DESARROLLO WEB CON MEAN

Curso: DESARROLLO WEB CON MEAN (WEB FULL STACK DEVELOPER)

Angular 4

- **Angular** es un framework desarrollado por Google para desarrollar SPAs (Single Page Applications).
- Para desarrollar aplicaciones usando Angular se recomienda usar **TypeScript**.
- Las principales características son:
  - Desarrollo móvil
  - Modularidad
  - Compatibilidad

Angular 4: Arquitectura

- Angular se compone de 8 bloques:
  - Módulos
  - Componentes
  - Directivas
  - Templates o plantillas
  - Metadatos
  - Data Binding
  - Servicios
  - Inyección de dependencias

Angular 4: Angular-CLI

- Es el intérprete de línea de comandos de Angular.
- Facilita la creación del proyecto, y de cada una de las partes que vamos a necesitar en la aplicación.
- Requisitos:
  - Node
  - NPM
- Se instala con el comando: **$ npm install -g @angular/cli**
- Podemos ver la versión con: **$ ng -v**

Angular 4: Angular-CLI

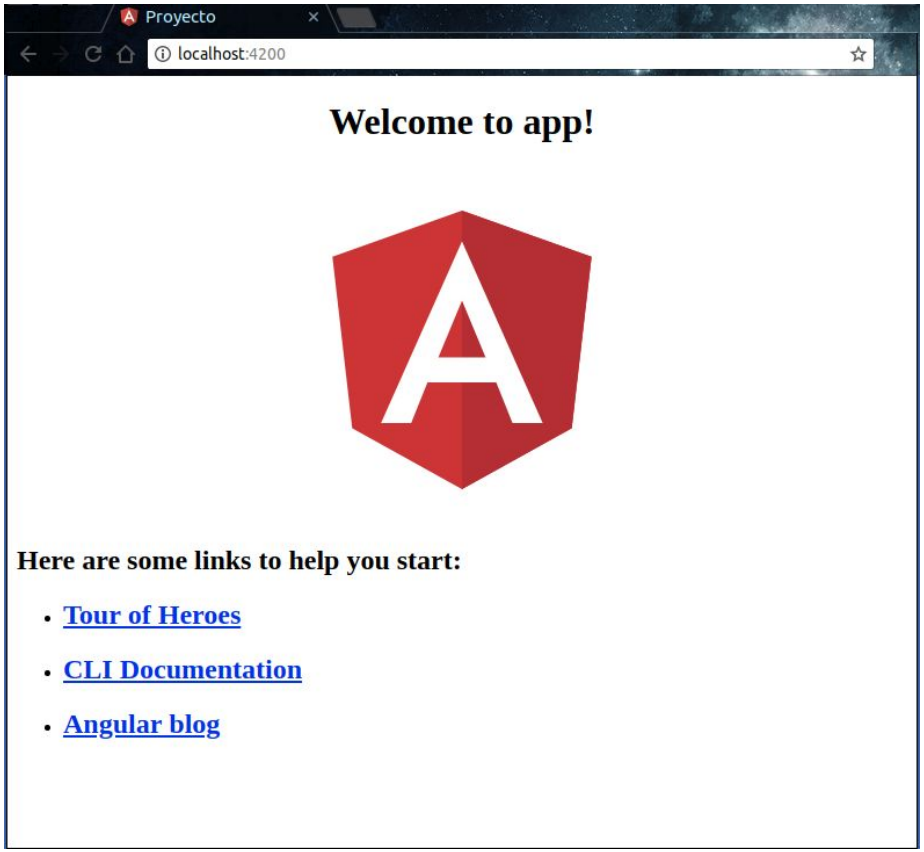
- Creamos un proyecto de Angular ya configurado con el comando:

$ ng new proyecto

Angular 4: Angular-CLI

- Levantamos la aplicación en un servidor local de desarrollo con:

```
$ ng serve
```



Angular 4: Angular-CLI

- Otros comandos:
  - Crear un componente: **$ ng g component un-componente**
  - Crear una directiva: **$ ng g directive una-directiva**
  - Crear un pipe: **$ ng g pipe un-pipe**
  - Crear un servicio: **$ ng g service un-servicio**
  - Crear una guard: **$ ng g guard una-guard**
  - Hay alguno más...

