameters from json file build the web app with the fields that we want. Once the goal is achieved, we will package and publish the library so that anyone can download, install and use it.	**Planned start date** 18/11/2019 **Planned end date:** 20/12/2019	
	Start event: 18/11/2019 **End event:** 20/12/2019	
Internal task T1: Get the configuration from the json file and be interpreted by the both components we have created (form and web image). **Internal task T2:** Verify that the fields of the forms and the web components belong to the file configuration. **Internal task T3:** Pack the project and create the library.	**Deliverables:**	**Dates:**

Table A1. 3 - Work Package 3

<table>
<tr><td colspan="2">Project: Implement calendar component</td><td colspan="2">WP ref: (WP4)</td></tr>
<tr><td colspan="2">Major constituent: Software</td><td colspan="2">Sheet 1 of 1</td></tr>
<tr><td colspan="2" rowspan="2">Short description:

To finalize the project we will implement a new type of component, the calendar. This component will complement the form and the image web component. In the json file we can indicate tasks and events, associated with its start and end date, which will be recorded in the calendar once the component is rendered.</td><td colspan="2">Planned start date 23/12/2019
Planned end date: 21/01/2020</td></tr>
<tr><td colspan="2">Start event: 23/12/2019
End event: 21/01/2020</td></tr>
<tr><td colspan="2">Internal task T1: Establish a sketch with the input data: start date, end date, description...

Internal task T2: Integrate this component to the existing library with the rest.

Internal task T3: Check that the data coming from the json corresponds to the tasks that appear in the calendar.

Internal task T4: Pack the project again and rebuild the library to add the new scheduler component</td><td>Deliverables:</td><td>Dates:</td></tr>
</table>

Table A1. 4 - Work Package 4

Appendix 2

Angular Framework

One of the big differences between a framework and a library is that the framework is a lot of “generic” functionalities prepared for us to do specific functionality. Instead, a library is a single generic function. Based on this definition we can say that a framework consists of several libraries written to be managed together. Under this thought, we can say that Angular prepared everything so that our application only uses the modules (or libraries) that we are going to need in our WebApp.

With this thought, when we start building our application with Angular, we will only have the main module called “core”, with it we will be able to run our application and write each of our components. If our application needs to generate routes we have to add the routing module, which Angular already provides us, and if we need to add forms, Angular also has an incredible module for that.

To build Angular applications we create:

- HTML templates that contain special Angular tags
- Class components that manage these templates
- Services that encapsulate application logic
- Modules that organize the components and services

Another of the great advantages of angular is its scalability. It allows to separate the roles that could be in a team of workers, such as engineers, designers, quality control, testers ... All this thanks to its component-based model.

Also, when maintaining applications, Angular covers this need. The fact of using Typescript allows to find bugs and errors with ease, and a quick and simple adaptation to the code of developers who are not familiar with it, due to its ability to immediately see the data that is manipulated in the application. Finally, Angular focuses on the ability to test and be tested.

Angular is reliable. Because it belongs to Google, it takes advantage of all its testing resources to help improve the framework. Each version of Angular, before being published, is already being used in hundreds of projects, and every change that is made, is validated against each Angular project within Google. That minimizes the possibility of errors or changes.

It is also good to note that Angular has a huge environment and there are thousands of tools at our disposal, such as libraries, code samples, consultation forums...

JSON format

JSON is able to define 6 types of values, 4 primitives (numbers, strings, nulls and booleans), and 2 structured (objects and matrices). JSON main features:

- JSON is just a data format.
- Requires using double quotes for strings and property names. Single quotes are not valid
- A comma or two points badly placed may cause a JSON file to not work
- It can take the form of any type of data that is valid to be included in a JSON, not just arrays or objects. For example, a string or a unique number could be valid JSON objects
- Unlike the JavaScript code, in which the object's properties may not be in quotes, in JSON only strings in quotes can be used as properties

An example of JSON format would be the storage of data related to a person, like this:

```
var person={
      "name":"Jaume Armengol",
      "age":22,
      "height":"184 cm",
      "weight":77
}
```

This creates the object named "person", since we indicate its value in keys. Inside the object we indicate its properties, with their respective name and value, separated by commas. All as many as we want. To access the information of an object, we can refer to its properties in this way:

`console.log('Age of person: ' + person.age)` → Age of person: 22

JSON format have some advantages and disadvantages.

Advantage:

- It is self-descriptive and easy to understand
- It is faster in any browser
- It is easier to read than XML
- It is lighter (bytes) in transmissions
- It parses faster
- High processing speed
- It can be understood natively by JavaScript parsers

Disadvantages:

- Some developers could find their brief notation confusing.
- It does not have the XML extensibility.
- It does not support large loads, only common data.
- For security it requires external mechanisms such as regular expressions.

Visual Studio Code

Visual Studio Code is a cross-platform programming editor developed by Microsoft. The first beta version of Visual Studio Code was published in November 2015 and the first stable version, Visual Studio Code 1.0, was published in April 2016. Since its inception, Visual Studio Code has maintained a very rapid development pace, and has publishes a new version at the beginning of each month. In addition, secondary versions that correct last minute failures are published almost every month.

Visual Studio Code is an application based on Electron. Electron is a framework for programming desktop graphic applications using web technologies, and includes Chromium (the free version of Google Chrome) as a graphics engine and the Node.js environment to run JavaScript.

Bitbucket

With this tool you can control all the changes that are made in the code. Thus, in a work team, if each one develops a different function, they can independently modify their respective lines of code to later mix all the work in a single branch. In the case of only one person working on the project (as is the case here), different versions can be saved in different branches as the project progresses. They also function as backups, the project is saved and if at any time you want to return to the point where we were in a branch, you can return without problems.

To perform all these functions bitbucket provides Sourcetree. Source Tree is a tool for both Git and Mercurial to work with these systems graphically. This way it is easier to make the commits (or changes) in the code, update it, undo what we do not want, upload the files to one or another branch ... It is a more visual way to use the repository.

Glossary

Framework: A framework or work environment is a standardized set of concepts, practices and criteria to focus on a particular type of problem that serves as a reference, to face and solve new problems of a similar nature.

SPA: A single-page application (SPA) is a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server.

CSS: Cascading Style Sheets (CSS) is a graphic design language to define and create the presentation of a structured document written in a markup language.

URL: A uniform resource locator (URL) is a Uniform Resource Identifier (URI) whose referred resources can change, that is, the address can point to time-varying resources. They are formed by a sequence of characters according to a model and standard format that designates resources in a network such as the Internet.

Bootstrapping: Bootstrapping is generally a term used to describe the boot, or startup process of any computer.

DOM: Document Object Model (DOM) is essentially a platform interface that provides a standard set of objects to represent HTML, XHTML and XML documents, a standard model on how these objects can be combined, and a standard interface to access and manipulate them. Through the DOM, programs can access and modify the content, structure and style of HTML and XML documents, which is what it was primarily designed for.

API: The application programming interface (API) is a set of subroutines, functions and procedures (or methods, in object-oriented programming) that offers a certain library to be used by other software as an abstraction layer. They are generally used in programming libraries.