



# Reto 4: Landing The Power Avanzada

## Descripción del reto

### Parte 1: Análisis del proyecto y añadiendo mejoras

Home

Listado de cursos

**Detalle de curso**

Incorporando una API

Variables de entorno

Añadiendo el HttpClient

Guardia para el formulario inscripción

### Parte 2: Añadiendo NgRx

Instalando librería

Creando estado, acciones y reducers

Creando efectos y selectores

Aplicando NgRx en las vistas

Redux Devtools

### Parte 3: Añadiendo SSR y SSG

Análisis

Añadiendo SSR

Configurando prerenderizado y SSR

Platform service

Cacheo de peticiones

Rehidratación del store de NgRx

## Descripción del reto

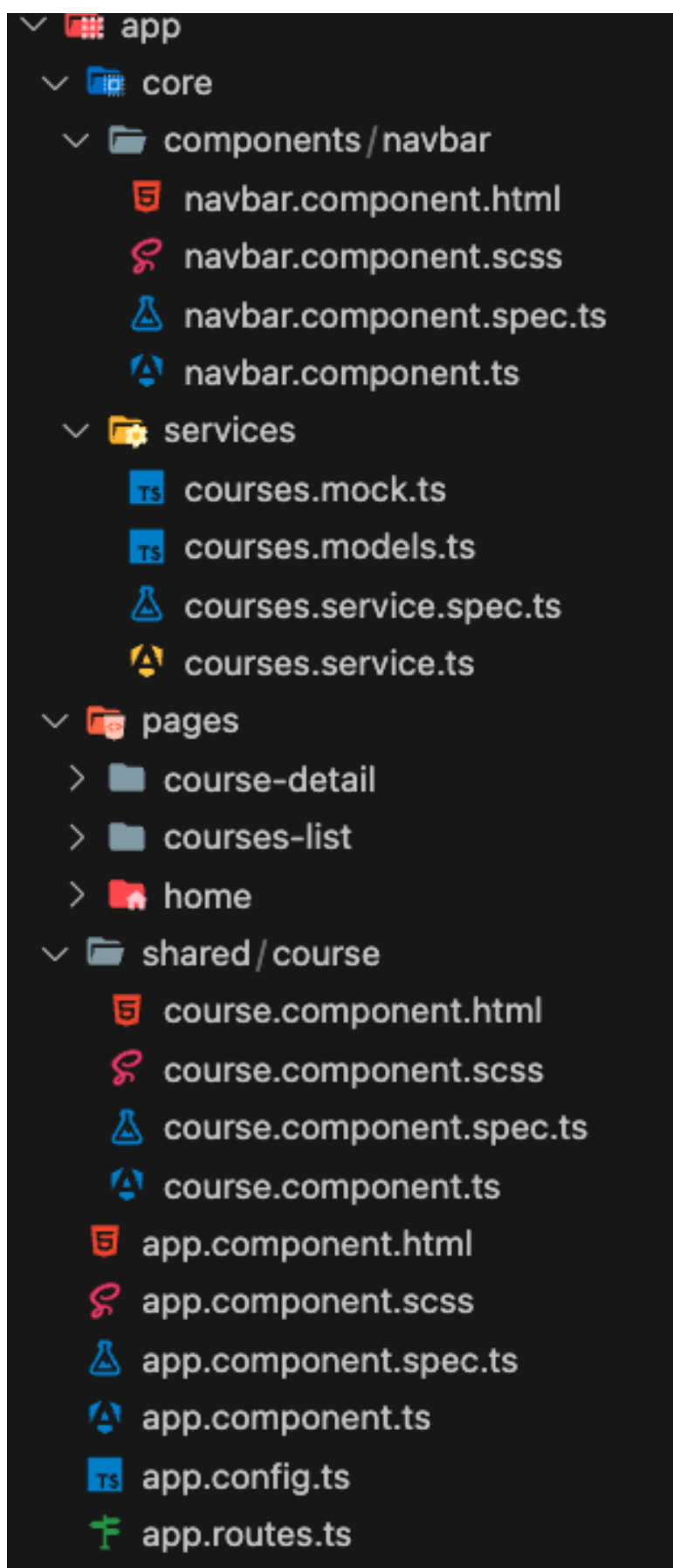
Durante este reto mejoraremos una landing de The Power que recibiremos como base, iremos aplicando todos los conceptos aprendidos durante el bloque. Añadiremos guardias, conectaremos todo con NgRx y añadiremos SSR y SSG al proyecto en aquellas páginas donde consideremos que puede mejorar el rendimiento.

En la landing los usuarios podrán navegar entre distintas páginas y acceder al catálogo de cursos disponibles. Además podrán inscribirse a cualquiera de los cursos haciendo que la información sobre el número de plazas disponibles se actualice.

## Parte 1: Análisis del proyecto y añadiendo mejoras

Vamos a comenzar haciendo un análisis de la aplicación que tenemos como base, iremos repasando las distintas vistas y los componentes utilizados en ellas.

Primero vamos a ver el aspecto general de la arquitectura de nuestro proyecto:



- Tenemos 3 páginas: la home, listado de cursos y el detalle de curso, donde el usuario podrá rellenar el formulario de inscripción.
- Dentro de nuestra carpeta **core** tenemos el componente navbar que se muestra en todas las páginas y el servicio **courses** que se utiliza para recuperar la información de los cursos mostrados en la web.
- Dentro de **shared** tenemos un único componente para mostrar la información de un curso.

## Home

En primer lugar tenemos la **home** de nuestra web. En ella se muestra un vídeo, el eslogan de la empresa y un listado de aquellos cursos que están considerados los más populares



Más que una universidad

## “La escuela que lidera la revolución en la educación”

### Cursos más populares



#### Máster ThePowerMBA

El MBA online que te ayuda a entender el negocio como los fundadores de Waze, YouTube, Tesla...

100% Online 80 horas

695€ 20 plazas

Apuntarme ya



#### Máster en Ventas

El primer máster de ventas en el que aprendes técnicas de negociación, comunicación de la mano del Ex Negociador del FBI.

100% Online 23 horas

695€ 20 plazas

Apuntarme ya

Como necesitamos pintar cursos también en la página de listado de cursos necesitaremos un componente reutilizable que vayamos importando en las páginas que lo requieran. En nuestro caso es el **course.component.ts**:

```
import { CommonModule, NgOptimizedImage } from '@angular/common';
import { Component, Input } from '@angular/core';
import { Course } from '../../../core/services/courses.models';
```

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

@Component({
  selector: 'app-course',
  standalone: true,
  imports: [
    CommonModule,
    NgOptimizedImage
  ],
  templateUrl: './course.component.html',
  styleUrls: ['./course.component.scss']
})
export class CourseComponent {
  @Input() course?: Course;

  constructor(private router: Router) {}

  public onSeeMoreInfo(): void {
    if (!this.course) { return; }
    this.router.navigate(['/course', this.course.id])
  }
}
```

Recibirá el curso que nos interesa mostrar como input, además tendrá un método disponible para navegar al detalle del curso si se pulsa sobre el botón de “Apuntarme ya”.

Para recuperar 4 cursos más populares necesitaremos un método especial en nuestro **courses.service.ts**.

```
import { Injectable } from '@angular/core';
import { Course } from '../courses.models';
import { COURSES } from '../courses.mock';

@Injectable({
  providedIn: 'root'
})
export class CoursesService {

  private courses: Course[] = COURSES;

  public getCourses(): Course[] {
    return this.courses;
  }

  public getTopCourses(): Course[] {
    return this.courses.slice(0, 4);
  }

  public getCourseById(id: string): Course | undefined {
    return this.courses.find(course => course.id === id);
  }

  public enrolleStudent(courseId: string): void {
    const course = this.courses.find(course => course.id === courseId);
    if (course) {
      course.numVacancies--;
    }
  }
}
```

```
}  
}
```

En este caso será **getTopCourses** que recupera los 4 primeros cursos del listado. También tenemos métodos para recuperar todos los cursos o un curso por su id.

El método `enroleStudent` permite disminuir el número de plazas disponibles en un curso cuando un alumno se inscribe.

## Listado de cursos

La página de listado de cursos es muy simple, se limita a mostrar el listado de todos los cursos utilizando el mismo componente **course** que la home. Recupera los cursos del método del servicio correspondiente:

```
import { CommonModule } from '@angular/common';  
import { Component, OnInit } from '@angular/core';  
import { CourseComponent } from '../../shared/course/course.component';  
import { Course } from '../../core/services/courses.models';  
import { CoursesService } from '../../core/services/courses.service';  
  