Angular 4: Angular-CLI

- Opciones de los comandos anteriores:
  - No generar los archivos de test: **--spec false**
  - No generar el archivo html: **-it**
  - No generar el archivo css: **-is**
  - No generar la carpeta contenedora de los archivos que se generan: **--flat**
- Ejemplo: **$ ng g component un-cmp --spec false -is -it --flat**

Angular 4: Componentes

- Las aplicaciones de Angular siguen el desarrollo basado en componentes.
- Los componentes son piezas de código (*piezas de lego*) que vamos a ir uniendo para crear la aplicación.
- Un componente puede estar compuesto por otros componentes.
- Dentro de un componente se distinguen tres partes:
  - Una plantilla
  - Una clase
  - Un decorador

Angular 4: Componentes

```ts
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```

Decorador

Clase

Plantilla o template

```html
<!-- app.component.html -->
<h1>
  Welcome to {{ title }}!
</h1>
```

Angular 4: Componentes

- Para usar un componente y que este se muestre en la aplicación, hay que usar una etiqueta html con el valor del atributo **selector** que hay dentro del decorador de dicho componente.

Angular 4: Ciclo de Vida de los Componentes

- El ciclo de vida de los componentes nos permite interactuar con ellos en ciertos momentos de la vida de estos, por ejemplo cuando se crea el componente…
- Los eventos o **hooks** que existen son:
  - ngOnChanges
  - ngOnInit
  - ngDoCheck
  - ngAfterContentInit
  - ngAfterContentChecked
  - ngAfterViewInit
  - ngAfterViewChecked
  - ngOnDestroy

Angular 4: Data Binding

- El **Data Binding** nos permite sincronizar los datos entre la plantilla y la clase del componente.
- Hay 4 tipos de Data Binding:
  - **String Interpolation**
  - **Property Binding**
  - **Event Binding**
  - **Two-Way Data Binding**

Angular 4: String Interpolation

- El **string interpolation** se usa para renderizar el valor de una propiedad del componente en la plantilla (en el **HTML**).
- Para mostrar el valor, hay que meter la propiedad entre llaves dobles.

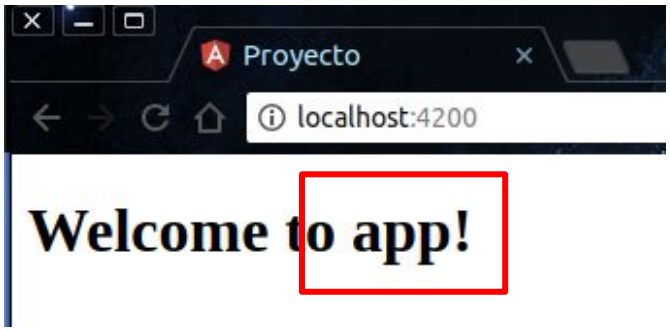
app.component.html

```
<h1>
  Welcome to {{ title }}!
</h1>
```

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```



Angular 4: Property Binding

- El **property binding** se usa para darle valor a las propiedades de las etiquetas HTML.
- El valor lo pueden obtener de una propiedad del componente, lo puede devolver un método del componente…
- Para usar este tipo de binding, hay que meter la propiedad entre corchetes, y el valor entre comillas.

```
export class AppComponent {
  title = 'app';
  deshabilitado = true;
}
```

```
<button type="button" [disabled]="deshabilitado">Enviar</button>
```

Angular 4: Event Binding

- Con el **event binding** podemos detectar un evento sobre un elemento HTML de la plantilla y ejecutar un método del componente.
- Para poder usarlo, el evento se mete entre paréntesis, y el método que se va a ejecutar entre comillas.

