



# Angular

<https://www.tektutorialshub.com/angular/angular-introduction/>

# Características Principales

- Enlace de datos bidireccional (2 way [Data Binding](#))
  - Está muy bien implementado en Angular. Es automático y rápido. Los cambios hechos en la Vista, se actualizan automáticamente en la clase del componente y viceversa
- Soporte potente para [Routing](#)
  - Carga paginas asíncronamente en la página en la que estamos, permitiendo generar aplicaciones SPA (Single Page Apps)
- HTML con expresiones
  - Angular permite usar elementos de programación como son condiciones if, bucles for, etc al visualizar y controlar nuestras páginas HTML
- Modular por Diseño
  - Angular permite crear y usar módulos para contener nuestros programas
- Built-in Back End Support
  - Angular se comunica fácilmente con los servidores back-end y sabe ejecutar lógica de negocio y recuperar datos de los mismos.
- Comunidad muy activa
  - Angular está soportado por Google y una extensa comunidad de desarrolladores

# Los Componentes Básicos



## Componentes

En Angular, un componente es un elemento de la interfaz de usuario. Puedes tener muchos componentes en una única página web. Cada componente es independiente de los demás y gestiona su propia "sección". Pueden comunicarse con sus componentes hijos y padres



## Directivas



## Enlace de Datos (Data binding)

- Interpolación
- Enlace de datos unidireccional
- Enlace de eventos
- Enlace de datos bidireccional
- Enlace de Propiedades



