Angular 5 and ASP.NET Core

Sobre la arquitectura

Construirá un cliente Angular 5 que consume un servicio RESTful Web API Core 2.

El lado del cliente:

- •Angular 5
- •CLI angular
- •Material angular

El lado del servidor:

- •.NET C # Web API Core 2
- •Dependencias de inyección
- •Autenticación JWT
- •Código de marco de entidad primero
- •servidor SQL

¿Que necesitas?

Comencemos eligiendo el IDE. Por supuesto, esta es solo mi preferencia, y puede usar la que le resulte más cómoda. En mi caso, usaré Visual Studio Code y Visual Studio 2017.

¿Por qué dos IDEs diferentes? Desde que Microsoft creó el Código de Visual Studio para el front-end, no puedo dejar de usar este IDE. De todos modos, también veremos cómo integrar Angular 5 dentro del proyecto de solución, que lo ayudará si es el tipo de desarrollador que prefiere depurar tanto el back-end como el front-end con solo un F5.

Sobre el back-end, puede instalar la última versión de Visual Studio 2017 que tiene una edición gratuita para desarrolladores pero es muy completa: Comunidad.

Entonces, aquí la lista de cosas que necesitamos instalar para este tutorial:

- Visual Studio Code
- Comunidad de Visual Studio 2017 (o cualquiera)
- Node.js v8.10.0
- SQL Server 2017

Nota

Verifique que está ejecutando al menos el Nodo 6.9.xy npm 3.xx ejecutándolo `node -v` y `npm -v` en una ventana de terminal o consola. Las versiones anteriores producen errores, pero las versiones más recientes están bien.

El frente

Inicio rápido

¡Que comience la fiesta! Lo primero que debemos hacer es instalar Angular CLI globalmente, así que abra el símbolo del sistema node.js y ejecute este comando:

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

Bien, ahora tenemos nuestro paquete de módulos. Esto generalmente instala el módulo bajo su carpeta de usuario. Un alias no debería ser necesario por defecto, pero si lo necesita puede ejecutar la siguiente línea:

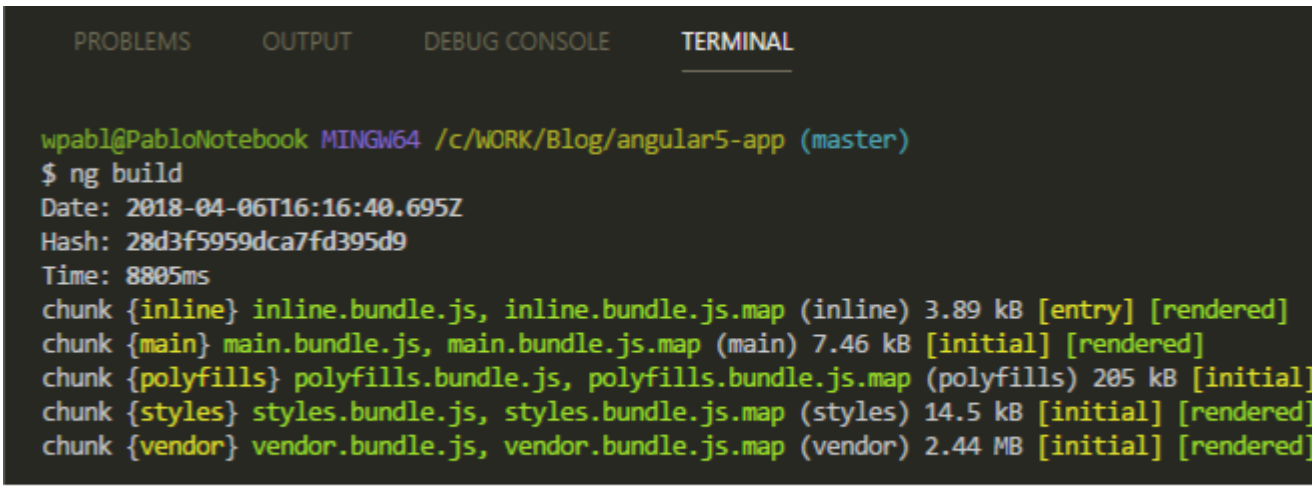
```
alias ng= "<UserFolder>/.npm/lib/node_modules/angular-cli/bin/ng"
```

El siguiente paso es crear el nuevo proyecto. Voy a llamarlo `angular5-app`. Primero, navegamos a la carpeta en la que queremos crear el sitio y luego:

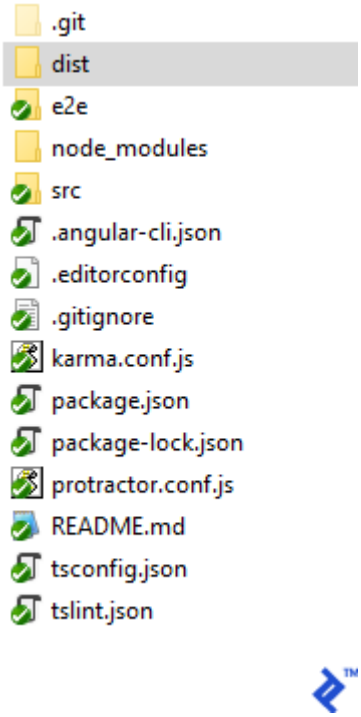
```
ng new angular5-app
```

Primera construcción

Si bien puede probar su nuevo sitio web recién en ejecución `ng serve --open`, le recomiendo probar el sitio desde su servicio web favorito. ¿Por qué? Bueno, algunos problemas solo pueden ocurrir en la producción, y construir el sitio con `ng build` es la forma más cercana de abordar este entorno. Luego podemos abrir la carpeta `angular5-app` con Visual Studio Code y ejecutar `ng build` en la terminal bash:



Se `dist` creará una nueva carpeta llamada y podemos servirla usando IIS o el servidor web que prefiera. Luego puede escribir la URL en el navegador y ... ¡listo!

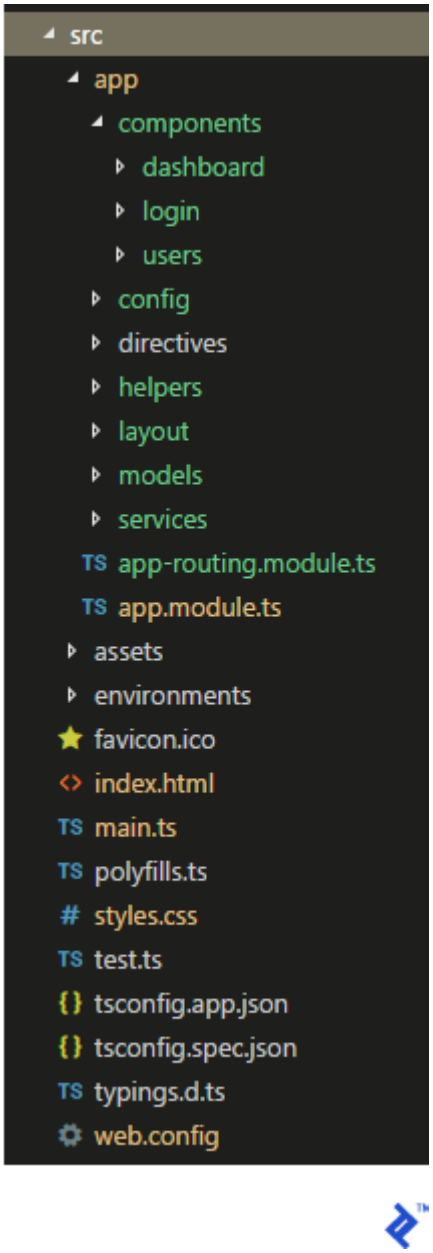


Nota

No es el propósito de este tutorial mostrar cómo configurar un servidor web, por lo que supongo que ya tiene ese conocimiento.

Welcome to app!!

La `src` carpeta



Mi `src` carpeta está estructurado de la siguiente manera: Dentro de la `app` carpeta que tenemos `components` en el que va a crear para cada componente angular de los `css`, `ts`, `spec`, y `html` archivos. También crearemos una `config` carpeta para mantener la configuración del sitio, `directives` tendremos todas nuestras directivas personalizadas, `helpers` albergará un código

común como el administrador de autenticación, `layout` contendrá los componentes principales como el cuerpo, la cabeza y los paneles laterales, `models` mantendrá lo que coincidirá con la parte posterior. Ver modelos finales, y finalmente `services` tendrá el código para todas las llamadas al back-end.
Fuera de la `app` carpeta mantendremos las carpetas creadas por defecto, como `assets` y `environments`, y también los archivos raíz.

Crear el archivo de configuración

Creemos un `config.ts` archivo dentro de nuestra `config` carpeta y llamemos a la clase `AppConfig`. Aquí es donde podemos establecer todos los valores que usaremos en diferentes lugares de nuestro código; por ejemplo, la URL de la API. Tenga en cuenta que la clase implementa una `get` propiedad que recibe, como parámetro, una estructura clave / valor y un método simple para obtener acceso al mismo valor. De esta manera, será fácil obtener los valores simplemente llamando `this.config.setting['PathAPI']` desde las clases que heredan de él.

```
import { Injectable } from '@angular/core';

@Injectable()

export class AppConfig {

    private _config: { [key: string]: string };

    constructor() {

        this._config = {

            PathAPI: 'http://localhost:50498/api/'

        };

    }
```

```
    get setting():{ [key: string ]: string } {
        return this._config;
    }
    get (key: any ) {
        return this._config[key];
    }
};
```

Material angular

Antes de comenzar el diseño, configuremos el marco del componente UI. Por supuesto, puede usar otros como Bootstrap, pero si le gusta el estilo de Material, lo recomiendo porque también es compatible con Google.