```
<button type="button" [disabled]="deshabilitado" (click)="enviarForm()">Enviar</button>
```

```
export class AppComponent {
  title = 'app';
  deshabilitado = true;

  enviarForm() {
    // Aquí se añade la lógica
  }
}
```

Angular 4: Two-Way Data Binding

- El **two-way data binding** permite leer y modificar los datos.
- Para usarlo, añadimos **[(ngModel)]** a un elemento HTML (normalmente un *input*) y le asignamos la propiedad que queremos sincronizar.
- Es necesario importar el módulo **FormsModule** en el módulo raíz de la aplicación.

```
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
```

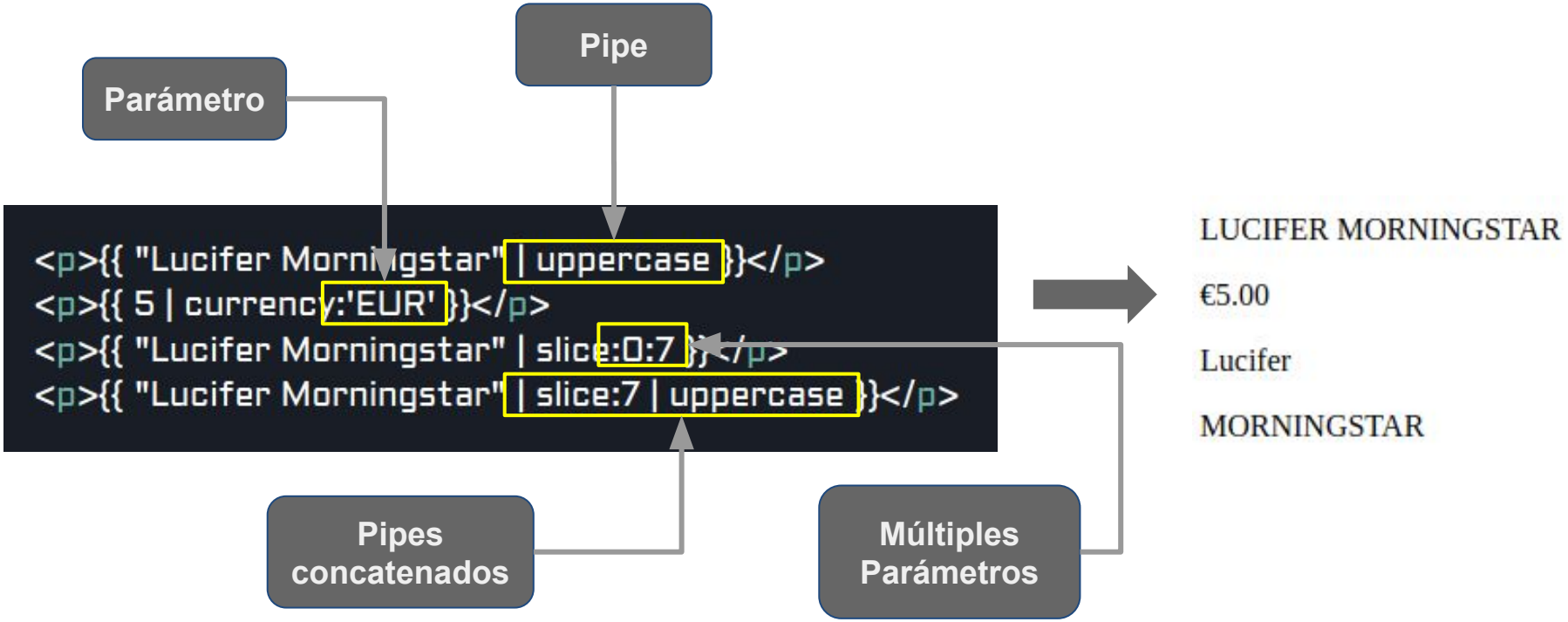
```
<input type="text" [(ngModel)]="title">
```

Angular 4: Pipes

- Los **Pipes** cogen un dato de entrada y modifican la forma en que se va a visualizar.
- Para usar un pipe, hay que añadir el símbolo del pipe y el nombre de aquel que queremos usar después del valor que se va a alterar.
- Hay dos tipos de pipes:
  - Pipes puros.
  - Pipes impuros.

Angular 4: Pipes puros

- Los **pipes puros** son aquellos que se aplican cuando un **valor primitivo** (*string*, *number*...) sufre un cambio.



Angular 4: Crear un pipe

- Para crear nuestros pipes personalizados tenemos que lanzar el comando: **$ ng g p nombre-pipe**
- Rellenamos el método **transform**.
- El valor que devuelve es la forma en que se va a visualizar en la plantilla.