## Pipas



## Bootstrap



## Servicios

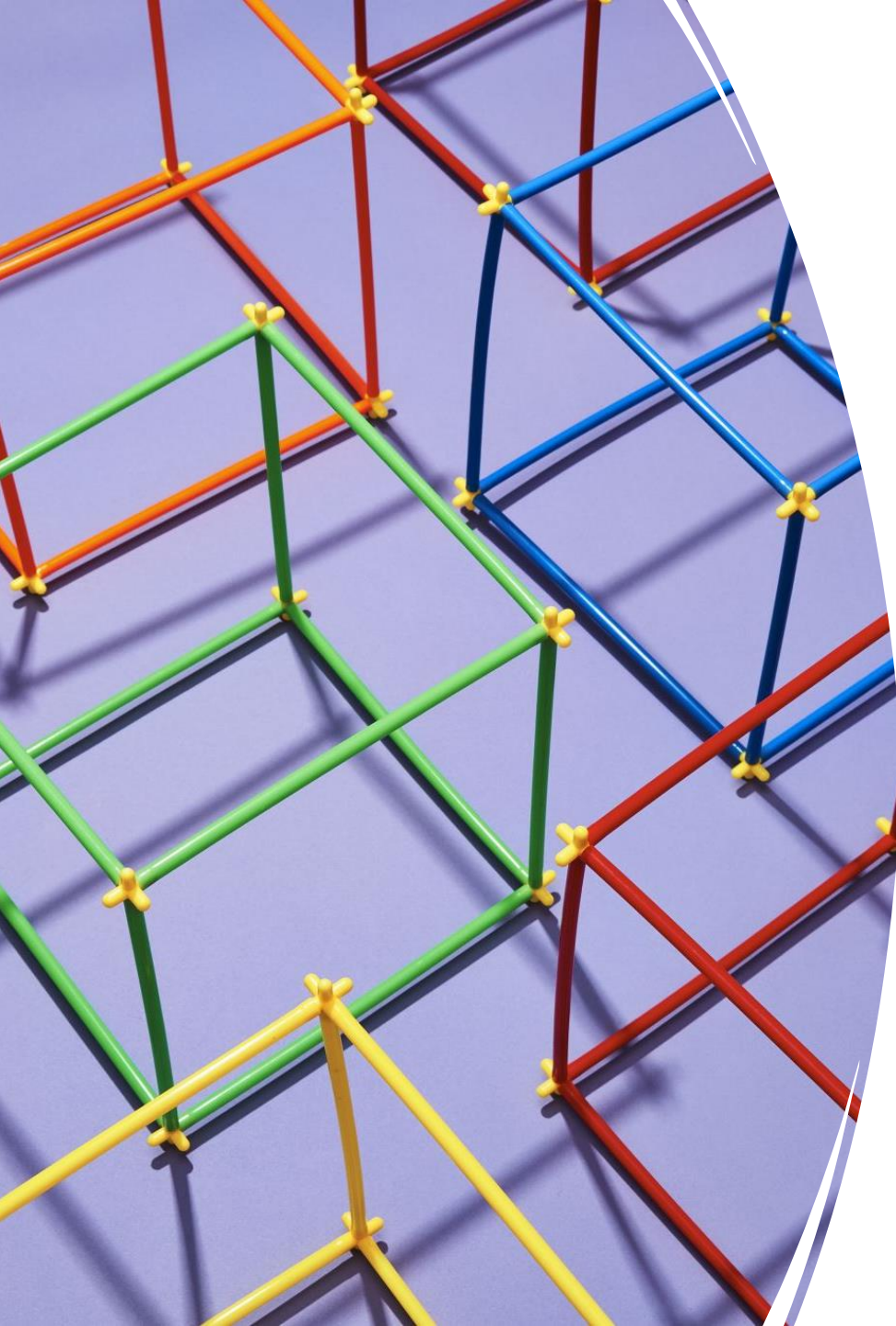


## Soporte Móvil



The background features a series of overlapping, wavy, ribbon-like shapes that flow from left to right. The colors transition from a bright yellow on the left, through orange and red, to a deep purple on the right. The shapes have a soft, glowing appearance with subtle gradients and shadows, giving them a three-dimensional feel. The word "Arquitectura" is centered over the middle of these waves.

Arquitectura



# Los 7 principales bloques de construcción

---

1. Componentes
2. Plantillas
3. Metadatos
4. Enlace de datos (Data Binding)
5. Directivas
6. Servicios
7. Inyección de Dependencias

# Bloques #1, 2 & 3:

## Componentes, Plantillas, Metadatos

- El Componente Angular es una clase, decorada con `@Component`
- El Componente controla la vista (la interfaz de usuario)

- El metadato es la información adicional que se agrega a una clase utilizando decoradores.
- Proporcionan información y configuración sobre la clase, como la forma en que se debe compilar, cómo se debe instanciar y cómo se debe comportar en tiempo de ejecución.

En el contexto de Angular, los metadatos se utilizan principalmente en los decoradores `@Component`, `@Directive`, `@Pipe` y `@Injectable`

```
1
2 import { Component } from '@angular/core';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.css']
8 })
9 export class AppComponent {
10   title = 'GettingStarted';
11 }
12
```

# Components

- [Angular Services](#) provide services to our Component
  - like fetching data from database using Task Service, logging application events using logger Services and making an HTTP request to the back-end server using HTTP service.
- The Responsibility to provide the instance of the Service to the Components falls on [Angular Injector](#).
  - It injects services into the components using [Dependency Injection](#)
- We also have [Directives](#), which help us to manipulate the structure ([structural directives](#)) or style ([attribute directive](#)) our application. The directives help us to transform the DOM as per our needs

# Para crear un componente...

ng generate component nombre\_del\_componente

```
import Component {} from '@angular/core';
@Component({
  selector: 'app-component-name',
  templateUrl: './component-name.component.html',
  styleUrls: ['./component-name.component.css']
})
export class ComponentNameComponent {
  // Component properties and methods go here
}
```



# Bloque #4: Enlace de Datos

## Interpolación, enlace de propiedades y eventos



### Interpolación:

La interpolación es una forma de enlazar los datos que permite mostrar datos dinámicos dentro de las plantillas Angular.

Se indica con llaves anguladas: (`{{ }}`). Dentro de ellas, puedes poner expresiones o variables que se evaluarán y sustituirán en el HTML por ese valor:

- `<h1>Hola, {{ usuario }}!</h1>`
- *: Data is bound from component to View*



### Enlace de Propiedades

Permite enlazar una propiedad de un elemento HTML a un valor o expresión en el componente. Se representa con el símbolo de corchetes: (`[]`).