@Component({  
  selector: 'app-courses-list',  
  standalone: true,  
  imports: [  
    CommonModule,  
    CourseComponent  
  ],  
  templateUrl: './courses-list.component.html',  
  styleUrls: ['./courses-list.component.scss']  
})  
export class CoursesListComponent implements OnInit {  
  
  public courses: Course[] = [];  
  
  constructor(private coursesService: CoursesService) {}  
  
  public ngOnInit() {  
    this.courses = this.coursesService.getCourses();  
  }  
  
}
```





### Máster ThePowerMBA

El MBA online que te ayuda a entender el negocio como los fundadores de Waze, YouTube, Tesla...

100% Online 80 horas

695€ 20 plazas

[Apuntarme ya](#)



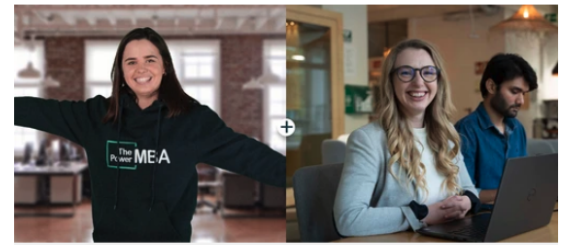
### Máster en Ventas

El primer máster de ventas en el que aprendes técnicas de negociación, comunicación de la mano del Ex Negociador del FBI.

100% Online 23 horas

695€ 20 plazas

[Apuntarme ya](#)



### ThePowerMBA + Ventas

Aprende de negocio de los fundadores de empresas como Netflix, y técnicas de negociación del Ex Negociador del FBI.

100% Online 103 horas

950€ 30 plazas

[Apuntarme ya](#)

## Detalle de curso

En la página de detalle de curso el usuario podrá completar un formulario para inscribirse:

### Rellena el formulario de inscripción

Curso

ThePower Digital Marketing

Nombre

Escribe tu nombre

Apellidos

Tus apellidos completos

Email

Escribe tu email

Teléfono

Dirección

Escribe tu dirección

☐ He leído y acepto los términos y condiciones y la política de privacidad

### Resumen de inscripción

#### ThePower Digital Marketing

El máster que te enseñará los conceptos del marketing digital para ver el mundo desde una perspectiva completamente nueva.

Total: 695€

El formulario será reactivo y incluirá los diferentes campos con sus validaciones específicas. El campo de “Curso” se rellenará con el título del curso que se obtendrá utilizando el **id** que llega como parámetro en la ruta.

```
public initializeEnrolmentForm(): void {
  this.enrolmentForm = new FormGroup({
    course: new FormControl(this.course?.title || '', Validators.required),
    name: new FormControl('', Validators.required),
    surname: new FormControl('', Validators.required),
    email: new FormControl('', [Validators.required, Validators.email]),
    phone: new FormControl('', [Validators.pattern('[0-9]{9}')] ),
  })
}
```

```

        address: new FormControl('', [Validators.required, validateNumberInAddress()]),
        terms: new FormControl(false, Validators.requiredTrue),
    });
}

```

Cuando se rellene el formulario con un valor válido se utilizará el método **enroleStudent** del servicio para inscribirlo, se reseteará el formulario y se navegará al listado de cursos:

```

public enroleStudent(): void {
    if (!this.enrolementForm || !this.enrolementForm.valid || !this.course) {
        return;
    }
    this.coursesService.enroleStudent(this.course.id);
    this.enrolementForm.reset();
    this.router.navigate(['/courses']);
}

```

¡Con eso hemos repasado todo lo que tenemos hasta ahora en el proyecto! Toca empezar a añadir mejoras.

## Incorporando una API

Nuestro servicio de cursos ahora mismo consume la información de unos datos mockeados:

```

import { Course } from "../courses.models";

export const COURSES: Course[] = [
    {
        id: '1',
        title: 'Máster ThePowerMBA',
        description: 'El MBA online que te ayuda a entender el negocio como los fundadores de Waze',
        image: 'https://framerusercontent.com/images/qTWHKIBpIzeAGFWug0roara5dtM.jpg?scale-down-to=512x512',
        price: 695,
        numHours: 80,
        isOnline: true,
        numVacancies: 20
    },
    {
        id: '2',
        title: 'Máster en Ventas',
        description: 'El primer máster de ventas en el que aprendes técnicas de negociación, comunicación y liderazgo',
        image: 'https://framerusercontent.com/images/IW1khi8heizJNm98ma308Et6Tp0.png?scale-down-to=512x512',
        isOnline: true,
        price: 695,
        numHours: 23,
        numVacancies: 20
    },
    {
        id: '3',
        title: 'ThePowerMBA + Ventas',
        description: 'Aprende de negocio de los fundadores de empresas como Netflix, y técnicas de negociación y liderazgo',
        image: 'https://framerusercontent.com/images/Lie5Q5oJuIBj34NwPl6Q3ZjT6qM.png?scale-down-to=512x512',
        isOnline: true,
        price: 950,
        numHours: 103,
        numVacancies: 30
    },
]

```

```

{
  id: '4',
  title: 'Máster en Consulting',
  description: 'Conviértete en consultor en 6 meses y accede a las firmas globales más prestigiosas',
  image: 'https://framerusercontent.com/images/tsfC49f2eYGXpoLlhRqj9500.jpg?scale-down-to=512',
  isOnline: true,
  price: 2499,
  numHours: 1145,
  numVacancies: 10
},
{
  id: '5',
  title: 'ThePower Digital Marketing',
  description: 'El máster que te enseñará los conceptos del marketing digital para ver el mundo desde una perspectiva diferente',
  image: 'https://framerusercontent.com/images/a7YapjJwdM5zBpyID4EYTgfjUc.jpg?scale-down-to=512',
  isOnline: true,
  price: 695,
  numHours: 95,
  numVacancies: 50
},
{
  id: '6',
  title: 'Máster en Inteligencia Artificial ',
  description: 'Aprende a utilizar las herramientas de Inteligencia Artificial sin miedo y con confianza',
  image: 'https://framerusercontent.com/images/saF6PFAmBzVMBdaJQ4PboNKMs.jpg?scale-down-to=512',
  isOnline: true,
  price: 1149,
  numHours: 200,
  numVacancies: 60
},
{
  id: '7',
  title: 'UX-UI Design',
  description: 'Aprende a diseñar productos digitales en 16 semanas con Figma con metodologías ágiles',
  image: 'https://framerusercontent.com/images/slf3rSb0KUMKtQqbj334KBZ4FwQ.jpg?scale-down-to=512',
  isOnline: true,
  price: 4200,
  numHours: 340,
  numVacancies: 70
},
{
  id: '8',
  title: 'Bootcamp de Desarrollo Full Stack',
  description: 'El bootcamp con el que te convertirás en uno de los perfiles más valiosos, escogiendo tu especialidad',
  image: 'https://framerusercontent.com/images/8ttCDKI5QMi9KYW2vGmTKmiG0.png?scale-down-to=512',
  isOnline: false,
  price: 4200,
  numHours: 400,
  numVacancies: 80
},
{
  id: '9',
  title: 'Angular',
  description: 'Angular is a platform that makes it easy to build applications with the web.',
  image: 'https://angular.io/assets/images/logos/angular/angular.svg',
  price: 100,
  numHours: 40,
  isOnline: true,

```



```

    numVacancies: 5
  },
  {
    id: '10',
    title: 'React',
    description: 'React is a JavaScript library for building user interfaces.',
    image: 'https://upload.wikimedia.org/wikipedia/commons/thumb/a/a7/React-icon.svg/1200px-React-icon.svg',
    price: 50,
    numHours: 20,
    isOnline: true,
    numVacancies: 10
  },
  {
    id: '11',
    title: 'Vue',
    description: 'Vue is a progressive framework for building user interfaces.',
    image: 'https://vuejs.org/images/logo.png',
    price: 75,
    numHours: 30,
    isOnline: false,
    numVacancies: 3
  },
  {
    id: '12',
    title: 'Svelte',
    description: 'Svelte is a radical new approach to building user interfaces.',
    image: 'https://upload.wikimedia.org/wikipedia/commons/thumb/1/1b/Svelte_Logo.svg/1200px-Svelte_Logo.svg',
    price: 25,
    numHours: 10,
    isOnline: true,
    numVacancies: 8
  },
  {
    id: '13',
    title: 'Ember',
    description: 'Ember.js is an open-source JavaScript web framework.',
    image: 'https://emberjs.com/images/tomster-twitter-card.png',
    price: 125,
    numHours: 50,
    isOnline: false,
    numVacancies: 1
  },
  {
    id: '14',
    title: 'Meteor',
    description: 'Meteor is an open-source platform for web, mobile, and desktop.',
    image: 'https://static.vecteezy.com/system/resources/thumbnails/008/257/657/small/meteor-icon.png',
    price: 150,
    numHours: 60,
    isOnline: true,
    numVacancies: 30
  },
  {
    id: '15',
    title: 'Node',
    description: 'Node.js is an open-source, cross-platform, JavaScript runtime environment.',
    image: 'https://upload.wikimedia.org/wikipedia/commons/thumb/d/d9/Node.js_logo.svg/1200px-Node.js_logo.svg',
    price: 200,

```