```
1  import { Pipe, PipeTransform } from '@angular/core';
2
3  @Pipe({
4    name: 'doble'
5  })
6  export class DoblePipe implements PipeTransform {
7    transform(value: any, args?: any): any {
8      return value*2;
9    }
10 }
```

```
<p>El doble de 5 es: {{ 5 | doble }}</p>
```

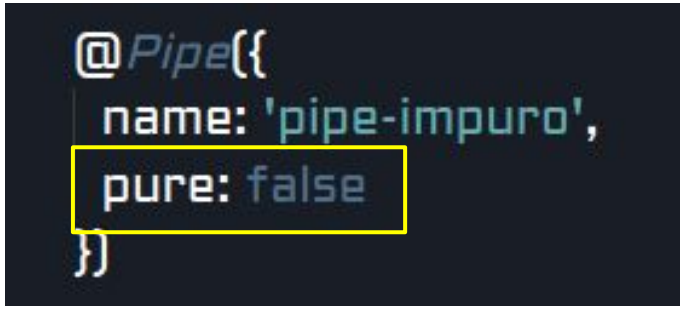
El doble de 5 es: 10

Angular 4: EJERCICIO

- Crear un pipe **reverse** que dado un string lo muestre al revés:
  - {{ “hola mundo” | reverse }} => “odnum aloh”
- Crear un pipe **times** que dado un string lo muestre tantas veces como se le pase como parámetro:
  - {{ “Penny… ” | times:3 }} => “Penny… Penny… Penny...”

Angular 4: Pipes impuros

- Los **pipes impuros** son aquellos que se usan con los **valores por referencia** (*arrays* y *objetos*).
- Un pipe pasa a ser impuro cuando se añade **pure: false** en el decorador.
- Estos pipes se aplican cada vez que Angular detecta un cambio en el componente.
- Son muy costosos en cuanto a recursos.



Angular 4: Pipe *async*

- El **pipe async** o pipe asíncrono (es un pipe impuro), es aquel que se subscribe a una **promesa** u **observable** y muestra los datos una vez que los recibe.

```
mensaje = new Promise<string>(resolve => {
  setTimeout(() => {
    resolve('Winter is coming...')
  }, 2000);
});
```

```
<p>Mensaje: {{ mensaje | async }}</p>
```

Después de 2 segundos

Mensaje: Winter is coming...

Angular 4: Directivas

- Las **directivas** le indican a Angular que tiene que hacer en algunas partes del código.
- Añaden comportamiento dinámico a la aplicación.
- Los componentes son directivas, pero con una plantilla.
- Hay dos tipos de directivas:
  - Directivas de atributo.
  - Directivas estructurales.

Angular 4: Directivas de atributo

- Las **directivas de atributo** interaccionan con el elemento al que se le aplican para alterar su apariencia o comportamiento.
- Algunas directivas que nos podemos encontrar son:
  - ngClass: añade clases al elemento.
  - ngStyle: añade estilos al elemento.
  - ngModel: aplica el two-way data binding.

```
<h1 [ngStyle]="{ color: 'blue', textDecoration: 'underline' }">Título</h1>
<h1 [ngClass]="{ letraAzul: true, subrayado: false }">Título</h1>
```

```
.subrayado {
  text-decoration: underline;
}
.letraAzul {
  color: blue;
}
```

Título

Título

Angular 4: Directivas estructurales

- Las **directivas estructurales** interaccionan con la vista cambiando la estructura del DOM.
- Las directivas que nos podemos encontrar son:
  - *ngIf
  - *ngFor
  - ngSwitch

Angular 4: Directivas estructurales

- ***ngIf**: añade un elemento o lo elimina del DOM dependiendo de una condición.