Para instalarlo, solo necesitamos ejecutar los siguientes tres comandos, que podemos ejecutar en el terminal de Visual Studio Code:

```
npm install --save @angular/material @angular/cdk
```

```
npm install --save @angular/animations
```

```
npm install --save hammerjs
```

El segundo comando se debe a que algunos componentes materiales dependen de animaciones angulares. También recomiendo leer la página oficial para comprender qué navegadores son compatibles y qué es un polyfill.

El tercer comando se debe a que algunos componentes de Material dependen de HammerJS para los gestos.

Ahora podemos proceder a importar los módulos componentes que queremos usar en nuestro `app.module.ts` archivo:

```
import {MatButtonModule, MatCheckboxModule} from '@angular/material';
import {MatInputModule} from '@angular/material/input';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatSidenavModule} from '@angular/material/sidenav';

// ...

@NgModule({
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        MatButtonModule,
        MatCheckboxModule,
        MatInputModule,
        MatFormFieldModule,
        MatSidenavModule,
        AppRoutingModule,
        HttpClientModule
    ],
```

El siguiente paso es cambiar el `style.css` archivo, agregando el tipo de tema que desea usar:

```
@import "~@angular/material/prebuilt-themes/deeppurple-amber.css";
```

Ahora importe HammerJS agregando esta línea en el `main.ts` archivo:

```
import 'hammerjs';
```

Y finalmente, todo lo que nos falta es agregar los íconos de Material `index.html`, dentro de la sección principal:

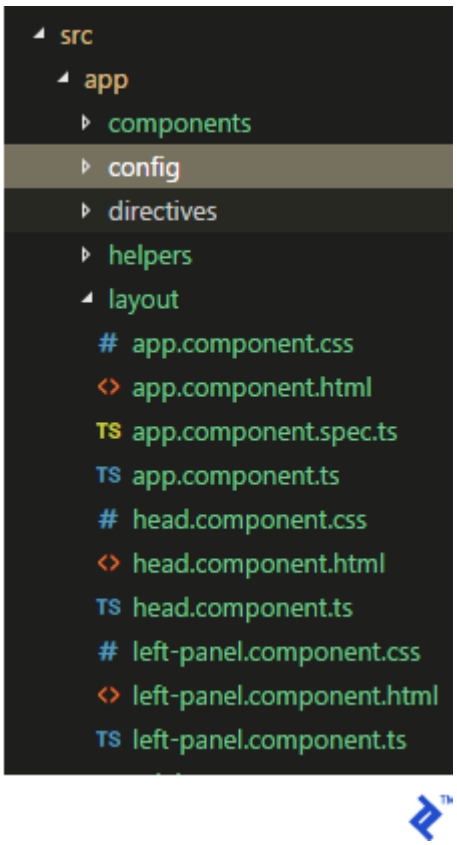
```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
```

El diseño

En este ejemplo, crearemos un diseño simple como este:

MENU

La idea es abrir / ocultar el menú haciendo clic en algún botón en el encabezado. Angular Responsive hará el resto del trabajo por nosotros. Para hacer esto, crearemos una `layout` carpeta y colocaremos dentro de ella los `app.component` archivos creados por defecto. Pero también crearemos los mismos archivos para cada sección del diseño como puede ver en la siguiente imagen. Entonces, `app.component` será el cuerpo, `head.component` el encabezado y `left-panel.component` el menú.



Ahora cambiemos de la `app.component.html` siguiente manera:

```
<div *ngIf="authentication">

    <app-head></app-head>

    <button type="button" mat-button (click)="drawer.toggle()">

        Menu
```

```
    </button>
    <mat-drawer-container class="example-container" autosize>
        <mat-drawer #drawer class="example-sidenav" mode="side">
            <app-left-panel></app-left-panel>
        </mat-drawer>
        <div>
            <router-outlet></router-outlet>
        </div>
    </mat-drawer-container>
</div>
<div *ngIf="!authentication"><app-login></app-login></div>
```

Básicamente tendremos una `authentication` propiedad en el componente que nos permitirá eliminar el encabezado y el menú si el usuario no ha iniciado sesión, y en su lugar, mostrar una página de inicio de sesión simple.
Se `head.component.html` ve así:

```
<h1> {{title}} </h1>
<button mat-button [routerLink]=" ['./logout'] "> Logout! </button>
```

Solo un botón para cerrar la sesión del usuario; volveremos a esto más tarde. En cuanto a `left-panel.component.html`, por ahora solo cambie el HTML a:

```
<nav>
    <a routerLink="/dashboard"> Dashboard </a>
    <a routerLink="/users"> Users </a>
```

```
    </nav>
```

Lo hemos mantenido simple: hasta ahora son solo dos enlaces para navegar a través de dos páginas diferentes. (También volveremos a esto más adelante).

Ahora, así es como se ven los archivos TypeScript de la cabeza y del componente izquierdo:

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-head',
    templateUrl: './head.component.html',
    styleUrls: [ './head.component.css' ]
})
export class HeadComponent {
    title = 'Angular 5 Seed';
}
```

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-left-panel',
    templateUrl: './left-panel.component.html',
    styleUrls: [ './left-panel.component.css' ]
```

```
})

export class LeftPanelComponent {

    title = 'Angular 5 Seed';

}
```

¿Pero para qué sirve el código TypeScript `app.component`? Dejaremos un pequeño misterio aquí y lo detendremos por un tiempo, y volveremos a esto después de implementar la autenticación.

Enrutamiento

Bien, ahora tenemos material angular que nos ayuda con la interfaz de usuario y un diseño simple para comenzar a construir nuestras páginas. Pero, ¿cómo podemos navegar entre páginas?

Para crear un ejemplo simple, creemos dos páginas: "Usuario", donde podemos obtener una lista de los usuarios existentes en la base de datos, y "Panel de control", una página donde podemos mostrar algunas estadísticas.

Dentro de la `app` carpeta crearemos un archivo llamado `app-routing.modules.ts` así:

```
import { NgModule }             from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { AuthGuard }            from './helpers/canActivateAuthGuard';

import { LoginComponent }       from './components/login/login.component';

import { LogoutComponent }      from './components/login/logout.component';

import { DashboardComponent }   from
'./components/dashboard/dashboard.component';
```

```
import { UsersComponent } from './components/users/users.component';

const routes: Routes = [
    { path: '', redirectTo: '/dashboard', pathMatch: 'full', canActivate: [AuthGuard] },
    { path: 'login', component: LoginComponent},
    { path: 'logout', component: LogoutComponent},
    { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] },
    { path: 'users', component: UsersComponent,canActivate: [AuthGuard] }
];

@NgModule({
    imports: [ RouterModule.forRoot(routes) ],
    exports: [ RouterModule ]
})

export class AppRoutingModule {}
```

Es así de simple: solo importando `RouterModule` y `Routes` desde `@angular/router`, podemos mapear las rutas que queremos implementar. Aquí estamos creando cuatro caminos:

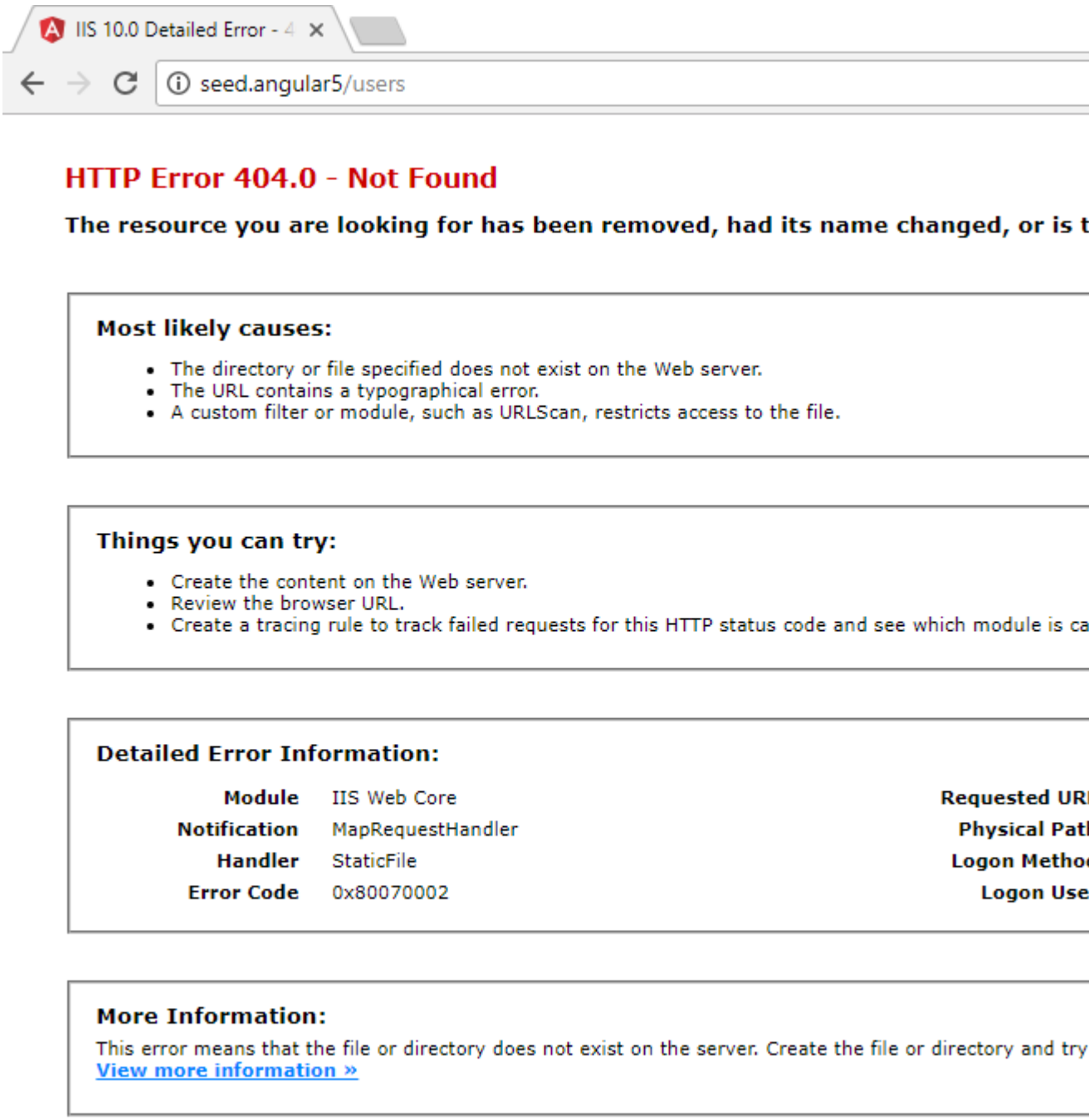
- `/dashboard`: Nuestra página de inicio
- `/login`: La página donde el usuario puede autenticarse
- `/logout`: Una ruta simple para cerrar la sesión del usuario
- `/users`: Nuestra primera página donde queremos enumerar los usuarios desde el back-end