```

    numHours: 80,
    isOnline: true,
    numVacancies: 20
  },
  {
    id: '16',
    title: 'Deno',
    description: 'Deno is a simple, modern and secure runtime for JavaScript and TypeScript.',
    image: 'https://upload.wikimedia.org/wikipedia/commons/thumb/e/e8/Deno_2021.svg/512px-Deno_2021.svg',
    price: 175,
    numHours: 70,
    isOnline: false,
    numVacancies: 15
  },
  {
    id: '17',
    title: 'Nest',
    description: 'Nest is a framework for building efficient, scalable Node.js server-side applications.',
    image: 'https://nestjs.com/img/logo_text.svg',
    price: 225,
    numHours: 90,
    isOnline: true,
    numVacancies: 25
  },
  {
    id: '18',
    title: 'Express',
    description: 'Express is a minimal and flexible Node.js web application framework.',
    image: 'https://upload.wikimedia.org/wikipedia/commons/6/64/Expressjs.png',
    price: 250,
    numHours: 100,
    isOnline: false,
    numVacancies: 2
  }
];

```

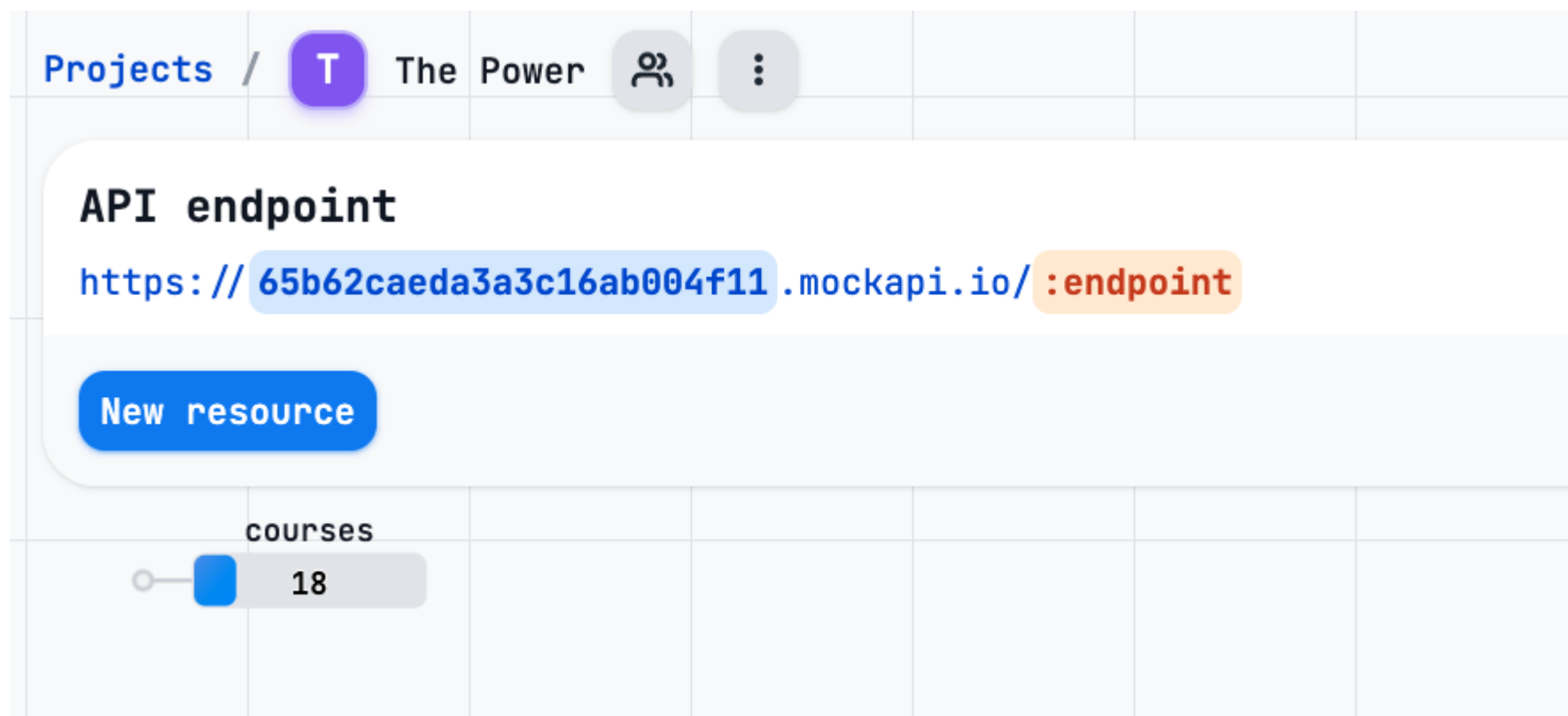
Vamos a coger estos datos y vamos a añadirlos a una API utilizando **MockApi**, de esa forma comenzaremos a recuperar los datos con el **HttpClient** y tendremos un proyecto más real.

Otra gran ventaja que nos da añadir la información a esta API es que podremos persistir los datos y utilizar un **PUT** para modificar el número de alumnos inscritos al curso.

Para ello vamos a copiar todos los datos y generamos un nuevo endpoint **courses** dentro de nuestra nueva MockApi.

Si queréis clonar la API que hemos generado para poder modificarla a vuestro antojo podéis utilizar el siguiente enlace:

<https://mockapi.io/clone/65b62caeda3a3c16ab004f12>



Ahora tenemos la ventaja de que podremos hacer llamadas **PUT** para modificar nuestros cursos y disminuir el número de plazas disponibles.

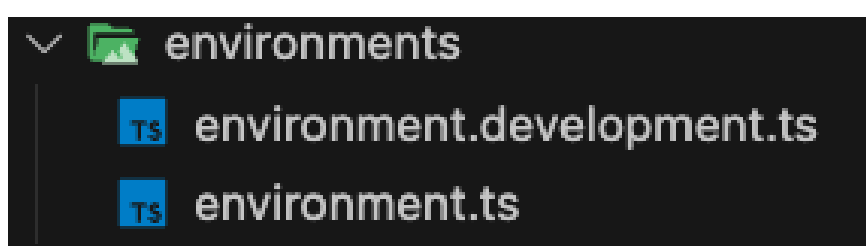
Vamos a hacer las adaptaciones en el código necesarias para utilizar nuestra nueva API:

## Variables de entorno

Primero necesitamos añadir una constante con la url de nuestra API de forma que hagamos ese dato configurable y accesible. A futuro podemos necesitar cambiar la url en función del entorno, por lo tanto sería recomendable añadir esta api url al archivo de variables de entorno.

Para ello desde la CLI vamos a generar nuestros archivos de entorno:

```
ng generate environments
```



Y añadimos nuestra variable con la url en ambos archivos `environment.ts` y `environment.development.ts`

```
export const environment = {  
  apiUrl: 'https://65b62caeda3a3c16ab004f11.mockapi.io/',  
};
```



No hemos añadido un archivo de entorno para producción porque por ahora no es necesario. Si en el futuro se requiriera podríamos crear el archivo `environment.production.ts` manualmente.

## Añadiendo el HttpClient

Es el momento de añadir el **HttpClient** a nuestra aplicación ya que lo vamos a necesitar para hacer las llamadas correspondientes a la API. Para empezar lo añadimos como provider en nuestro `app.config.ts`:

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter, withInMemoryScrolling } from '@angular/router';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withInMemoryScrolling({ scrollPositionRestoration: 'top' })),
    provideHttpClient()
  ]
};
```

En el interior del `courses.service.ts` lo añadimos al constructor y modificamos los métodos para que hagan directamente la llamada a la API y devuelvan un observable.

```
import { Injectable } from '@angular/core';
import { Course } from './courses.models';
import { COURSES } from './courses.mock';
import { HttpClient } from '@angular/common/http';
import { Observable, map } from 'rxjs';
import { environment } from '../../environments/environment.development';

@Injectable({
  providedIn: 'root'
})
export class CoursesService {

  constructor(private http: HttpClient) {}

  public getCourses(): Observable<Course[]> {
    return this.http.get<Course[]>(`${environment.apiUrl}courses`);
  }

  public getTopCourses(): Observable<Course[]> {
    return this.http.get<Course[]>(`${environment.apiUrl}courses`).pipe(
      map(courses => courses.slice(0, 4))
    );
  }

  public getCourseById(id: string): Observable<Course> {
    return this.http.get<Course>(`${environment.apiUrl}courses/${id}`);
  }

  public enroleStudent(
    courseId: string,
    newNumVacancies: number
  ): Observable<Course> {
    return this.http.put<Course>(`${environment.apiUrl}courses/${courseId}`, {
      numVacancies: newNumVacancies
    });
  }
}
```

Ahora ya no es necesario almacenar los cursos en una variable dentro del servicio ya que queremos utilizar los métodos para recuperar los datos directamente de la API.