Así se pueden variar dinámicamente los valores de esas propiedades HTML con valores del componente:

- `<button [disabled]="isDisabled">Clícame</button>`
- *Data is bound from component to the property of an HTML control in the view*



### Event Binding:

El enlace de eventos permite enlazar los eventos HTML con métodos del componente.

- `<button (click)="handleClick()">Clícame</button>`
- *The DOM Events are bound from View to a method in the Component*

## Two way binding in Angular



## Bloque #5: Directivas

- Las directivas Angular nos permiten manipular la DOM
- Son clases que creamos con el decorador @Directive.
- Contienen los metadatos y la lógica para manipular la DOM
- Puedes cambiar la apariencia, comportamiento, o la disposición de los elementos DOM con las directivas. De hecho "extienden" el HTML
  - Componentes: Directivas con una plantilla
  - Directivas Estructurales (\*nfor, \*ngif, \*ngrepeat, \*ngswitch ...)
  - Directivas de Atributos
    - ngModel [Para conseguir el [two-way data binding](#)]
    - ngClass [añaden o eliminan clases CSS de un elemento HTML]
    - ngStyle [cambian las propiedades de estilo de los elementos HTML]
  - Directivas personalizadas
    - Igual que si creáramos un componente Angular, cambiando el decorador @Component por @Directive

## Bloque #6: Servicios

- Angular no tiene una definición específica para los servicios
- Un Servicio es un trozo de Código focalizado con un objetivo concreto que se usará en muchos componentes de la aplicación
- Simplemente, creas una clase, exportas un método que haga algo específico y ya tienes un servicio.
- La magia de Angular es que es capaz de inyectar esos servicios a sus componentes.



Para crear un servicio...

```
ng generate service service-name
```

```
import Component from '@angular/core';
import ServiceNameService from './service-name.service';
@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  constructor private serviceName: ServiceNameService) { }
  // Access the service's properties and methods
  ngOnInit() {
    this.serviceName someMethod();
    console log this.serviceName.someProperty);
  }
}
```

# Uso de los Servicios

1. **Code Reusability:** Services allow you to write reusable code that can be used by multiple components. By encapsulating common functionality in a service, you can avoid code duplication and improve maintainability.
2. **Data Sharing:** Services provide a way to share data between components. They can act as a central store or data source that components can interact with. This allows components to communicate and exchange data without having a direct dependency on each other.
3. **Separation of Concerns:** Services help separate business logic, data access, and other non-UI-related tasks from the components. This promotes a more modular and maintainable codebase, as components can focus on presentation and user interaction, while services handle the underlying functionality.
4. **Dependency Injection:** Angular leverages dependency injection (DI) to provide instances of services to the components that require them. This allows components to be loosely coupled with services, making it easier to replace or modify dependencies. Services can be injected into components, other services, or even other services.
5. **Testing:** Services can be easily tested in isolation since they encapsulate specific functionalities. Unit testing services ensures that the underlying business logic or data access is working correctly without the need for UI interactions.

## Bloque #7: Inyección de Dependencias

- La [Inyección de Dependencias](#) es un método por el cual una nueva instancia de un servicio se inyecta a un componente (o pipe, directive, servicio, etc) que lo necesita.
- Consumidores del servicio declaran las dependencias que necesitarán en su constructor
- El inyector leerá las dependencias del constructor del consumidor.
  - Buscará esa dependencia en el suministrador de Angular
    - El proveedor la suministrará la instancia y el inyector la inyecta al consumidor
- El Inyector es un decorador que debe ser añadido al consumidor de la dependencia
  - Este decorador le indica a Angular que debe inyectar los argumentos del constructor a través de su sistema de inyección de dependencias



# Angular Dependency Injection

Angular dependency injection is now the core part of [Angular](#). It allows us to inject dependencies into the [Component](#), [Directives](#), [Pipes](#), or [Services](#).