Tenga en cuenta que `dashboard` es nuestra página de forma predeterminada, por lo que si el usuario escribe la URL `/`, la página se redirigirá automáticamente a esta página. Además, eche un vistazo al `canActivate` parámetro: aquí estamos creando una referencia a la clase `AuthGuard`, que nos permitirá verificar si el

usuario ha iniciado sesión. Si no, redirige a la página de inicio de sesión. En la siguiente sección, te mostraré cómo crear esta clase. Ahora, todo lo que tenemos que hacer es crear el menú. ¿Recuerdas en la sección de diseño cuando creamos el `left-panel.component.html` archivo para que se vea así?

```
<nav>

    <a routerLink="/dashboard"> Dashboard </a>

    <a routerLink="/users"> Users </a>

  </nav>
```

Aquí es donde nuestro código se encuentra con la realidad. Ahora podemos compilar el código y probarlo en la URL: debería poder navegar desde la página Panel a Usuarios, pero ¿qué sucede si escribe la URL `our.site.url/users` en el navegador directamente?



Tenga en cuenta que este error también aparece si actualiza el navegador después de navegar con éxito a esa URL a través del panel lateral de la aplicación. Para comprender este error,

permítame referirme a los documentos oficiales donde está realmente claro:

> Una aplicación enrutada debe soportar enlaces profundos. Un enlace profundo es una URL que especifica una ruta a un componente dentro de la aplicación. Por ejemplo, `http://www.mysite.com/users/42` es un enlace profundo a la página de detalles del héroe que muestra al héroe con id: 42.
> No hay ningún problema cuando el usuario navega a esa URL desde un cliente en ejecución. El enrutador angular interpreta la URL y las rutas a esa página y héroe.
>
> Pero al hacer clic en un enlace en un correo electrónico, ingresarlo en la barra de direcciones del navegador o simplemente actualizar el navegador mientras está en la página de detalles del héroe: todas estas acciones son

> manejadas por el navegador, fuera de la aplicación en ejecución. El navegador realiza una solicitud directa al servidor para esa URL, sin pasar por el enrutador.
>
> Un servidor estático devuelve rutinariamente index.html cuando recibe una solicitud de `http://www.mysite.com/`. Pero rechaza `http://www.mysite.com/users/42` y devuelve un error 404 - No encontrado a menos que esté configurado para devolver index.html en su lugar.

Para solucionar este problema es muy simple, solo necesitamos crear la configuración del archivo del proveedor de servicios. Como estoy trabajando con IIS aquí, le mostraré cómo hacerlo en este entorno, pero el concepto es similar para Apache o cualquier otro servidor web.

Entonces creamos un archivo dentro de la `src` carpeta llamada `web.config` que se ve así:

```
<?xml version="1.0"?>

<configuration>

    <system.webServer>

    <rewrite>
```

```
      <rules>
        <rule name="Angular Routes" stopProcessing="true">
          <match url=".*" />
          <conditions logicalGrouping="MatchAll">
            <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
            <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
          </conditions>
          <action type="Rewrite" url="/index.html" />
        </rule>
      </rules>
    </rewrite>
  </system.webServer>
  <system.web>
    <compilation debug="true"/>
  </system.web>
</configuration>
```

Luego, debemos asegurarnos de que este activo se copiará en la carpeta desplegada. Todo lo que necesitamos hacer es cambiar nuestro archivo de configuración de CLI angular `angular-cli.json`:

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "project": {
```

```
    "name": "angular5-app"
  },
  "apps": [
    {
      "root": "src",
      "outDir": "dist",
      "assets": [
        "assets",
        "favicon.ico",
        "web.config" // or whatever equivalent is required by your web server
      ],
      "index": "index.html",
      "main": "main.ts",
      "polyfills": "polyfills.ts",
      "test": "test.ts",
      "tsconfig": "tsconfig.app.json",
      "testTsconfig": "tsconfig.spec.json",
      "prefix": "app",
      "styles": [
        "styles.css"
      ],
```

```
      "scripts": [],
      "environmentSource": "environments/environment.ts",
      "environments": {
        "dev": "environments/environment.ts",
        "prod": "environments/environment.prod.ts"
      }
    }
  ],
  "e2e": {
    "protractor": {
      "config": "./protractor.conf.js"
    }
  },
  "lint": [
    {
      "project": "src/tsconfig.app.json",
      "exclude": "**/node_modules/**"
    },
    {
      "project": "src/tsconfig.spec.json",
      "exclude": "**/node_modules/**"
```

```
    },
    {
      "project": "e2e/tsconfig.e2e.json",
      "exclude": "**/node_modules/**"
    }
  ],
  "test": {
    "karma": {
      "config": "./karma.conf.js"
    }
  },
  "defaults": {
    "styleExt": "css",
    "component": {}
  }
}
```

Autenticación

¿Recuerdas cómo `AuthGuard` implementamos la clase para establecer la configuración de enrutamiento? Cada vez que navegamos a una página diferente, usaremos esta clase para verificar si el usuario está autenticado con un token. Si no, redirigiremos automáticamente a la página de inicio de sesión. El archivo para

esto es: `canActivateAuthGuard.ts` créelo dentro de la `helpers` carpeta y haga que se vea así:

```
import { CanActivate, Router } from '@angular/router';

import { Injectable } from '@angular/core';

import { Observable } from 'rxjs/Observable';

import { Helpers } from './helpers';

import { ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable()

export class AuthGuard implements CanActivate {

    constructor( private router: Router, private helper: Helpers ) {}

    canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
Observable< boolean > | boolean {

        if (! this .helper.isAuthenticated()) {

            this .router.navigate([ '/login' ]);

            return false ;

        }

        return true ;

    }

}
```

Entonces, cada vez que cambiemos la página `canActivate`, se llamará al método , que verificará si el usuario está autenticado, y si no, usamos nuestra `Router` instancia para redirigir a la página de inicio de sesión. ¿Pero cuál es este nuevo método en la `Helper` clase? Debajo de la `helpers` carpeta, creemos un

archivo `helpers.ts`. Aquí tenemos que administrar `localStorage`, donde almacenaremos el token que obtenemos del back-end.

Nota

Al respecto `localStorage`, también puede usar cookies o `sessionStorage`, y la decisión dependerá del comportamiento que queramos implementar. Como su nombre lo indica, `sessionStorage` solo está disponible durante la sesión del navegador y se elimina cuando se cierra la pestaña o ventana; sin embargo, sobrevive a la recarga de la página. Si los datos que está almacenando deben estar disponibles de forma continua, entonces `localStorage` es preferible hacerlo `sessionStorage`. Las cookies son principalmente para leer del lado del servidor, mientras `localStorage` que solo se pueden leer del lado del cliente. Entonces, la pregunta es, en su aplicación, ¿quién necesita estos datos, el cliente o el servidor?

```
import { Injectable } from '@angular/core';

import { Observable } from 'rxjs';

import { Subject } from 'rxjs/Subject';

@Injectable()

export class Helpers {

    private authenticationChanged = new Subject<boolean>();

    constructor() {



    }

    public isAuthenticated(): boolean {

        return (!(window.localStorage['token'] === undefined ||

            window.localStorage['token'] === null ||

            window.localStorage['token'] === 'null' ||

            window.localStorage['token'] === 'undefined' ||
```

```
            window.localStorage['token'] === '' ));
    }
    public isAuthenticationChanged(): any {
        return this.authenticationChanged.asObservable();
    }
    public getToken(): any {
        if ( window.localStorage['token'] === undefined ||
            window.localStorage['token'] === null ||
            window.localStorage['token'] === 'null' ||
            window.localStorage['token'] === 'undefined' ||
            window.localStorage['token'] === '' ) {
            return '';
        }
        let obj = JSON.parse( window.localStorage['token']);
        return obj.token;
    }
    public setToken(data: any ): void {
        this.setStorageToken( JSON.stringify(data));
    }
    public failToken(): void {
        this.setStorageToken( undefined );
```

```
    }

    public logout(): void {
        this.setStorageToken(undefined);
    }

    private setStorageToken(value: any): void {
        window.localStorage['token'] = value;
        this.authenticationChanged.next(this.isAuthenticated());
    }
}
```

¿Nuestro código de autenticación tiene sentido ahora? Regresaremos a la `Subject` clase más tarde, pero ahora volvamos un minuto a la configuración de enrutamiento. Echa un vistazo a esta línea:

```
  _{ path: 'logout', component: LogoutComponent},_

_~~~_
```

This is our component to log out of the site, and it's just a simple class to clean out the `localStorage`. Let's create it under the `components/login` folder with the name of `logout.component.ts`:

```
~~~ts
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { Helpers } from '../../helpers/helpers';

@Component({
```

```
    selector: 'app-logout',
    template: '<ng-content></ng-content>'
})
export class LogoutComponent implements OnInit {
    constructor( private router: Router, private helpers: Helpers ) { }
    ngOnInit() {
        this.helpers.logout();
        this.router.navigate([ '/login' ]);
    }
}
```

Entonces, cada vez que vamos a la URL `/logout`, `localStorage` se eliminará y el sitio redirigirá a la página de inicio de sesión. Finalmente, creemos `login.component.ts` así:

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { TokenService } from '../../services/token.service';
import { Helpers } from '../../helpers/helpers';
@Component({
    selector: 'app-login',
    templateUrl: './login.component.html',
    styleUrls: [ './login.component.css' ]
})
```

```
export class LoginComponent implements OnInit {

    constructor ( private helpers: Helpers, private router: Router, private tokenService:
TokenService ) { }

    ngOnInit() {

    }

    login(): void {

        let authValues = { "Username" : "pablo" , "Password" : "secret" };

        this .tokenService.auth(authValues).subscribe( token => {

            this .helpers.setToken(token);

            this .router.navigate([ '/dashboard' ]);

        });

    }

}
```

Como puede ver, por el momento hemos codificado nuestras credenciales aquí. Tenga en cuenta que aquí estamos llamando a una clase de servicio; crearemos estas clases de servicios para obtener acceso a nuestro back end en la siguiente sección.

Finalmente, necesitamos volver al `app.component.ts` archivo, el diseño del sitio. Aquí, si el usuario está autenticado, mostrará las secciones de menú y encabezado, pero si no, el diseño cambiará para mostrar solo nuestra página de inicio de sesión.

```
export class AppComponent implements AfterViewInit {

    subscription: Subscription;
```

```
    authentication:  boolean ;

    constructor ( private helpers: Helpers ) {

    }

    ngAfterViewInit() {

        this .subscription =  this .helpers.isAuthenticationChanged().pipe(

            startWith( this .helpers.isAuthenticated()),

            delay( 0 )).subscribe( (value) =>

                this .authentication = value

            );

    }

    title =  'Angular 5 Seed' ;

    ngOnDestroy() {

        this .subscription.unsubscribe();

    }

}
```

¿Recuerdas la `Subject` clase en nuestra clase de ayuda? Este es un `Observable` . `Observable` s brindan soporte para pasar mensajes entre editores y suscriptores en su aplicación. Cada vez que cambie el token de autenticación, la `authentication` propiedad se actualizará. Al revisar el `app.component.html` archivo, probablemente tenga más sentido ahora:

```
<div *ngIf= "authentication" >

    <app-head>< /app-head>
```

```
  <button type="button" mat-button (click)="drawer.toggle()">
    Menu
  </ button>
  <mat-drawer-container class = "example-container" autosize>
    <mat-drawer #drawer class = "example-sidenav" mode= "side" >
      <app-left-panel>< /app-left-panel>
    </m at-drawer>
    <div>
      <router-outlet>< /router-outlet>
    </ div>
  < /mat-drawer-container>
</ div>
<div *ngIf= "!authentication" ><app-login>< /app-login></ div>
```

Servicios

En este punto, estamos navegando a diferentes páginas, autenticando nuestro lado del cliente y presentando un diseño muy simple. Pero, ¿cómo podemos obtener datos del back-end? Recomiendo encarecidamente hacer todo el acceso de fondo de las clases de servicio en particular. Nuestro primer servicio estará dentro de la `services` carpeta, llamada `token.service.ts`:

```
import { Injectable } from '@angular/core' ;
import { HttpClient, HttpHeaders } from '@angular/common/http' ;
import { Observable } from 'rxjs/Observable' ;
```

```
import { of } from 'rxjs/observable/of';

import { catchError, map, tap } from 'rxjs/operators';

import { AppConfig } from '../config/config';

import { BaseService } from './base.service';

import { Token } from '../models/token';

import { Helpers } from '../helpers/helpers';

@Injectable()

export class TokenService extends BaseService {

    private pathAPI = this.config.setting['PathAPI'];

    public errorMessage: string;

    constructor( private http: HttpClient, private config: AppConfig, helper: Helpers ) { super(helper); }

    auth(data: any): any {

        let body = JSON.stringify(data);

        return this.getToken(body);

    }

    private getToken (body: any): Observable< any > {

        return this.http.post< any >( this.pathAPI + 'token', body, super.header()).pipe(

            catchError( super.handleError)

        );

    }
```

```
}
```

La primera llamada al back-end es una llamada POST a la API de token. La API de token no necesita la cadena de token en el encabezado, pero ¿qué sucede si llamamos a otro punto final? Como puede ver aquí, `TokenService` (y las clases de servicio en general) heredan de la `BaseService` clase. Echemos un vistazo a esto:

```
import { Injectable } from '@angular/core';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable } from 'rxjs/Observable';

import { of } from 'rxjs/observable/of';

import { catchError, map, tap } from 'rxjs/operators';

import { Helpers } from '../helpers/helpers';

@Injectable()

export class BaseService {

    constructor( private helper: Helpers ) { }

    public extractData(res: Response) {

        let body = res.json();

        return body || {};

    }


    public handleError(error: Response | any ) {

        // In a real-world app, we might use a remote logging infrastructure

        let errMsg: string;
```

```
            if (error instanceof Response) {
                const body = error.json() || '';
                const err = body || JSON.stringify(body);
                errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
            } else {
                errMsg = error.message ? error.message : error.toString();
            }
            console.error(errMsg);
            return Observable.throw(errMsg);

        }

        public header() {

            let header = new HttpHeaders({ 'Content-Type': 'application/json' });

            if (this.helper.isAuthenticated()) {
                header = header.append('Authorization', 'Bearer ' +
this.helper.getToken());
            }

```

```
            return { headers: header };
        }

        public setToken(data: any ) {

            this.helper.setToken(data);
        }
        public failToken(error: Response | any ) {

            this.helper.failToken();
            return this.handleError(Response);
        }
    }
```

Entonces, cada vez que hacemos una llamada HTTP, implementamos el encabezado de la solicitud simplemente usando `super.header`. Si el token está dentro `localStorage`, se agregará dentro del encabezado, pero si no, solo configuraremos el formato JSON. Otra cosa que podemos ver aquí es lo que sucede si falla la autenticación.

El componente de inicio de sesión llamará a la clase de servicio y la clase de servicio llamará al back-end. Una vez que tengamos el token, la clase auxiliar administrará el token, y ahora estamos listos para obtener la lista de usuarios de nuestra base de datos.

Para obtener datos de la base de datos, primero asegúrese de hacer coincidir las clases de modelo con los modelos de vista de fondo en nuestra respuesta.

En `user.ts`:

```
export class User {

    id: number;

    name: string;

}
```

Y podemos crear ahora el `user.service.ts` archivo:

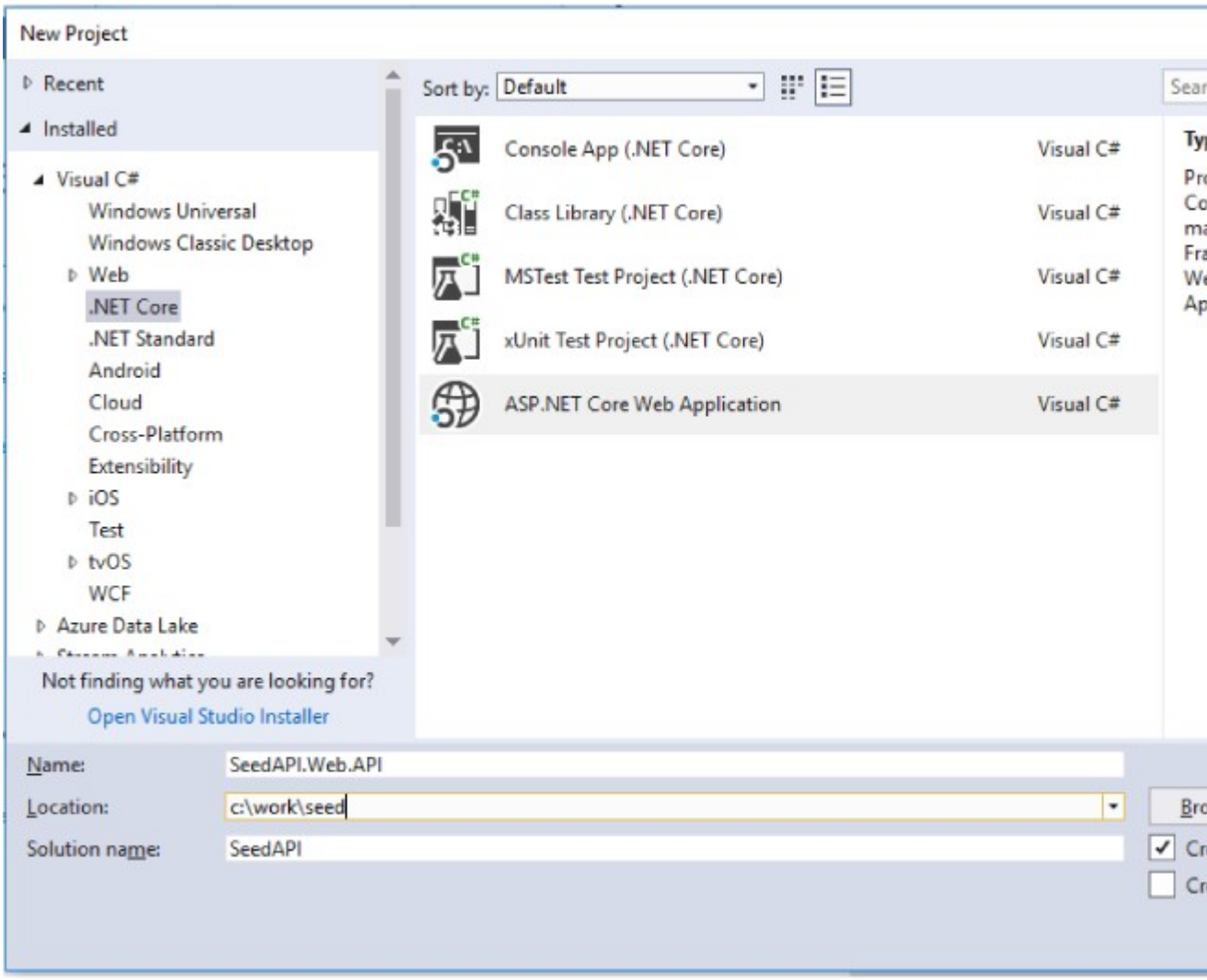
```
import { Injectable } from '@angular/core';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable } from 'rxjs/Observable';