- El método `getTopCourses` recupera los 4 primeros cursos de la API, para ello tras haber recuperado todos los datos utiliza el operador `map` de RxJs para quedarse solo con esos cursos que nos interesan. A futuro cuando añadamos NgRx sustituiremos este método por un selector que tenga esta lógica integrada y que nos permita recuperar los cursos solo al inicio de la aplicación y cuando algún dato se haya modificado.
- `enroleStudent` recibe el id del curso a modificar y el nuevo número de plazas que pasa a tener el curso disponible tras la inscripción. Hacemos un **PUT** y pasamos como body de la petición el objeto con el número de plazas actualizado.



No hemos añadido operadores para transformar los datos porque toda la información que actualmente recuperamos de la API se utiliza en la vista.

Estos cambios en el servicio suponen también adaptaciones en la vista, en todos los componentes tendremos que pasar a almacenar los datos en un observable y consumirlos desde la vista con la pipe `async`.

#### home.component.ts

```
export class HomeComponent implements OnInit {

  public topCourses$: Observable<Course[]>;

  constructor(private coursesServices: CoursesService) {}

  public ngOnInit() {
    this.topCourses$ = this.coursesServices.getTopCourses();
  }
}
```

#### home.component.html

```
<div class="home__courses">
  <app-course
    *ngFor="let course of (topCourses$ | async)"
    [course]="course"
  ></app-course>
</div>
```

#### courses-list.component.ts

```
export class CoursesListComponent implements OnInit {

  public courses$: Observable<Course[]>;

  constructor(private coursesService: CoursesService) {}

  public ngOnInit() {
    this.courses$ = this.coursesService.getCourses();
  }

}
```

#### courses-list.component.html

```
<div class="courses-list__container">
  <app-course
    *ngFor="let course of (courses$ | async)"
    [course]="course"
  ></app-course>
</div>
```



```
<</app-course>
</div>
```

Por último en el detalle de curso tendremos que esperar a que se hayan modificado los datos de inscripción del nuevo alumno antes de redirigir al listado de cursos, de forma que nos aseguremos que al recuperar la información de los cursos de la API ya está al día.

#### course-detail.component.ts

```
public listenRouteParamsChanges() {
  this.activatedRoute.params.subscribe((params) => {
    const courseId = params['id'];
    if (!courseId) { return; }
    this.coursesService.getCourseById(courseId).subscribe((course) => {
      this.course = course;
      this.initializeEnrolmentForm();
    });
  });
}

public enrolleStudent(): void {
  if (!this.enrolmentForm || !this.enrolmentForm.valid || !this.course) {
    return;
  }
  this.coursesService.enrolleStudent(
    this.course.id, this.course.numVacancies - 1
  ).subscribe(
    () => {
      this.enrolmentForm?.reset();
      this.router.navigate(['/courses']);
    }
  );
}
```

Ya tenemos nuestra aplicación consumiendo toda la información de la API. Ahora podemos empezar a trabajar sobre funcionalidades añadidas que mejoren la experiencia de usuario.

## Guardia para el formulario inscripción

Vamos a añadir una nueva guardia de tipo `CanDeactivate` a nuestra aplicación, la utilizaremos para confirmar que el usuario está seguro de que quiere dejar la página cuando tenga a medias el formulario de inscripción.

En primer lugar la generamos desde la CLI dentro de una carpeta **guards** que crearemos en el interior de la carpeta **core**:

```
cd src/app/core
mkdir guards
cd guards
ng generate guard exit-form
```

Seleccionaremos el tipo **CanDeactivate** y añadiremos la lógica necesaria a nuestro nuevo archivo **exit-form.guard.ts** generado:

```
import { CanDeactivateFn } from '@angular/router';
import { CourseDetailComponent } from '../pages/course-detail/course-detail.component';

export const exitFormGuard: CanDeactivateFn<CourseDetailComponent> = (component) => {
```

```

if (!component.enrolmentForm?.dirty) { return true; }
return window.confirm(
  'No has finalizado el proceso de matriculación. ¿Estás seguro de que quieres salir?'
);
};

```

Comprobamos dentro de nuestra vista de detalle de curso si el usuario ya ha interactuado con el formulario a través de su campo `dirty`. Si todavía no ha interactuado le dejamos salir, si no le preguntamos a través de una alerta si está seguro de que quiere dejar el proceso de matriculación en el curso a medias.

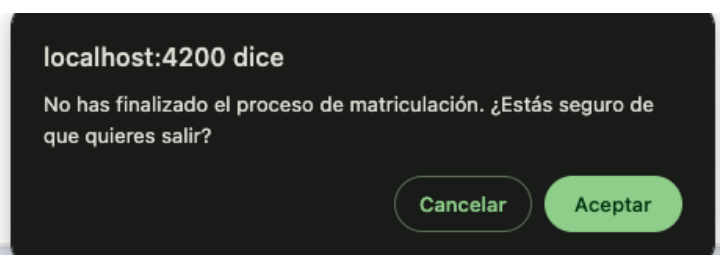
Ahora toca aplicar la nueva guardia a la ruta correspondiente en **app.routes.ts**:

```

{
  path: 'course/:id',
  loadComponent: () => import(
    './pages/course-detail/course-detail.component'
  ).then(c => c.CourseDetailComponent),
  canActivate: [exitFormGuard]
},

```

¡Ya tenemos nuestra guardia cumpliendo su función dentro de nuestra web!



## Rellena el formulario de inscripción

Curso

Nombre

Apellidos

## Parte 2: Añadiendo NgRx

Es hora de añadir NgRx a nuestra aplicación, esto nos permitirá gestionar la información de los cursos de una forma global y nos permitirá reducir el número de llamadas a la API para que solo se hagan al inicio y tras una inscripción.



En una aplicación en la que tenemos una alta concurrencia de usuarios y por lo tanto hay riesgo de que los datos se modifiquen habitualmente, sí que sería recomendable pedir los datos cada vez que accedemos al listado de cursos, detalle de curso o a la home. Para este caso vamos a intentar minimizar el número de peticiones realizadas asumiendo que si hay varios usuarios interactuando con la web los datos pueden no mostrarse actualizados hasta que se recargue.

## Instalando librería

El primer paso que tenemos que llevar a cabo es añadir las librerías de estado y de efectos de NgRx, para ello sobre nuestra aplicación ejecutamos los comandos correspondientes en la CLI:

```
ng add @ngrx/store
ng add @ngrx/effects
```

Esto añadirá las dependencias al `package.json` y los providers al `app.config.ts`:

### app.config.ts

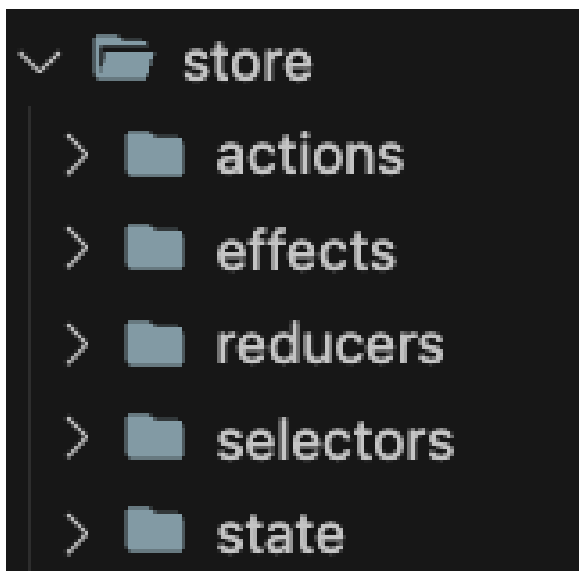
```
import { ApplicationConfig } from '@angular/core';
import { provideRouter, withInMemoryScrolling } from '@angular/router';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';
import { provideStore } from '@ngrx/store';
import { provideEffects } from '@ngrx/effects';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withInMemoryScrolling({ scrollPositionRestoration: 'top' })),
    provideHttpClient(),
    provideStore(),
    provideEffects()
  ]
};
```

## Creando estado, acciones y reducers

Vamos a crear dentro de **core** una carpeta **store** en la que iremos añadiendo todos los elementos de NgRX que necesitamos para nuestra aplicación:



Empezaremos añadiendo dentro de **store** nuestro estado global al que añadiremos una propiedad **courses** que contendrá el listado de cursos disponibles:

### store.state.ts

```
import { Course } from "../../services/courses.models";

export interface AppState {
  courses: Course[];
}
```

```
export const initialState: AppState = {
  courses: []
};
```



Ya que la información almacenada en el estado global es bastante simple no va es necesario crear un objeto con varias propiedades para nuestro subestado **courses**, es suficiente con que contenga directamente el listado de cursos.