```
<p *ngIf="true">Se está mostrando</p>
```

Se está mostrando

Angular 4: Directivas estructurales

- ***ngFor**: se encarga de crear tantos elementos HTML como elementos haya en el array que se recorre.

```
<ul>
  <li *ngFor="let num of [1, 2, 3, 4]; let i = index">{{ i }}: {{ num }}</li>
</ul>
```

- 0: 1
- 1: 2
- 2: 3
- 3: 4

Angular 4: Directivas estructurales

- **ngSwitch**: funciona como un **switch**, va a mostrar el elemento HTML cuyo **case** coincida con el valor de esta directiva.

```
<div [ngSwitch]="'canario'">
  <p *ngSwitchCase="'perro'">Tengo un perro</p>
  <p *ngSwitchCase="'gato'">Tengo un gato</p>
  <p *ngSwitchDefault>No tengo ni perro, ni gato...</p>
</div>
```

No tengo ni perro, ni gato...

Angular 4: Servicios

- Los **servicios** son clases donde se va a encontrar la lógica de negocio, el código de acceso a la BBDD, constantes…
- Don’t repeat yourself.
- Para crear un servicio hay que usar el siguiente comando: **$ ng g s un-servicio**

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

@Injectable()
export class UnServicioService {
  mostrarMensaje(mensaje) {
    console.log('Mensaje: ' + mensaje);
  }
}
```

Angular 4: Inyección de dependencias

- La **inyección de dependencias** proporciona nuevas instancias de servicios para poder usarlos en los componentes de la aplicación.
- Para inyectar un servicio en un componente hay que añadirlo en el **constructor** de este.
- Además hay que registrar el servicio como proveedor en un módulo (si queremos que toda la aplicación comparta la misma instancia de ese servicio, lo haremos en el **providers** del módulo raíz).

Angular 4: Inyección de dependencias

```
// Resto de import
import { UnServicioService } from './un-servicio.service';

@NgModule({
  declarations: [
    AppComponent,
    DoblePipe
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [UnServicioService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
})
export class AppComponent {
  title = 'app';
  deshabilitado = true;

  constructor(private unServicio: UnServicioService) { }

  enviarMsg(msg) {
    this.unServicio.mostrarMensaje(msg);
  }
```

Angular 4: Inyectando un servicio en otro

- Para poder usar un servicio dentro de otro, hay que añadir el decorador **@Injectable()**.

```
import { Injectable } from '@angular/core';
import { UnServicioService } from './un-servicio.service';

@Injectable()
export class OtroServicioService {
 constructor(private unServicio: UnServicioService) { }

 mostrarMensaje(mensaje) {
  let num = Math.floor(Math.random() * 10);
  this.unServicio.mostrarMensaje('el número aleatorio es ' + num);
 }
}
```

Angular 4: Routing

- El routing nos va a permitir cambiar entre los distintos componentes de la aplicación.
- El routing de las SPA es distinto al routing tradicional (cada vez que se cambia de ruta se pide una página HTML al servidor).
- Las rutas se declaran en un archivo **app.routing.ts**
- La mayoría de las rutas tendrán:
  - **path**: forma que tiene que tener la URL para activar esta ruta.
  - **component**: componente que hay que mostrar cuando la URL coincide con el **path**.
- Hay que importar en el **módulo raíz** un módulo con las rutas definidas.