**Dependency Injection (DI)** is a technique in which a class receives its dependencies from external sources rather than creating them itself.



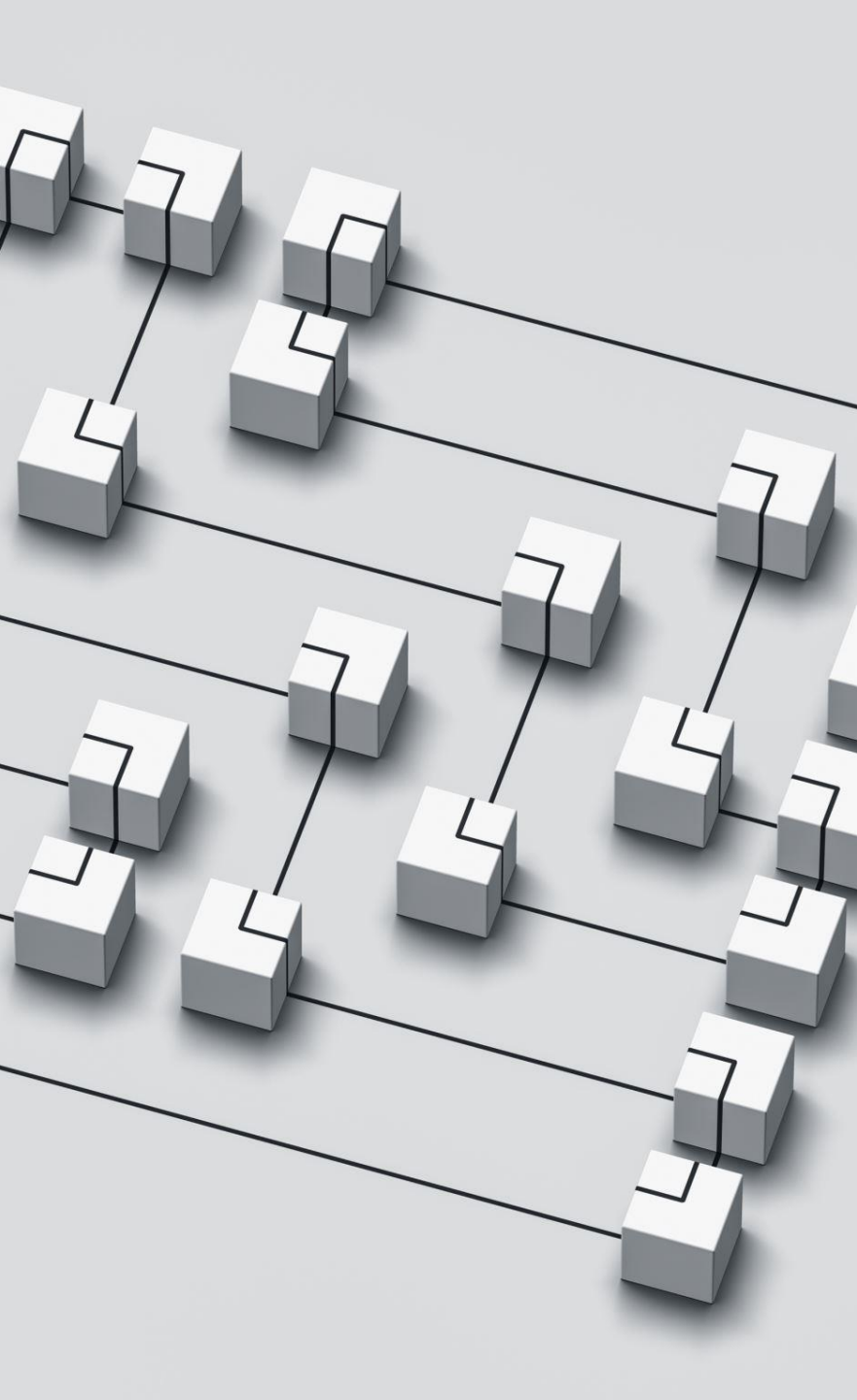
# Para inyectar un servicio a un componente en Angular