Pasamos a definir las acciones que necesitaremos utilizar para actualizar los datos en nuestro estado, lo haremos dentro de **courses.actions.ts**:

```
import { createAction, props } from "@ngrx/store";
import { Course } from "../../services/courses.models";

export const retrieveCourses = createAction(
  '[Courses] Retrieve Courses',
);

export const setCourses = createAction(
  '[Courses] Set Courses',
  props<{ courses: Course[] }>(),
);

export const enrolleStudent = createAction(
  '[Courses] Enrole Student',
  props<{ courseId: string, newNumVacancies: number }>(),
);
```

Nuestras acciones serán:

- **retrieveCourses** será un efecto que llamará a la API a través del servicio para recuperar el listado de cursos con sus últimos datos disponibles.
- **setCourses** con un reducer modificará el estado para añadir los cursos recuperados tras la acción **retrieveCourses**.
- **enroleStudent** llamará al **PUT** que acabará modificando el número de plazas disponibles para el curso cuyo id se pase en el **payload** de la acción.

Es hora de añadir los **reducers**, en nuestro caso solo necesitamos uno, el que se encarga de almacenar los cursos.

**courses.reducers.ts**

```
import { createReducer, on } from "@ngrx/store";
import { setCourses } from "../../actions/courses.actions";
import { Course } from "../../services/courses.models";

export const coursesReducer = createReducer(
  [] as Course[],
  on(setCourses, (state, { courses }) => courses),
);
```

Damos de alta el reducer en el **app.config.ts**:

```
provideStore({
  courses: coursesReducer
```

```
}},
```

## Creando efectos y selectores

Nuestra aplicación necesitará utilizar dos efectos, uno para recuperar los cursos y el otro para inscribir a un nuevo estudiante en un curso. Los añadiremos sobre nuestro archivo **courses.effects.ts**:

```
import { Actions, ofType } from "@ngrx/effects";
import { CoursesService } from "../../services/courses.service";
import { enrolleStudent, retrieveCourses, setCourses } from "../../actions/courses.actions";
import { map, switchMap, tap } from "rxjs";
import { Router } from "@angular/router";

@Injectable()
export class CourseEffects {

  retrieveCourses$ = this.actions$.pipe(
    ofType(retrieveCourses),
    switchMap(() => this.coursesService.getCourses()),
    map(courses => setCourses({ courses })),
  );

  enrolleStudent$ = this.actions$.pipe(
    ofType(enrolleStudent),
    switchMap(({ courseId, newNumVacancies }) => this.coursesService.enrolleStudent(
      courseId, newNumVacancies
    )),
    tap(() => this.router.navigate(['/courses'])),
    map(() => retrieveCourses()),
  );

  constructor(
    private actions$: Actions,
    private coursesService: CoursesService,
    private router: Router
  ) {}
}
```

- El efecto `retrieveCourses$` utiliza el método `getCourses` del servicio para recuperar el listado de cursos y luego a través de la acción `setCourses` los añade al estado de la aplicación.
- El efecto `enrolleStudent$` llama al método `enrolleStudent` para que se haga un **POST** a la api actualizando el número de plazas. Como efecto secundario navega al listado de cursos y acaba devolviendo la acción `retrieveCourses` que provoca que se actualice la lista de cursos con los nuevos datos.

Damos de alta los efectos en el **app.config.ts**:

```
provideEffects([
  CourseEffects
])
```

Para finalizar con los elementos necesarios para que nuestra aplicación funcione con NgRx vamos a añadir selectores. Lo haremos sobre el archivo **courses.selectors.ts**:

```
import { createFeatureSelector, createSelector } from "@ngrx/store";
import { Course } from "../../services/courses.models";
```



```

export const selectCourses = createFeatureSelector<Course[]>('courses');

export const selectTopCourses = createSelector(
  selectCourses,
  courses => courses.slice(0, 4)
);

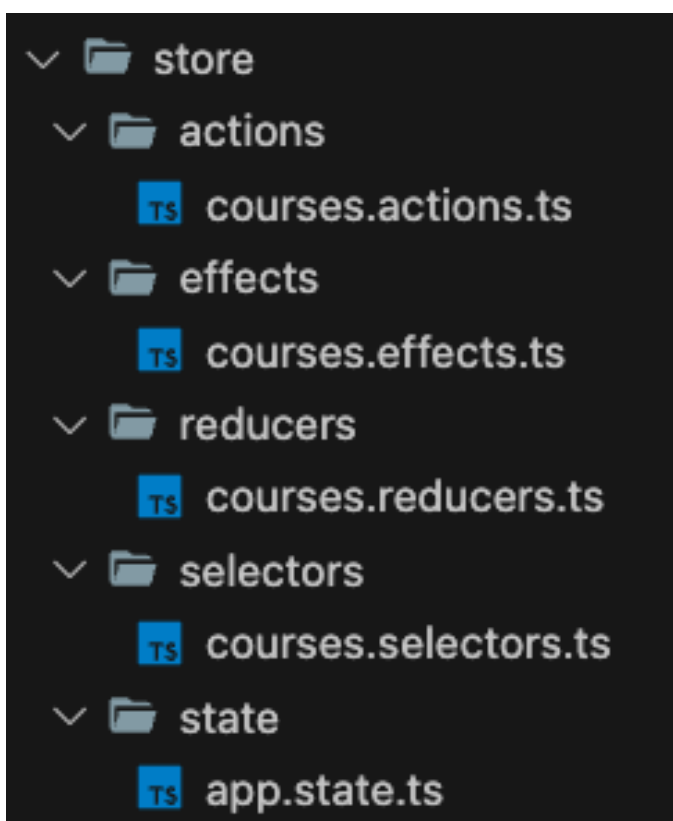
export const selectCourseById = (id: string) => createSelector(
  selectCourses,
  courses => courses.find(course => course.id === id)
);

```

Creamos 3 selectores:

- `selectCourses` recupera todos los cursos en el estado.
- `selectTopCourses` transforma los datos del primer selector para quedarse solo con los 4 primeros cursos (esto nos permite eliminar el método `getTopCourses` del servicio).
- `selectCourseById` recupera el curso con el id que recibe como argumento (esto nos permite eliminar el método `getCourseById` del servicio).

Y con esto tenemos nuestra carpeta de estado terminada:



## Aplicando NgRx en las vistas

Ahora que tenemos todos los elementos de NgRx añadidos y funcionando tenemos que empezar a utilizarlos en los diferentes sitios de nuestra web. Empezaremos recuperando la información utilizando los selectores en lugar del servicio.

### home.component.ts

```

export class HomeComponent implements OnInit {

  public topCourses$: Observable<Course[]>;

  constructor(private store: Store<AppState>) {}

  public ngOnInit() {
    this.topCourses$ = this.store.select(selectTopCourses);
  }
}

```

```
}  
}
```

#### **courses-list.component.ts**

```
export class CoursesListComponent implements OnInit {  
  
  public courses$?: Observable<Course[]>;  
  
  constructor(private store: Store<AppState>) {}  
  
  public ngOnInit() {  
    this.courses$ = this.store.select(selectCourses);  
  }  
  
}
```

#### **course-detail.component.ts**

```
constructor(  
  private store: Store<AppState>,  
  private activatedRoute: ActivatedRoute,  
) {}  
  
public ngOnInit() {  
  this.listenRouteParamsChanges();  
}  
  
public listenRouteParamsChanges() {  
  this.activatedRoute.params.subscribe((params) => {  
    const courseId = params['id'];  
    if (!courseId) { return; }  
    this.store.select(selectCourseById(courseId)).subscribe((course) => {  
      this.course = course;  
      this.initializeEnrolmentForm();  
    });  
  });  
}
```

Simplemente hemos sustituido las llamadas al servicio por el selector correspondiente que se encarga de recuperar la misma información.

Pero tener los selectores recuperando datos no sirve de nada si no estamos realizando el **dispatch** de las acciones que permite recuperar la información de los cursos y almacenarla.

Queremos utilizar nuestra acción `retrieveCourses` en dos situaciones:

- Nuestra aplicación se acaba de inicializar y hay que recuperar todos los datos de los cursos. Para este caso añadiremos el dispatch de nuestra acción en el **OnInit** de nuestro **app.component.ts**.

#### **app.component.ts**

```
export class AppComponent implements OnInit {  
  
  constructor(private store: Store<AppState>) { }  
  
  public ngOnInit() {  
    this.store.dispatch(retrieveCourses());  
  }  
}
```

```
}
}
```

- Cuando se haya finalizado la inscripción de un alumno y por lo tanto necesitamos recuperar la información de los cursos actualizada con el nuevo número de plazas. Esto lo hará automáticamente la acción `enroleStudent` que lanzaremos desde `course-detail.component.ts`.