Angular 4: Routing

```
import { Routes, RouterModule } from "@angular/router";
import { InicioComponent } from "inicio.component";
import { ListaDatosComponent } from "lista-datos.component";

const APP_ROUTES: Routes = [
  { path: '', component: InicioComponent },
  { path: 'lista', component: ListaDatosComponent }
];

export const routing = RouterModule.forRoot(APP_ROUTES);
```

```
import { UnServicioService } from './un-servicio.service';
import { routing } from './app.routing';

@NgModule({
  declarations: [
    AppComponent,
    DoblePipe
  ],
  imports: [
    BrowserModule,
    FormsModule,
    routing
  ],
  providers: [UnServicioService]
```

Angular 4: Routing

- Para navegar entre las distintas rutas, se usa **routerLink** en lugar de href.
- Y hay que añadir la etiqueta **router-outlet** en el lugar donde se vayan a mostrar los componentes.

```
<h1>Routing</h1>
<a [routerLink]="['/']"></a>
<a [routerLink]="['/lista']"></a>
<hr>
<router-outlet></router-outlet>
```

Angular 4: Parámetros en las rutas

- Se pueden pasar parámetros en las rutas para hacer éstas más dinámicas.
- Los parámetros se añaden con **:param** en la declaración de las rutas.

```
import { Routes, RouterModule } from "@angular/router";
import { InicioComponent } from "inicio.component";
import { ListaDatosComponent } from "lista-datos.component";

const APP_ROUTES: Routes = [
  { path: '', component: InicioComponent },
  { path: 'lista/:categoria', component: ListaDatosComponent }
];

export const routing = RouterModule.forRoot(APP_ROUTES);
```

Angular 4: Parámetros en las rutas

- Para obtener los parámetros de una ruta en un componente, hay que importar **ActivatedRoute** y subscribirse a **paramMap**.
- Cada vez que los parámetros de la ruta cambian, se ejecuta la función que se le pasa al método **subscribe**.

```
<h1>Lista datos de {{ categoria | uppercase }}</h1>
```

Angular 4: Parámetros en las rutas

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-lista-datos',
  templateUrl: './lista-datos.component.html',
  styleUrls: ['./lista-datos.component.css']
})
export class ListaDatosComponent implements OnInit {
  categoria: string;
  constructor(private activatedRoute: ActivatedRoute) { }

  ngOnInit() {
    this.activatedRoute.paramMap.subscribe(
      (params) => this.categoria = params.get('categoria')
    );
  }
}
```

Angular 4: Rutas hijas

- Las **rutas hijas** son aquellas que cuelgan de otras rutas.
- Por ejemplo, si tenemos una ruta como **producto/:id**, puede tener como rutas hijas:
  - **editar**
  - **info**
- Hay que indicar en el archivo de rutas, que la ruta padre, tiene unas rutas hijas con la propiedad **children**.

Angular 4: Rutas hijas

```
import { Routes } from "@angular/router";
import { EditarProdComponent } from "editar-prod.component";
import { InfoProdComponent } from "info-prod.component";

export const PRODUCTO_ROUTES: Routes = [
  { path: 'editar', component: EditarProdComponent },
  { path: 'info', component: InfoProdComponent }
];
```

```
import { Routes, RouterModule } from "@angular/router";
import { InicioComponent } from "inicio.component";
import { ListaDatosComponent } from "lista-datos.component";
import { PRODUCTO_ROUTES } from './producto.routing';

const APP_ROUTES: Routes = [
  { path: '', component: InicioComponent },
  { path: 'lista/:categoria', component: ListaDatosComponent, children: PRODUCTO_ROUTES }
];

export const routing = RouterModule.forRoot(APP_ROUTES);
```

Angular 4: Rutas hijas

- Una vez definidas las rutas, hay que añadir la etiqueta **router-outlet** en el componente para indicar donde se tienen que mostrar los componentes de las rutas hijas.

Angular 4: EJERCICIO

- Crear una aplicación para manejar tareas con las siguientes funcionalidades:
  - Tiene que mostrar un listado con todas las tareas.
  - Tiene que permitir añadir nuevas tareas.
  - Tiene que permitir eliminar tareas.
  - Tiene que permitir marcar/desmarcar las tareas como completadas.
  - Las tareas que están marcadas como completadas, tienen que aparecer tachadas.
  - Tiene que haber mínimo dos rutas:
    - Una para mostrar el listado.
    - Una para mostrar el formulario con el que añadir nuevas tareas.