- Identifica el componente donde quieres usar el servicio
- En su fichero .component.ts importa el servicio
  - `import { ServiceNameService } from './service-name.service';`
  - Añade un constructor que incluya como parámetro al servicio.
    - `constructor(private serviceName: ServiceNameService) { }`

```
import { Component } from '@angular/core';
import { ServiceNameService } from './service-name.service';
@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  constructor(private serviceName: ServiceNameService) { }
  // Access the service's properties and methods
  ngOnInit() {
    this.serviceName someMethod();
    console.log this.serviceName.someProperty);
  }
}
```

# Beneficios de la inyección de dependencias

1. Desacoplamiento de componentes: La inyección de dependencias ayuda a desacoplar los componentes entre sí y de las implementaciones concretas de las dependencias. En lugar de que un componente cree sus propias dependencias, las recibe desde una fuente externa. Esto facilita la modificación y el reemplazo de las dependencias sin afectar el componente que las utiliza.
2. Reutilización de código: Al inyectar dependencias en lugar de crear instancias directamente en un componente, se promueve la reutilización de código. Las dependencias pueden ser compartidas y utilizadas en varios componentes, lo que evita la duplicación de código y mejora la mantenibilidad.
3. Testing más fácil: La inyección de dependencias facilita la escritura de pruebas unitarias para los componentes. Al separar las dependencias del componente y proporcionar interfaces claras, se pueden simular fácilmente las dependencias en las pruebas para aislar y probar el comportamiento específico del componente.
4. Configuración flexible: La inyección de dependencias permite la configuración flexible de las dependencias en tiempo de ejecución. Puedes proporcionar diferentes implementaciones de una dependencia según el entorno o los requisitos específicos. Esto facilita la personalización y adaptación de la aplicación en diferentes escenarios.
5. Mantenibilidad y escalabilidad: Al utilizar la inyección de dependencias, se facilita el mantenimiento y la escalabilidad de la aplicación. Los componentes se vuelven más independientes y fáciles de entender, lo que simplifica la identificación y solución de problemas. Además, al agregar nuevas funcionalidades, se pueden agregar nuevas dependencias sin afectar la estructura existente de la aplicación.
6. Interoperabilidad: La inyección de dependencias promueve la interoperabilidad entre bibliotecas y módulos de terceros. Al utilizar interfaces comunes para las dependencias, se puede integrar fácilmente código externo y aprovechar las funcionalidades proporcionadas por otras bibliotecas.



# Y además Angular Modules

- Una Aplicación angular consiste de varios bloques como son *Componentes, Servicios, Directivas*.
  - Que vamos creando a medida que la aplicación crece
- Es posible organizarlos en [Módulos Angular](#).
- No confundirlos con módulos JavaScript. Los de Angular son propios del lenguaje
- Angular usa los módulos de JavaScript a través del TypeScript

# Angular is Single Page application (SPA)

- The only page shown to you is index.html.
  - Components are added and removed from it
- Contents of app.component.html are inserted into the location `<app-root></app-root>`.
- Because app-root is the selector we used for the AppComponent it is the first component that Angular loads, hence it is our Root Component.
- If the app.component.html refers to `<customer-list></customer-list>`, Angular will look for that component (for example CustomerListComponent) with selector customer-list and render it there. This component now becomes the child component of the AppComponent.
- The Template of CustomerListComponent may include more selectors. Angular is recursively going to look for them and render them. They will become child of CustomerListComponent.
- This will create a tree of components. You can refer to the tutorial on how to create a child component from our tutorial [Angular child component](#)



# Una SPA – single Page Application

- Aplicación web que se carga inicialmente como una sola página HTML
  - A medida que el usuario interactúa con la aplicación, los cambios de contenido y navegación se realizan de forma dinámica sin necesidad de recargar completamente la página.
  - En lugar de cargar páginas HTML separadas, una SPA carga los recursos necesarios (como datos, vistas y scripts) de forma asíncrona a medida que el usuario navega por la aplicación.