#### course-detail.component.ts

```
public enroleStudent(): void {
  if (!this.enrollementForm || !this.enrollementForm.valid || !this.course) {
    return;
  }
  this.store.dispatch(enroleStudent({
    courseId: this.course.id, newNumVacancies: this.course.numVacancies - 1
  }));
  this.enrollementForm.reset();
}
```

## Redux Devtools

Para comprobar que todo el trabajo que hemos realizado funciona tal y como esperamos vamos a añadir las **Redux Devtools** de forma que podamos validar nuestro flujo de acciones. Para ello en la CLI lanzamos el siguiente comando:

```
ng add @ngrx/store-devtools
```

Esto añadirá el provider correspondiente al `app.config.ts`:

```
provideStoreDevtools({ maxAge: 25, logOnly: !isDevMode() })
```



Para utilizar la extensión recordad instalarla en Chrome

Con la extensión instalada vamos a probar los dos principales flujos de acciones que se dan en la aplicación:

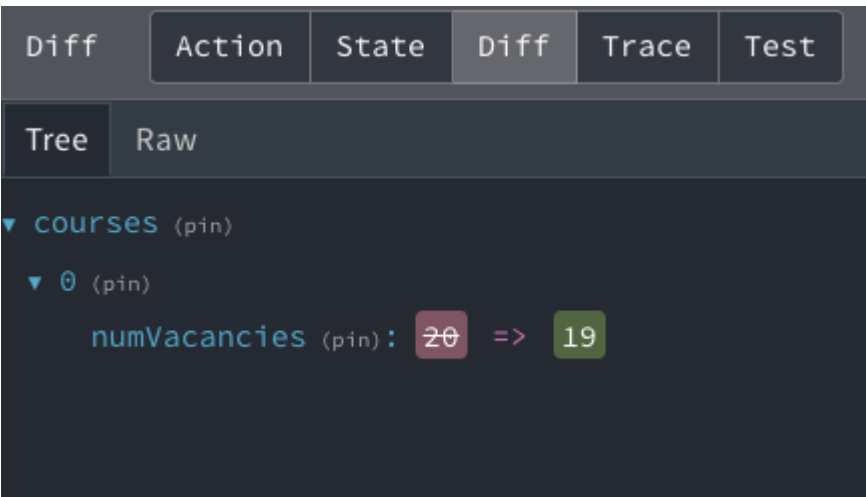
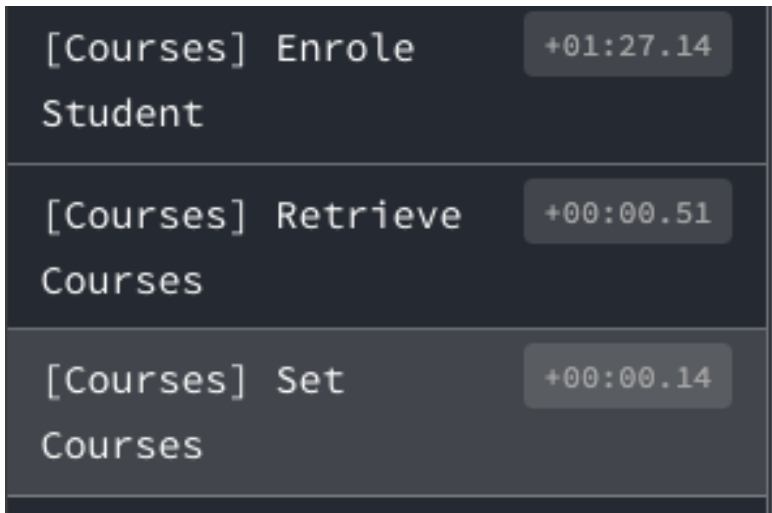
- Cuando se inicializa en cualquier página la aplicación deben recuperarse los cursos y añadirse al estado global.

filter...	Time	Diff	Action	State	Diff	Trace	Test
@ngrx/store/init	1:36:40.49						
@ngrx/effects/init	+00:00.00						
[Courses] Retrieve Courses	+00:00.00						
[Courses] Set Courses	+00:00.11						

Tree	Raw
▼ courses (pin)	
0 (pin):	{id:'1',title:'Máster ThePower...ancies:20}
1 (pin):	{id:'2',title:'Máster en Venta...ancies:19}
2 (pin):	{id:'3',title:'ThePowerMBA + V...ancies:29}
3 (pin):	{id:'4',title:'Máster en Consu...ancies:10}
4 (pin):	{id:'5',title:'ThePower Digita...ancies:50}
5 (pin):	{id:'6',title:'Máster en Intel...ancies:60}

- Cuando se inscribe un nuevo alumno en un curso debe lanzarse la acción para actualizar el número de plazas del curso y debe recuperarse la información de los cursos actualizada.



¡Con estos cambios ya tenemos nuestra aplicación lista para funcionar con NgRx! Todo se comporta tal y como hasta ahora, con la diferencia de que ahora se realizan muchas menos llamadas a la API.

### Parte 3: Añadiendo SSR y SSG

Todavía hay un recurso adicional que podemos añadir a nuestra aplicación para mejorar su rendimiento aún más: SSR.

#### Análisis

Vamos a analizar para que páginas de las que actualmente existen dentro de nuestra aplicación tiene sentido que añadamos SSR o SSG:

##### Home:

En el caso de la home tenemos un contenido estático que siempre va a mostrar el mismo vídeo y el mismo eslogan. Lo único que puede llegar a cambiar son los 4 cursos que están considerados los más populares, pero esto tampoco cambiará habitualmente.

Teniendo en cuenta que nuestro contenido va a ser el mismo siempre tiene sentido que lo **prerendericemos** al compilar la aplicación y pasemos a servir directamente el propio HTML estático que hemos generado.

##### Listado de cursos:

Con el listado de cursos la cosa cambia, la información de las plazas disponibles en los cursos, su descripción, su imagen y sus diferentes datos pueden ser cambiados por el equipo de comercial en cualquier momento. Esto provocaría que si prerenderizamos el contenido sea muy probable que lo que luego se muestre en cliente al usuario sea totalmente distinto.

Por lo tanto para esta vista SSG no tiene sentido, sin embargo sí que podemos aplicar **SSR** para que el contenido de los cursos se renderice en servidor y la respuesta de la petición HTTP se cachee en el HTML.

##### Detalle de curso:

En el detalle de curso la información referente al curso que se muestra en la vista es muy poca: nombre, precio y descripción, además esos datos cambiarán con mucha menos frecuencia. Al no tener el número de plazas pintado esto hace que nuestros datos sean más susceptibles de ser cacheados.

Aun así tenemos una limitación, nuestra ruta incluye parámetros, esto nos obligaría a cachear uno a uno todos los cursos existentes. Teniendo siempre el riesgo de que se añadan nuevos cursos y tengamos problemas.

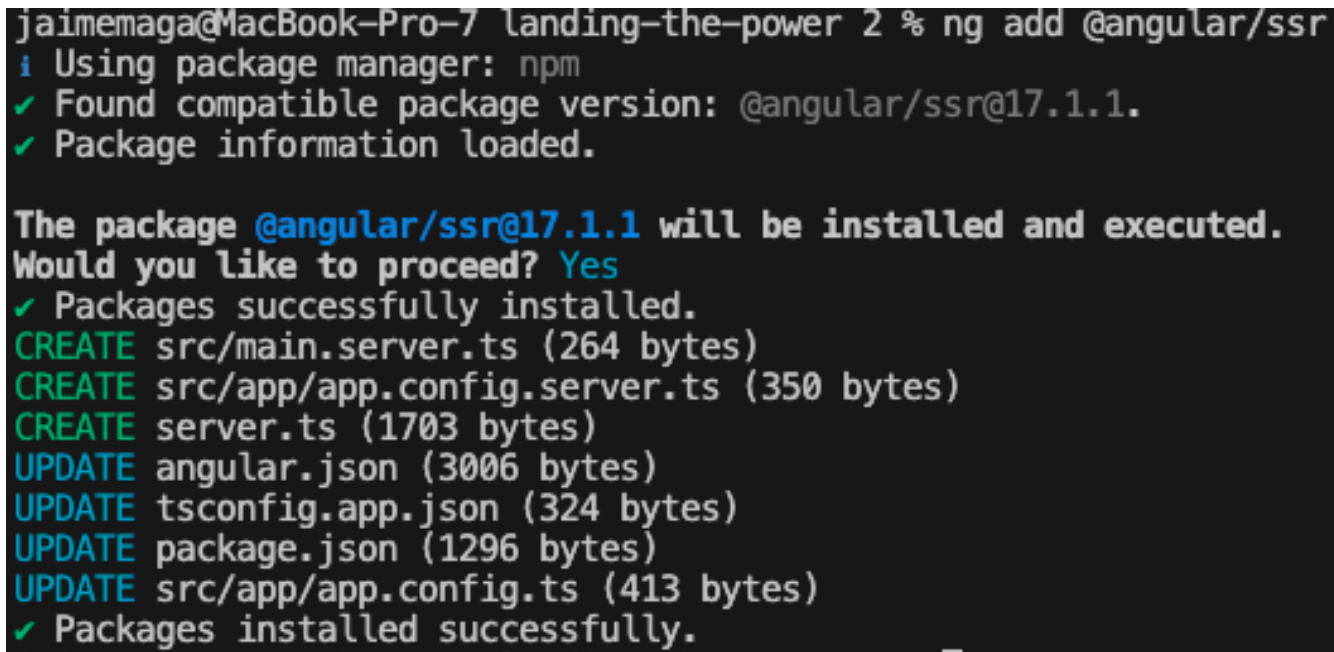
Por lo tanto para este caso no sería recomendable prerenderizar todos los cursos, pero si sería una buena estrategia **prerenderizar los cursos más populares**, ya que estos se muestran en la home y por lo tanto serán accedidos mucho más. Para el resto de cursos simplemente aplicaremos **SSR**.