import { of } from 'rxjs/observable/of';

import { catchError, map, tap } from 'rxjs/operators';

import { BaseService } from './base.service';

import { User } from '../models/user';

import { AppConfig } from '../config/config';

import { Helpers } from '../helpers/helpers';

@Injectable()

export class UserService extends BaseService {

    private pathAPI = this.config.setting['PathAPI'];

    constructor( private http: HttpClient, private config: AppConfig, helper: Helpers ) {
super(helper); }

    /** GET heroes from the server */

    getUsers (): Observable<User[]> {
```

```
        return this.http.get(this.pathAPI + 'user', super.header()).pipe(
        catchError(super.handleError));
    }
```

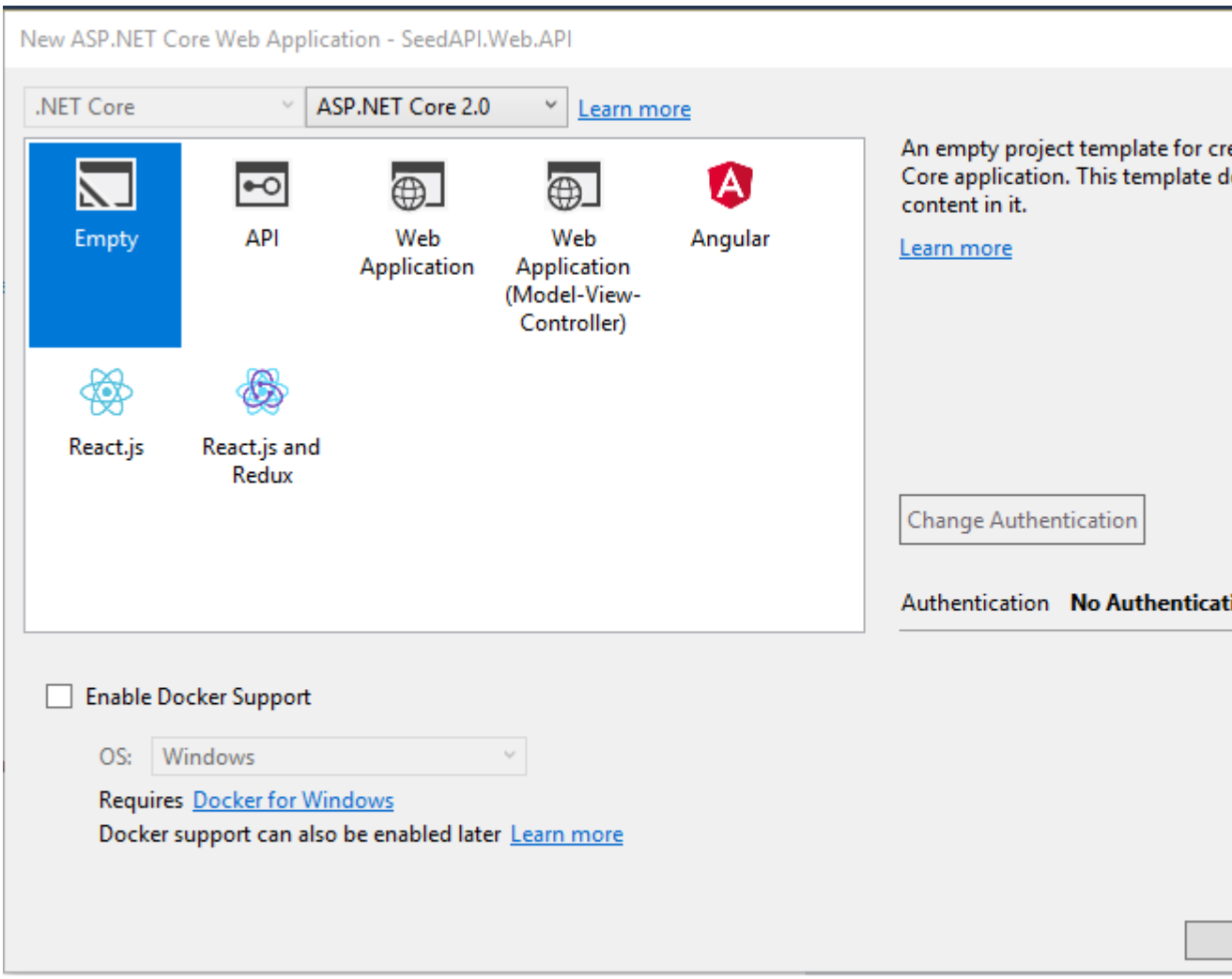
El back end

Inicio rápido

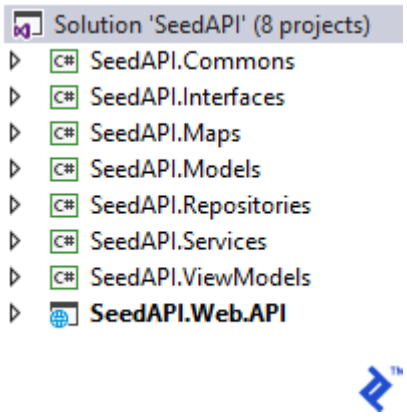
Bienvenido al primer paso de nuestra aplicación Web API Core 2. Lo primero que necesitamos es crear una aplicación web ASP.Net Core, a la que llamaremos `SeedAPI.Web.API`.



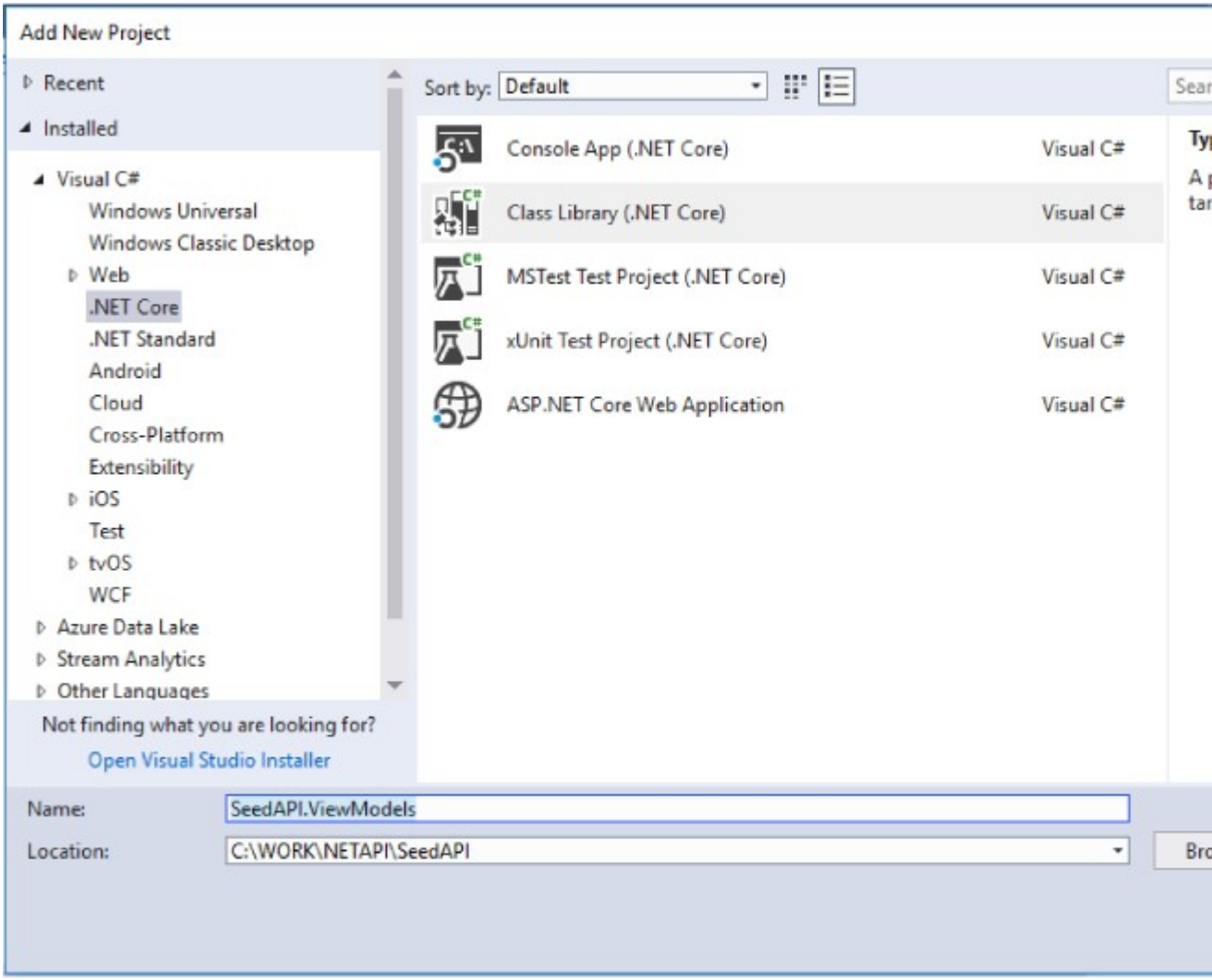
Asegúrese de elegir la plantilla Vacío para un comienzo limpio como puede ver a continuación:



Eso es todo, creamos la solución comenzando con una aplicación web vacía. Ahora nuestra arquitectura será la que enumeramos a continuación, por lo que tendrá que crear los diferentes proyectos:



Para hacer esto, para cada uno simplemente haga clic derecho en la Solución y agregue un proyecto de “Biblioteca de clases (.NET Core)”.

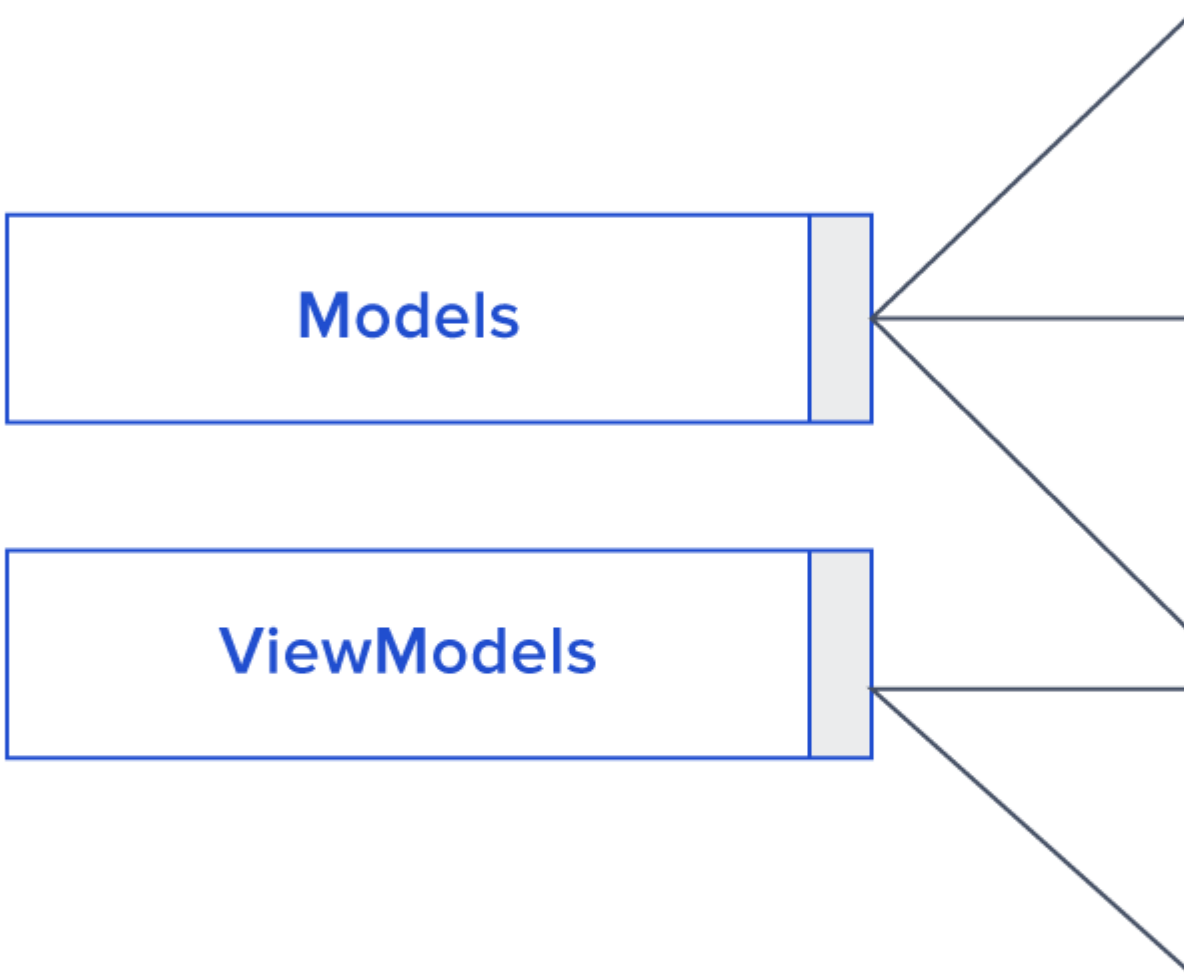


La arquitectura

En la sección anterior creamos ocho proyectos, pero ¿para qué sirven? Aquí hay una descripción simple de cada uno:

- `Web.API`: Este es nuestro proyecto de inicio y donde se crean los puntos finales. Aquí configuraremos JWT, dependencias de inyección y controladores.
- `ViewModels`: Aquí realizamos conversiones del tipo de datos que los controladores devolverán en las respuestas al front-end. Es una buena práctica hacer coincidir estas clases con los modelos front-end.
- `Interfaces`: Esto será útil para implementar dependencias de inyección. El beneficio convincente de un lenguaje de tipo estático es que el compilador puede ayudar a verificar que se cumpla realmente un contrato en el que se basa su código.
- `Commons`: Todos los comportamientos compartidos y el código de utilidad estarán aquí.
- `Models`: Es una buena práctica no hacer coincidir la base de datos directamente con el front-end `ViewModels`, por lo que el objetivo `Models` es crear clases de base de datos de entidades independientes del front-end. Eso nos permitirá en el futuro cambiar nuestra base de datos sin tener necesariamente un impacto en nuestro front-end. También ayuda cuando simplemente queremos refactorizar.
- `Maps`: Aquí es donde hacemos un mapa `ViewModels` de `Models` y viceversa. Este paso se llama entre controladores y servicios.
- `Services`: Una biblioteca para almacenar toda la lógica de negocios.
- `Repositories`: Este es el único lugar donde llamamos a la base de datos.

Las referencias se verán así:


Models
ViewModels

Autenticación basada en JWT

En esta sección, veremos la configuración básica de la autenticación de tokens y profundizaremos un poco en el tema de la seguridad.

Para comenzar a configurar el token web JSON (JWT), creemos la siguiente clase dentro de la `App_Start` carpeta llamada `JwtTokenConfig.cs`. El código dentro se verá así:

```
namespace SeedAPI.Web.API.App_Start

{

    public class JwtTokenConfig

    {

        public static void AddAuthentication(IServiceCollection services, IConfiguration configuration)

        {

            services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

            .AddJwtBearer(options =>

            {

                options.TokenValidationParameters = new TokenValidationParameters

                {

                    ValidateIssuer = true,

                    ValidateAudience = true,

                    ValidateLifetime = true,
```

```
                    ValidateIssuerSigningKey = true,
                    ValidIssuer = configuration["Jwt:Issuer"],
                    ValidAudience = configuration["Jwt:Issuer"],
                    IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["Jwt:Key"]))
                };
                services.AddCors();
            });
        }
    }
}
```

Los valores de los parámetros de validación dependerán de los requisitos de cada proyecto. El usuario y audiencia válidos que podemos establecer leyendo el archivo de configuración `appsettings.json`:

```
"Jwt": {
    "Key": "veryVerySecretKey",
    "Issuer": "http://localhost:50498/"
  }
```

Entonces solo necesitamos llamarlo desde el `ConfigureServices` método en `startup.cs`:

```
  // This method gets called by the runtime. Use this method to add services to the
container.
        public void ConfigureServices(IServiceCollection services)
```

```
        {
            DependencyInjectionConfig.AddScope(services);
            JwtTokenConfig.AddAuthentication(services, Configuration);
            DBContextConfig.Initialize(services, Configuration);
            services.AddMvc();
        }
```

Ahora estamos listos para crear nuestro primer controlador llamado `TokenController.cs`. El valor fijamos en `appsettings.json` que `"veryVerySecretKey"` debe coincidir con el que usamos para crear el token, pero en primer lugar, vamos a crear el `LoginViewModel` interior de nuestro `ViewModels` proyecto:

```
namespace SeedAPI.ViewModels
{
    public class LoginViewModel : IBaseViewModel
    {
        public string username { get; set; }
        public string password { get; set; }
    }
}
```

Y finalmente el controlador:

```
namespace SeedAPI.Web.API.Controllers
{
```

```
    [Route("api/Token")]
    public class TokenController : Controller
    {
        private IConfiguration _config;
        public TokenController(IConfiguration config)
        {
            _config = config;
        }
        [AllowAnonymous]
        [HttpPost]
        public dynamic Post([FromBody]LoginViewModel login)
        {
            IActionResult response = Unauthorized();
            var user = Authenticate(login);
            if (user != null)
            {
                var tokenString = BuildToken(user);
                response = Ok(new { token = tokenString });
            }
            return response;
        }
```

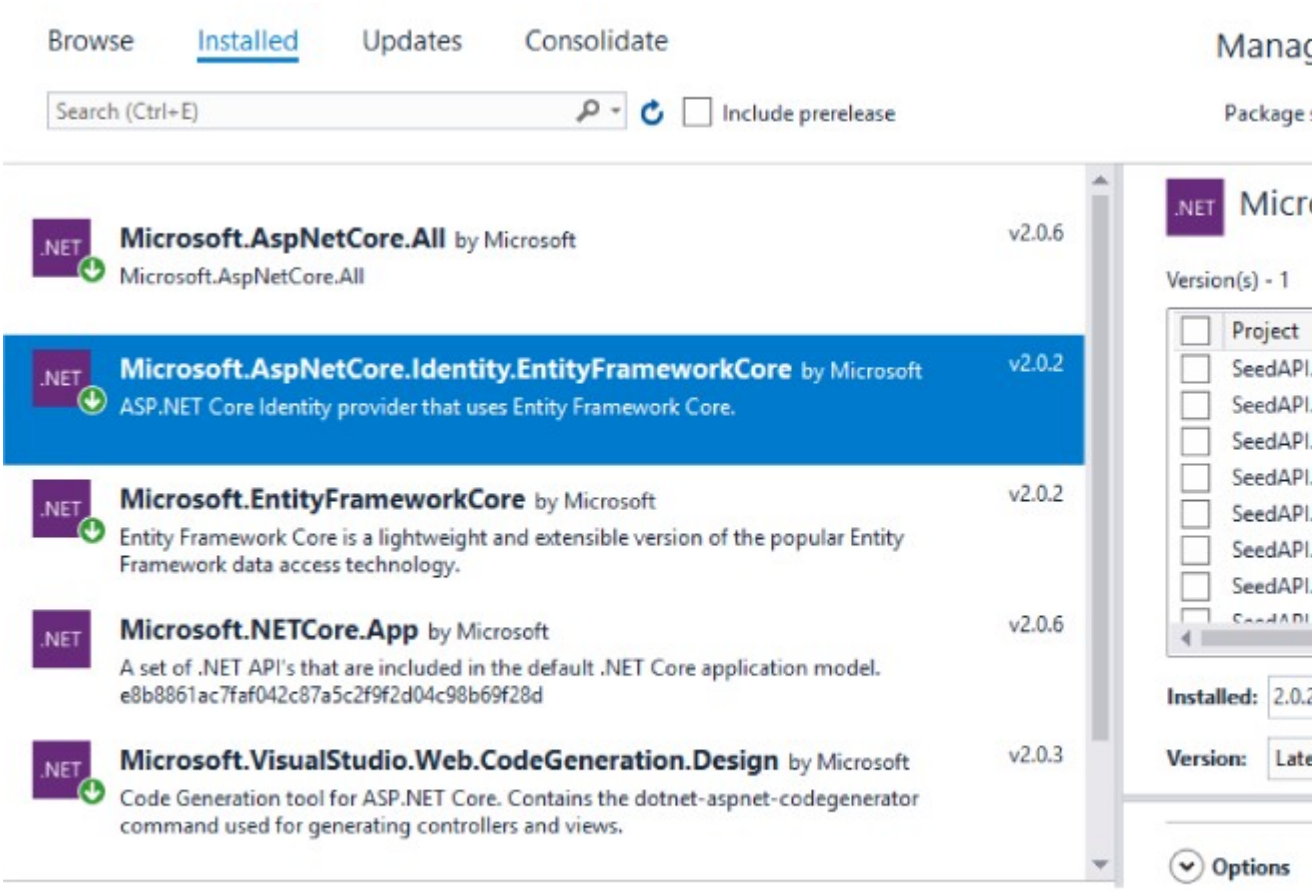
```
        private string BuildToken(UserViewModel user)
        {
            var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
            var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
            var token = new JwtSecurityToken(_config["Jwt:Issuer"],
              _config["Jwt:Issuer"],
              expires: DateTime.Now.AddMinutes(30),
              signingCredentials: creds);
            return new JwtSecurityTokenHandler().WriteToken(token);
        }
        private UserViewModel Authenticate(LoginViewModel login)
        {
            UserViewModel user = null;
            if (login.username == "pablo" && login.password == "secret")
            {
                user = new UserViewModel { name = "Pablo" };
            }
            return user;
        }
    }
```

```
}
```

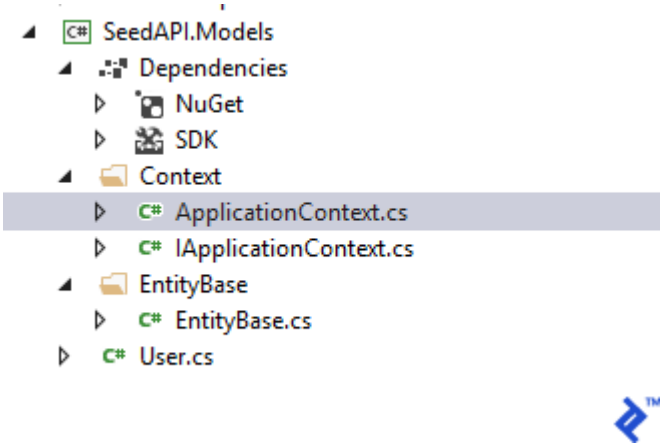
El `BuildToken` método creará el token con el código de seguridad dado. El `Authenticate` método solo tiene una validación de usuario codificada por el momento, pero al final tendremos que llamar a la base de datos para validarla.

El contexto de la aplicación

Configurar Entity Framework es realmente fácil desde que Microsoft lanzó la versión Core 2.0: _EF Core 2_ para abreviar. Vamos a profundizar con un modelo de código primero `identityDbContext`, así que primero asegúrese de haber instalado todas las dependencias. Puede usar NuGet para administrarlo:



Usando el `Models` proyecto podemos crear aquí dentro de la `Context` carpeta dos archivos, `ApplicationContext.cs` y `IApplicationContext.cs`. Además, necesitaremos una `EntityBase` clase.



Los `EntityBase` archivos serán heredados por cada modelo de entidad, pero `User.cs` es una clase de identidad y la única entidad que heredará `IdentityUser`. A continuación se encuentran ambas clases:

```
namespace SeedAPI.Models

{

    public class User : IdentityUser

    {

        public string Name { get; set; }

    }

}

namespace SeedAPI.Models.EntityBase

{

    public class EntityBase
```

```
    {
        public DateTime? Created { get; set; }
        public DateTime? Updated { get; set; }
        public bool Deleted { get; set; }
        public EntityBase()
        {
            Deleted = false;
        }
        public virtual int IdentityID()
        {
            return 0;
        }
        public virtual object[] IdentityID(bool dummy = true)
        {
            return new List<object>().ToArray();
        }
    }
}
```

Ahora estamos listos para crear `ApplicationContext.cs`, que se verá así:

```
namespace SeedAPI.Models.Context
{
```

```
    public class ApplicationContext : IdentityDbContext<User>,
IApplicationContext
    {
        private IDbContextTransaction dbContextTransaction;
        public ApplicationContext(DbContextOptions options)
            : base(options)
        {
                    }

        public DbSet<User> UsersDB { get; set; }
        public new void SaveChanges()
        {
            base.SaveChanges();
        }
        public new DbSet<T> Set<T>() where T : class
        {
            return base.Set<T>();
        }
        public void BeginTransaction()
        {
            dbContextTransaction = Database.BeginTransaction();
        }
```

```
        public void CommitTransaction()
        {
            if (dbContextTransaction != null)
            {
                dbContextTransaction.Commit();
            }
        }
        public void RollbackTransaction()
        {
            if (dbContextTransaction != null)
            {
                dbContextTransaction.Rollback();
            }
        }
        public void DisposeTransaction()
        {
            if (dbContextTransaction != null)
            {
                dbContextTransaction.Dispose();
            }
        }
```

```
    }

}
```

Estamos muy cerca, pero primero necesitaremos crear más clases, esta vez en la `App_Start` carpeta ubicada en el `Web.API` proyecto. La primera clase es inicializar el contexto de la aplicación y la segunda es crear datos de muestra solo con el propósito de probar durante el desarrollo.

```
namespace SeedAPI.Web.API.App_Start

{

    public class DBContextConfig

    {

        public static void Initialize(IConfiguration configuration, IHostingEnvironment env, IServiceProvider svp)

        {

            var optionsBuilder = new DbContextOptionsBuilder();

            if (env.IsDevelopment())
optionsBuilder.UseSqlServer(configuration.GetConnectionString("DefaultConnection"));

            else if (env.IsStaging())
optionsBuilder.UseSqlServer(configuration.GetConnectionString("DefaultConnection"));

            else if (env.IsProduction())
optionsBuilder.UseSqlServer(configuration.GetConnectionString("DefaultConnection"));

            var context = new ApplicationContext(optionsBuilder.Options);

            if (context.Database.EnsureCreated())

            {
```

```
                IUserMap service = svp.GetService( typeof (IUserMap)) as
IUserMap;

                new DBInitializeConfig(service).DataTest();

            }

        }

        public static void Initialize(IServiceCollection services, IConfiguration
configuration)

        {

            services.AddDbContext<ApplicationContext>(options =>

                options.UseSqlServer(configuration.GetConnectionString( "DefaultConnection" )));

        }

    }

}

namespace SeedAPI.Web.API.App_Start

{

    public class DBInitializeConfig

    {

        private IUserMap userMap;

        public DBInitializeConfig (IUserMap _userMap)

        {

            userMap = _userMap;
```

```
        }
        public void DataTest()
        {
            Users();
        }
        private void Users()
        {
            userMap.Create( new  UserViewModel() { id =  1 , name =  "Pablo"  });
            userMap.Create( new  UserViewModel() { id =  2 , name =  "Diego"  });
        }
    }
}
```

Y los llamamos desde nuestro archivo de inicio:

```
    // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            DependencyInjectionConfig.AddScope(services);
            JwtTokenConfig.AddAuthentication(services, Configuration);
            DBContextConfig.Initialize(services, Configuration);
            services.AddMvc();
```

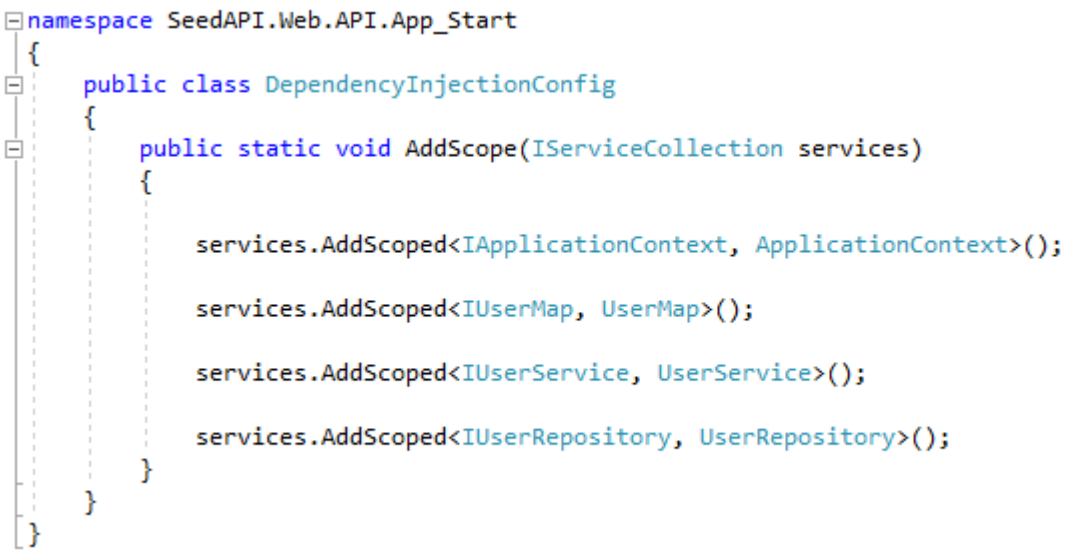
```
        }

// ...

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

        public void Configure(IApplicationBuilder app, IHostingEnvironment env, IServiceProvider svp)

        {

            if (env.IsDevelopment())

            {

                app.UseDeveloperExceptionPage();

            }

            DBContextConfig.Initialize(Configuration, env, svp);

            app.UseCors(builder => builder

                .AllowAnyOrigin()

                .AllowAnyMethod()

                .AllowAnyHeader()

                .AllowCredentials());

            app.UseAuthentication();

            app.UseMvc();

        }
```

Inyección de dependencia

Es una buena práctica utilizar la inyección de dependencia para moverse entre diferentes proyectos. Esto nos ayudará a comunicarnos entre controladores y mapeadores, mapeadores y servicios, y servicios y repositorios.

Dentro de la carpeta `App_Start` crearemos el archivo `DependencyInjectionConfig.cs` y se verá así:

```
namespace SeedAPI.Web.API.App_Start
{
    public class DependencyInjectionConfig
    {
        public static void AddScope(IServiceCollection services)
        {
            services.AddScoped<IApplicationContext, ApplicationContext>();
            services.AddScoped<IUserMap, UserMap>();
            services.AddScoped<IUserService, UserService>();
            services.AddScoped<IUserRepository, UserRepository>();
        }
    }
}
```



```
namespace SeedAPI.Web.API.App_Start
{
    public class DependencyInjectionConfig
    {
        public static void AddScope(IServiceCollection services)
        {

            services.AddScoped<IApplicationContext, ApplicationContext>();

            services.AddScoped<IUserMap, UserMap>();

            services.AddScoped<IUserService, UserService>();

            services.AddScoped<IUserRepository, UserRepository>();
        }
    }
}
```

Tendremos que crear para cada nueva entidad una nueva `Map`, `Service` y `Repository`, y les partido a este archivo. Entonces solo necesitamos llamarlo desde el `startup.cs` archivo:

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    DependencyInjectionConfig.AddScope(services);
    JwtTokenConfig.AddAuthentication(services, Configuration);
    DBContextConfig.Initialize(services, Configuration);
    services.AddMvc();
}
```

Finalmente, cuando necesitamos obtener la lista de usuarios de la base de datos, podemos crear un controlador usando esta inyección de dependencia:

```
namespace SeedAPI.Web.API.Controllers
{
    [Route("api/[controller]")]
    [Authorize]
    public class UserController : Controller
    {
        IUserMap userMap;
        public UserController(IUserMap map)
        {
            userMap = map;
        }
        // GET api/user
        [HttpGet]
        public IEnumerable<UserViewModel> Get()
        {
            return userMap.GetAll(); ;
        }
        // GET api/user/5
```

```
        [HttpGet("{id}")]
        public string Get(int id)
        {
            return "value";
        }
        // POST api/user
        [HttpPost]
        public void Post([FromBody]string user)
        {
        }
        // PUT api/user/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody]string user)
        {
        }
        // DELETE api/user/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
```

```
}
```

Mire cómo `Authorize` está presente el atributo aquí para asegurarse de que el front-end ha iniciado sesión y cómo funciona la inyección de dependencia en el constructor de la clase.
Finalmente tenemos una llamada a la base de datos, pero primero, necesitamos entender el `Map` proyecto.

El `Maps` proyecto

Este paso es solo para mapear `ViewModels` ay desde modelos de bases de datos. Debemos crear uno para cada entidad y, siguiendo nuestro ejemplo anterior, el `UserMap.cs` archivo se verá así:

```
namespace SeedAPI.Maps

{

    public class UserMap : IUserMap

    {

        IUserService userService;

        public UserMap(IUserService service)

        {

            userService = service;

        }

        public UserViewModel Create(UserViewModel viewModel)

        {

            User user = ViewModelToDomain(viewModel);

            return DomainToViewModel(userService.Create(user));

        }

        public bool Update(UserViewModel viewModel)
```

```
        {
            User user = ViewModelToDomain(viewModel);
            return userService.Update(user);
        }
        public bool Delete(int id)
        {
            return userService.Delete(id);
        }
        public List<UserViewModel> GetAll()
        {
            return DomainToViewModel(userService.GetAll());
        }
        public UserViewModel DomainToViewModel(User domain)
        {
            UserViewModel model = new UserViewModel();
            model.name = domain.Name;
            return model;
        }
        public List<UserViewModel> DomainToViewModel(List<User> domain)
        {
            List<UserViewModel> model = new List<UserViewModel>();
```

```
            foreach (User of in domain)
            {
                model.Add(DomainToViewModel(of));
            }
            return model;
        }
        public User ViewModelToDomain(UserViewModel officeViewModel)
        {
            User domain = new User();
            domain.Name = officeViewModel.name;
            return domain;
        }
    }
}
```

Parece que una vez más, la inyección de dependencias está funcionando en el constructor de la clase, vinculando Maps al proyecto de Servicios.

El `Services` proyecto

No hay mucho que decir aquí: nuestro ejemplo es realmente simple y no tenemos lógica o código comercial para escribir aquí. Este proyecto sería útil en futuros requisitos avanzados cuando necesitemos calcular o hacer algo de lógica antes o después de los

pasos de la base de datos o del controlador. Siguiendo el ejemplo, la clase se verá bastante desnuda:

```
namespace SeedAPI.Services
{
    public class UserService : IUserService
    {
        private IUserRepository repository;
        public UserService(IUserRepository userRepository)
        {
            repository = userRepository;
        }
        public User Create(User domain)
        {
            return repository.Save(domain);
        }
        public bool Update(User domain)
        {
            return repository.Update(domain);
        }
        public bool Delete(int id)
        {
```

```
            return repository.Delete(id);
        }
        public List<User> GetAll()
        {
            return repository.GetAll();
        }
    }
}
```

El `Repositories` proyecto

Estamos llegando a la última sección de este tutorial: solo necesitamos hacer llamadas a la base de datos, por lo que creamos un `UserRepository.cs` archivo donde podemos leer, insertar o actualizar usuarios en la base de datos.

```
namespace SeedAPI.Repositories
{
    public class UserRepository : BaseRepository, IUserRepository
    {
        public UserRepository(IApplicationContext context)
            : base(context)
        { }
        public User Save(User domain)
        {
            try
```

```
        {
            var us = InsertUser<User>(domain);
            return us;
        }
        catch (Exception ex)
        {
            //ErrorManager.ErrorHandler.HandleError(ex);
            throw ex;
        }
    }
    public bool Update(User domain)
    {
        try
        {
            //domain.Updated = DateTime.Now;
            UpdateUser<User>(domain);
            return true;
        }
        catch (Exception ex)
        {
            //ErrorManager.ErrorHandler.HandleError(ex);
```

```
                throw ex;
            }
        }
        public bool Delete(int id)
        {
            try
            {
                User user = Context.UsersDB.Where(x =>
x.Id.Equals(id)).FirstOrDefault();

                if (user != null)
                {
                    //Delete<User>(user);
                    return true;
                }
                else
                {
                    return false;
                }
            }
            catch (Exception ex)
```

```
            {
                //ErrorManager.ErrorHandler.HandleError(ex);
                throw ex;
            }
        }
        public List<User> GetAll()
        {
            try
            {
                return Context.UsersDB.OrderBy(x => x.Name).ToList();
            }
            catch (Exception ex)
            {
                //ErrorManager.ErrorHandler.HandleError(ex);
                throw ex;
            }
        }
    }
}
```