Arquitectura

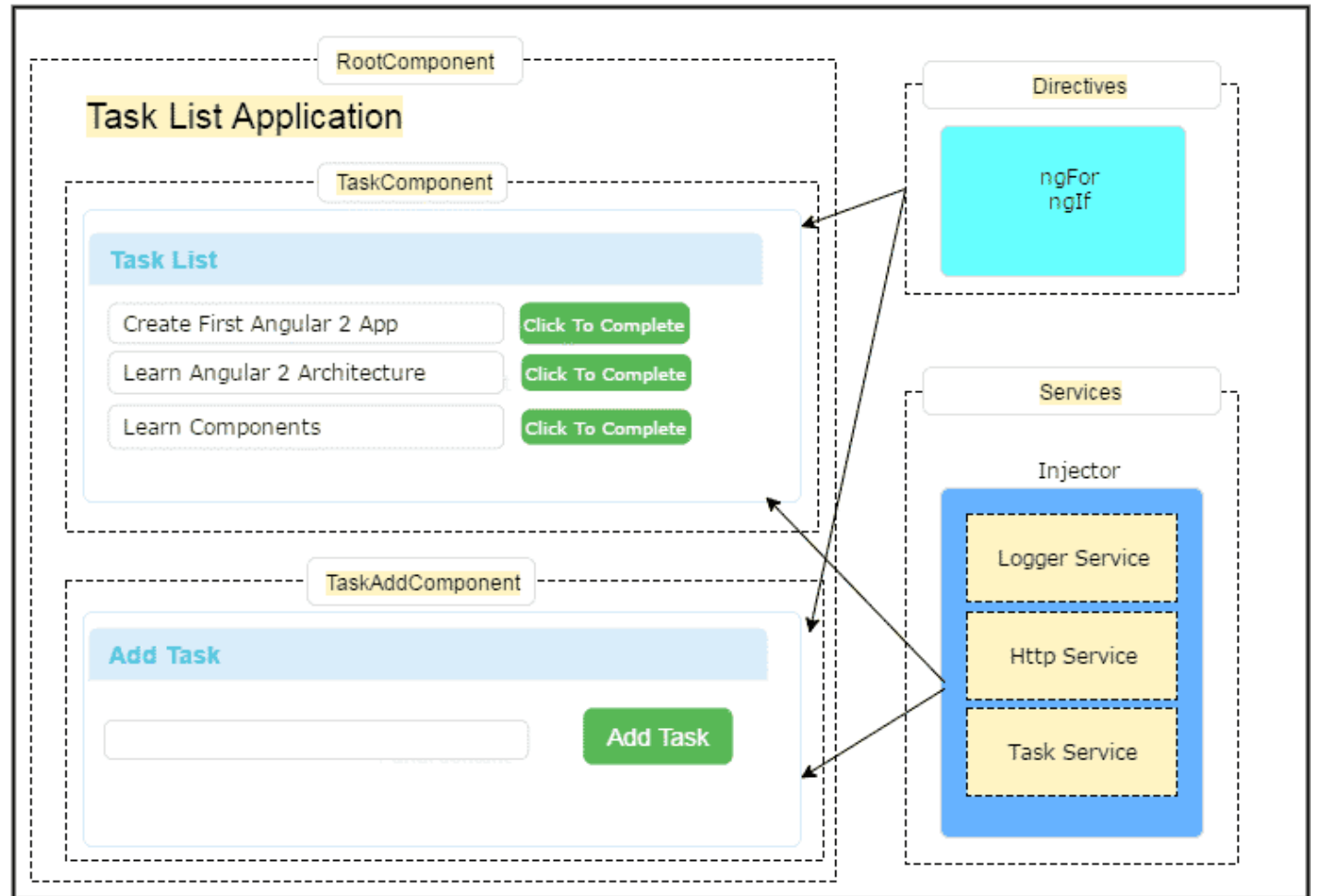


# Arquitectura de una aplicación Angular

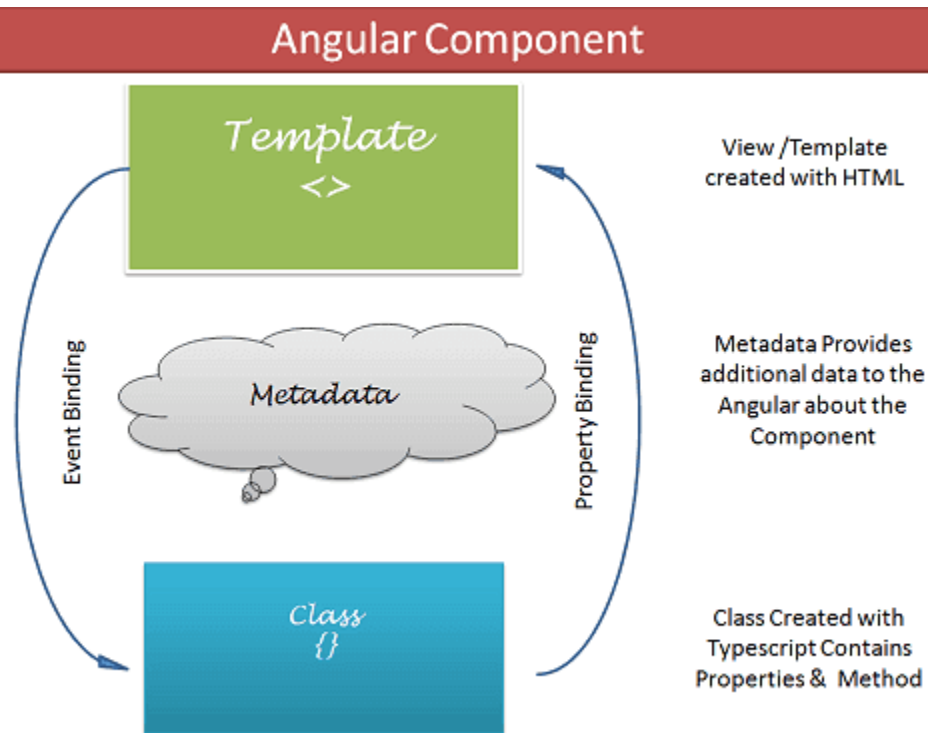
- La arquitectura de una aplicación Angular se basa en la idea de los componentes.
- Una aplicación Angular empieza con el componente en el nivel más alto: el componente raíz
  - Se añaden componentes "hijos" a una estructura en árbol

En este ejemplo tenemos 3 componentes.

- En el primer nivel está el rootComponent.
- Debajo de éste, tenemos dos mas:
  1. TaskComponent, que muestra la lista de tareas
  2. TaskAddComponent, que nos permite crear nuevas tareas



# Componentes



- Un Componente contiene la definición de la Vista y los datos que definen cómo se ve y cómo se comporta la vista
- Los Componentes de Angular son clases JavaScript definidas con el decorador `@component`
- Este decorador provee al componente con la vista a mostrar (plantilla HTML) y metadatos de la clase
- Los Componentes usan [data binding](#) para enviar los datos del componente a su vista (Plantilla HTML) [Usando la sintáxis [Angular Template Syntax](#) ]
  - El envío se consigue enlazando los elementos de la DOM con las propiedades del componente.
  -



# Cómo crear un componente:

- En src ir a *app.component.ts*

```
1
2 import { Component } from '@angular/core';
1
2 @Component({
3 })
1
2 @Component({
3   selector: 'app-root',
4   templateUrl: './app.component.html',
5   styleUrls: ['./app.component.css']
6 })
7
```

# Creando la plantilla

- En src ir a *app.component.html*

```
2 <h1>
3   Welcome to {{title}}!
4 </h1>
5
```

Al mostrar la vista, Angular busca la propiedad **title** en nuestro componente y enlaza esa propiedad con la vista.

```
1
2 <!--The content below is only a placeholder and can be replaced.-->
3 <div style="text-align:center">
4   <h1>
5     Welcome to {{title}}!
6   </h1>
7   Tour of Heroes</a></h2>
13   </li>
14   <li>
15     <h2><a target="_blank" href="https://github.com/angular/angular-cli/wiki">CLI Docum
16   </li>
17   <li>
18     <h2><a target="_blank" href="https://blog.angular.io/">Angular blog</a></h2>
19   </li>
20 </ul>
21
```

# Crear el fichero css

El siguiente paso es  
añadir los estilos  
CSS.