### Añadiendo SSR

Una vez hemos decidido la mejor estrategia para cada una de nuestras vistas, es hora de añadir SSR a nuestra aplicación. Como nuestro proyecto ya había sido creado no podemos seleccionar la opción de SSR que solíamos escoger al crear el proyecto, pero no hay problema, ejecutando el siguiente comando en la CLI añadiremos SSR al proyecto:

```
ng add @angular/ssr
```

Esto generará todos los archivos que necesitamos para que nuestra aplicación se ejecute en el servidor



```
jaimemaga@MacBook-Pro-7 landing-the-power 2 % ng add @angular/ssr
i Using package manager: npm
✓ Found compatible package version: @angular/ssr@17.1.1.
✓ Package information loaded.

The package @angular/ssr@17.1.1 will be installed and executed.
Would you like to proceed? Yes
✓ Packages successfully installed.
CREATE src/main.server.ts (264 bytes)
CREATE src/app/app.config.server.ts (350 bytes)
CREATE server.ts (1703 bytes)
UPDATE angular.json (3006 bytes)
UPDATE tsconfig.app.json (324 bytes)
UPDATE package.json (1296 bytes)
UPDATE src/app/app.config.ts (413 bytes)
✓ Packages installed successfully.
```

Dentro de nuestro **app.config.ts** tendremos nuestro método `provideClientHydration` que activará la hidratación del contenido procedente del servidor:

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes, withInMemoryScrolling({ scrollPositionRestoration: 'top' })),
    provideHttpClient(),
    provideStore({
      courses: coursesReducer
    }),
    provideStoreDevtools({ maxAge: 25, logOnly: !isDevMode() }),
    provideEffects([CourseEffects]),
    provideClientHydration(),
  ]
};
```

En nuestro **app.config.server.ts** tendremos el método `provideServerRendering` que activará el **SSR**:

```
import { mergeApplicationConfig, ApplicationConfig } from '@angular/core';
import { provideServerRendering } from '@angular/platform-server';
import { appConfig } from './app.config';

const serverConfig: ApplicationConfig = {
  providers: [
    provideServerRendering()
  ]
};

export const config = mergeApplicationConfig(appConfig, serverConfig);
```

Además tendremos nuestro archivo **server.ts** con el servidor de **Express** que usaremos para ejecutar la aplicación en el lado del servidor.



## Configurando prerenderizado y SSR

Vamos a irnos al **angular.json** y vamos a dejar lista en el entorno de desarrollo la configuración de SSR y SSG que necesitamos para el proyecto:

```
"development": {
  "optimization": false,
  "extractLicenses": false,
  "sourceMap": true,
  "fileReplacements": [
    {
      "replace": "src/environments/environment.ts",
      "with": "src/environments/environment.development.ts"
    }
  ],
  "ssr": {
    "entry": "server.ts"
  },
  "prerender": {
    "discoverRoutes": false,
    "routesFile": "src/prerendered-routes.txt"
  }
}
```

Activamos SSR y especificamos que queremos utilizar nuestro **server.ts** como punto de entrada. A nivel de prerenderizado lo configuramos para que no sea Angular automáticamente el que decida qué rutas deben prerenderizarse poniendo **discoverRoutes** a **false**. Para personalizar las rutas le pasamos en la propiedad **routesFile** la url de un nuevo archivo que crearemos dentro de la carpeta src y que indicará las rutas que deben generarse durante la compilación.

Dentro de nuestro nuevo archivo **prerendered-routes.txt** añadimos la home y la url de los 4 cursos más populares separadas por un salto de línea. Esas serán las rutas que angular genere durante la compilación de nuestra aplicación.

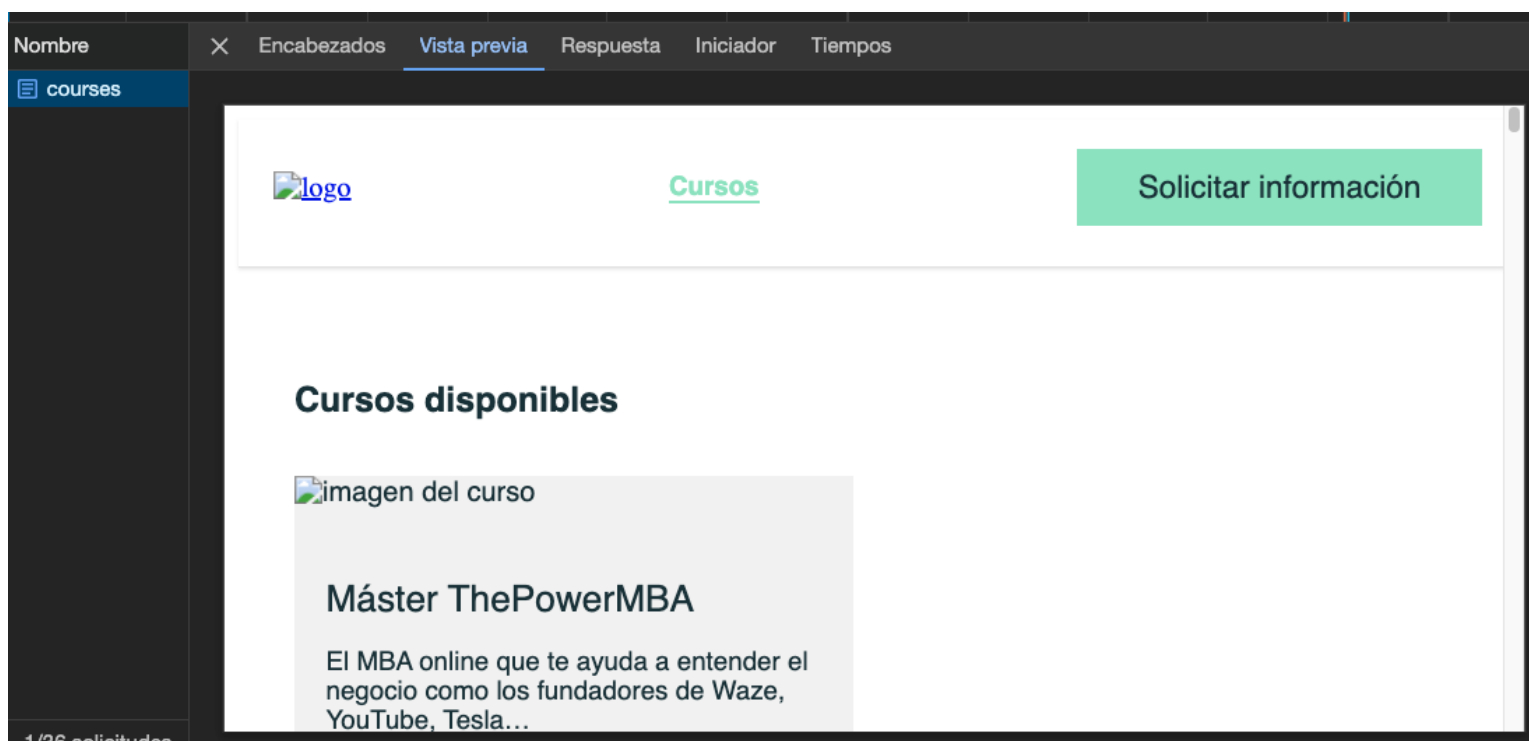
```
home
course/1
course/2
course/3
course/4
```

El resto de detalles de curso y la página con el listado de cursos las dejamos sin prerenderizar para que se generen utilizando SSR.

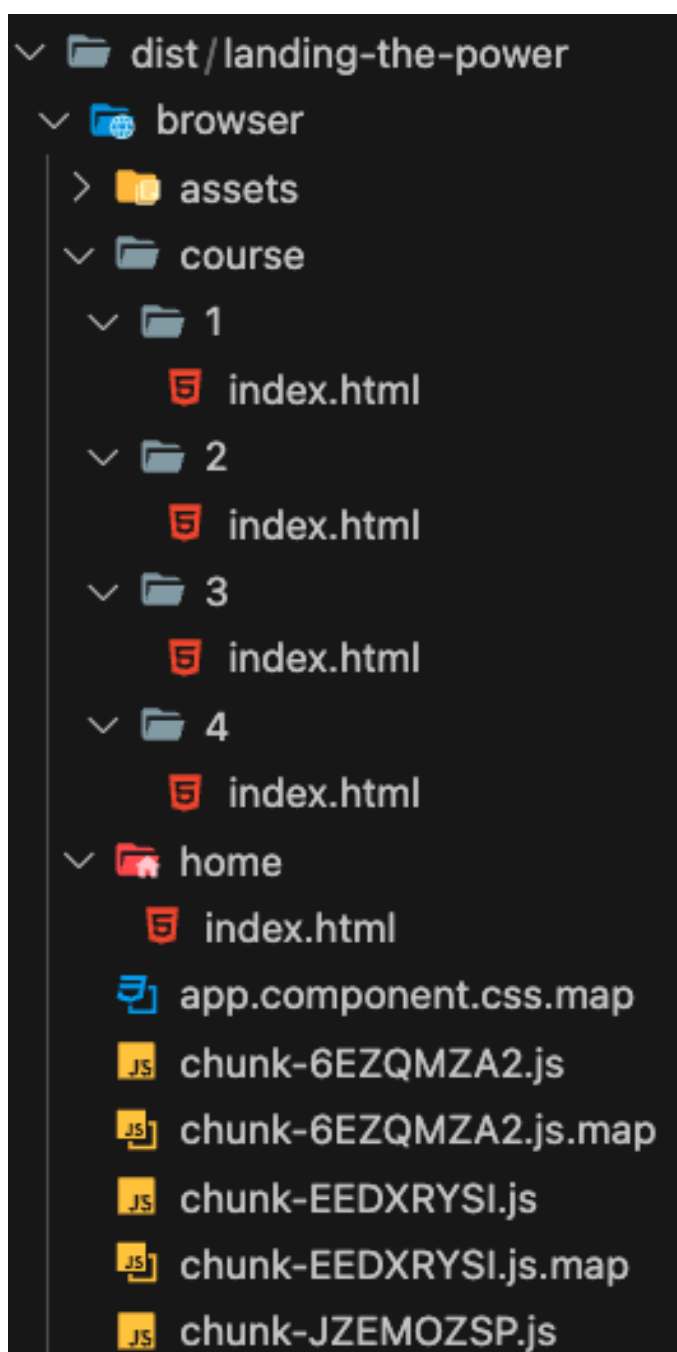


Este archivo a futuro sería interesante generarlo con un proceso automático que recuperará con una petición los cursos más populares y añadiera sus urls al txt.

Levantamos nuestra aplicación en primer lugar con **ng serve** para comprobar que el contenido esté viniendo tal y como esperamos del servidor:



Y a continuación hacemos un `ng build --configuration=development` para validar que se estén generando los HTML de las rutas correctas al compilar el proyecto:



También podemos levantar nuestro servidor con node **server.mjs** dentro de la carpeta **dist** para ver cómo se comporta nuestro proyecto sirviendo archivos estáticos.



A futuro habría que añadir la misma configuración en producción

## Platform service

Durante el desarrollo de nuestra aplicación con SSR necesitaremos saber en ocasiones si estamos ejecutando la aplicación en servidor o en cliente, para evitar realizar accesos a `window` que provoquen errores o para ejecutar lógica distinta. Para poder tener la información de la plataforma en la que se ejecuta la aplicación vamos a generar un nuevo **platform.service.ts** en la carpeta **core**.