El metadato styleUrls  
le dice a Angular  
donde encontrar el  
fichero CSS

En este caso esta  
propiedad apunta al  
fichero externo  
app.component.css

# Activando el componente...

- En cada componente definimos

```
2 @Component({  
3   selector: 'app-root',  
4
```

Cada componente de Angular es un nuevo element de HTML.

Su nombre se define en la propiedad selector de los metadatos del componente.

Y... a usarlo en HTML

- Angular busca al selector en la vista actual.
- Si lo encuentra, muestra la vista del componente asociado con el selector en esa ubicacion.

- Y llamamos a nuestro componente simplemente con un:

- `<app-root></app-root>`

```
1  
2 <!doctype html>  
3 <html lang="en">  
4   <body>  
5     <app-root></app-root>  
6   </body>  
7 </html>  
8  
9
```

```
1
2 import { BrowserModule } from
3 '@angular/platform-browser';
4 import { NgModule } from '@angular/core';
5
6 import { AppComponent } from
7 './app.component';
8
9 @NgModule({
10   declarations: [
11     AppComponent
12   ],
13   imports: [
14     BrowserModule
15   ],
16   providers: [],
17   bootstrap: [AppComponent]
18 })
  export class AppModule { }
```

## Registrar el modulo

- Los módulos se registran en app.module.ts
  - Un Módulo Angular Module abarca los componentes, directivas, pipas y servicios que están relacionados y los convierte a bloques funcionales
  - El decorador @NgModule define un Módulo Angular Module y provee metadatos de los Módulos.
  - En los arrays se incluyen los componentes, pipas y directives que forman el módulo.
- Se indican todos los otros módulos utilizados en el array de imports.
- Y todos los Servicios en el de providers.
- Por último, el componente que Angular debe cargar al arrancar app.module.ts se indica en la propiedad bootstrap

# El routing en Angular

- El enrutamiento (routing) en una aplicación Angular se refiere a la capacidad de navegar entre diferentes componentes o vistas dentro de la aplicación en función de las URL y las interacciones del usuario.
- Es esencial para construir aplicaciones de una sola página (SPA) y proporciona una experiencia de usuario fluida al permitir la carga y visualización dinámica de contenido.
- Las rutas se definen en un archivo de configuración llamado `app-routing.module.ts`.
  - Generado automáticamente por el Angular CLI al crear un nuevo proyecto o componente.
- En el archivo `app.component.html`, donde se encuentra el componente raíz de la aplicación, debes agregar la directiva `router-outlet` que muestra el contenido del componente correspondiente a la ruta actualmente activa.

```
<router-outlet></router-outlet>
```



# Navegando entre las rutas

- Se puede utilizar la directiva routerLink en los enlaces o programáticamente a través del servicio Router.

- Por ejemplo, para crear un enlace a la ruta 'about', puedes hacer lo siguiente:

```
<a routerLink="/about">About</a>
```

- O:

```
import { Router } from '@angular/router';  
constructor private router: Router) {  
  navigateToAbout() {  
    this.router.navigate(['/about']);  
  }  
}
```

# Peticiones Http con HttpClient

- `import { HttpClient } from '@angular/common/http';`
- En tu servicio inyecta el HttpClient:
  - `constructor private http: HttpClient) { }`
- Y utilízalo.

```
this.http.get('https://api.example.com/data').subscribe(  
  (response) => {  
    // Manipula la respuesta exitosa aquí  
    console.log(response);  
  }, (error) => {  
    // Manipula el error aquí  
    console.error(error);  
  });
```

<https://angular.io/start>