```
ng g service platform
```

```
import { isPlatformBrowser, isPlatformServer } from '@angular/common';
import { Inject, Injectable, PLATFORM_ID } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class PlatformService {

  constructor(
    @Inject(PLATFORM_ID) public platformId: string
  ) { }

  public isBrowser(): boolean {
    return isPlatformBrowser(this.platformId);
  }

  public isServer(): boolean {
    return isPlatformServer(this.platformId);
  }
}
```

Los métodos `isBrowser` e `isServer` nos permitirán controlar en todo momento donde ejecutamos nuestro código.

## Cacheo de peticiones

En cuanto a estrategia de cacheo de peticiones HTTP vamos a dejar la que incluye el SSR de Angular por defecto, cacheamos todas las peticiones realizadas en servidor en el HTML devuelto de forma que no haya que repetirlas en cliente.

Gracias a esto seguimos lanzando las acciones de NgRx pero las peticiones utilizan la información cacheada en lugar de realizarse.

## Rehidratación del store de NgRx

Aunque cacheemos las peticiones en el servidor siempre es interesante aprovechar el estado de NgRx que se generó en el servidor para luego rehidratarlo una vez en cliente.

La rehidratación del estado de NgRx consiste en transferir el estado generado en el servidor al cliente a través del HTML devuelto. Esto se logra utilizando el `TransferState` de Angular.

Cuando se renderiza una página en el servidor, el estado de NgRx se almacena en la `TransferState`. Luego, cuando la aplicación se carga en el cliente, se recupera ese estado almacenado y se utiliza para inicializar el estado de NgRx en el cliente.

El `TransferState` actúa como un almacén temporal que permite compartir el estado entre el servidor y el cliente durante la transición. Esto es especialmente útil para mantener la coherencia del estado entre el servidor y el cliente.

Cuando estemos en el servidor queremos ir almacenando el estado de NgRx actualizado tras ejecutar cada `reducer`, de forma que lo que quede finalmente cacheado sea el estado actualizado y con toda la información necesaria para renderizar la vista. Para ello podemos recurrir a un `metareducer` ya que nos permitirá ejecutar esta lógica de seteo en el `TransferState` para cada `reducer` ejecutado dentro de nuestra aplicación.

Dentro de la carpeta **store** creamos un archivo **app.metareducers.ts**:

```
import { ActionReducer } from "@ngrx/store";
import { AppState } from "../state/app.state";
import { inject, makeStateKey } from "@angular/core";
import { TransferState } from "@angular/core";
import { PlatformService } from "../../services/platform.service";

const stateKey = makeStateKey<AppState>('state');

export const rehydrateMetaReducer = (
  reducer: ActionReducer<AppState>
): ActionReducer<AppState> => {
  const transferState = inject<TransferState>(TransferState);
  const platformService = inject(PlatformService);
  return (state, action) => {
    if (platformService.isBrowser() && transferState.hasKey<AppState | undefined>(stateKey)) {
      const stateFromServer = transferState.get<AppState | undefined>(stateKey, undefined);
      if (stateFromServer) {
        transferState.remove(stateKey);
        return stateFromServer;
      }
    }
    const newState = reducer(state, action);
    if (platformService.isServer()) {
      transferState.set(stateKey, newState);
    }
    return newState;
  };
}
```

Nuestro **metareducer** tiene varias partes importantes:

- En la función `rehydrateMetaReducer` tenemos contexto de inyección lo que nos permite añadir providers utilizando el `inject` y nos evita el tener que crear una clase con constructor. Aquí añadimos tanto el `PlatformService` como el `TransferState` que necesitaremos a continuación.
- Si estamos en cliente revisamos si hay información almacenada en el `TransferState` y en caso de que la haya devolvemos esos datos en lugar de ejecutar el reducer por defecto. Además eliminamos los datos del `TransferState` de forma que en próximas acciones ya no se utilice esa información.
- Tras calcular el nuevo estado y haber pasado por el reducer correspondiente, si estamos en servidor almacenamos en el `TransferState` los datos del estado ya actualizados.

También debemos registrar nuestro nuevo **metareducer** en el archivo **app.config.ts**:

```
provideStore({
  courses: coursesReducer
}, {
  metaReducers: [rehydrateMetaReducer]
}),
```

Gracias a estos cambios ya no debería ser necesario recuperar los cursos al inicializar la aplicación si estamos en cliente, ya que el propio **metareducer** al ejecutar la acción nativa `@ngrx/store/init` se encargará de rehidratar el estado con los datos provenientes del servidor.

**app.component.ts**

```
export class AppComponent implements OnInit {

  constructor(
    private store: Store<AppState>,
    private platformService: PlatformService
  ) { }

  public ngOnInit() {
    if (this.platformService.isServer()) {
      this.store.dispatch(retrieveCourses());
    }
  }
}
```



Sería posible desactivar la caché de HTTP nativa de SSR, así evitaríamos estar cacheando los mismos datos dos veces: la respuesta de la petición y el estado de NgRx. El estado de NgRx nos da la versatilidad de poder cachear todos los datos que son necesarios para que nuestra vista se renderice, ya con la información transformada y con datos que pueden no venir de peticiones HTTP.

#### app.config.ts

```
provideClientHydration(
  withNoHttpTransferCache()
)
```

Ahora no solo evitamos llamar a las peticiones iniciales al cargar nuestra aplicación en cliente si no que además tampoco necesitamos lanzar ninguna acción de NgRx de inicio. Solo se volverán a lanzar acciones y a pedir datos a la API cuando el usuario se inscriba en un curso.



Siempre que cacheemos peticiones como estas en el HTML debemos tener cuidado de que el archivo no se cachee tiempos largos, o que si lo hace tengamos algún mecanismo para limpiar nuestra caché. Si no tenemos cuidado con esto podremos tener problemas de que lleguen datos desactualizados a los usuarios.