E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

# Elaboración de una guía pormenorizada / teórico-práctica para la migración y reingeniería de aplicaciones basadas en el Framework Google AngularJS al Framework Angular.

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

Autor: Iñigo Rezusta Iglesias
Director: César Arriaga Egüés
Pamplona, 02/07/2021

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

# Índice

# Índice de figuras



# Palabras clave

- Framework
- SPA (Single Page Application)
- FrontEnd
- JavaScript
- AngularJS
- Angular
- Migración

# Resumen

El nivel de interactividad y usabilidad requerido por las aplicaciones Web actuales, así como la necesidad de aplicación de las mejores prácticas de desarrollo, hace indispensable el uso de soluciones y Frameworks orientados a resolver el nivel Frontend de la arquitectura de este tipo de aplicaciones bajo el paradigma de desarrollo *Single Page Application*.

Plataformas como Facebook, Youtube, o Netflix están construidas con este tipo de tecnología, por lo que su popularidad y uso no hace más que aumentar en la medida que se confirma su idoneidad.

En este trabajo se propone el estudio y análisis en profundidad del Framework JavaScript por excelencia entre los años 2010 y 2016: AngularJS; que a pesar de contar con una alta implantación fue abandonado por el fabricante (Google) en pos de una reingeniería completa del mismo en la que se mantenía el paradigma Framework y el nombre (Angular) pero rompiendo toda compatibilidad hacia atrás.

En este trabajo se realiza un análisis exhaustivo de AngularJS y Angular que queda completado con una guía de migración con una implementación práctica entre ambos Frameworks para poder actualizar cualquier proyecto construido AngularJS, Framework actualmente obsoleto y descontinuado por Google.

# **Conceptos del dominio del problema:** desarrollo de aplicaciones web modernas.

*1. Lenguajes de programación para el desarrollo web:*

- **HTML**

**HyperText Markup Language (HTML)** es un lenguaje de marcado (confundir con lenguaje de programación) que **define la estructura de una página web** y sus contenidos. Emplea etiquetas de encierre para modificar el comportamiento o estética de partes del contenido.

HTML es el estándar a cargo del World Wide Web Consortium (W3C) que todos los navegadores han adoptado para la visualización de páginas web. Es un lenguaje que ofrece una gran estructuración lógica, adaptabilidad y facilidad de interpretación.

- **JavaScript**

**JavaScript** es un lenguaje de programación web interpretado basado en prototipos (orientación a objetos en clases que no se definen explícitamente, o lo que es lo mismo, creación de objetos sin definir la clase previamente) imperativo, débilmente tipado y dinámico.

Comúnmente se emplea en el lado del cliente o FrontEnd, siendo interpretado por el navegador. JavaScript **dota a las aplicaciones web de dinamismo y mejoras a nivel de interfaz**. También puede emplearse en el lado del servidor o BackEnd (Node JS), aunque en este caso no nos interesa dicha aplicación.

JavaScript sigue el estándar ECMAScript (lenguaje de scripting). Desde 2012 la totalidad de los navegadores web soportan completamente ECMAScript 5.1, y por lo consiguiente son capaces de interpretar código JavaScript.

- **TypeScript**

**TypeScript** es un lenguaje de programación web construido sobre JavaScript que extiende su sintaxis y **añade tipado estático y objetos** basados en clases. Se trata de un lenguaje compilado que se transpila a JavaScript y es interpretado como código JavaScript en el lado del navegador.

Ofrece multitud de ventajas respecto a su predecesor, siendo la captura de errores en tiempo de compilación, integración con herramientas de edición de texto, optimización de código o tipado algunas de las más destacables.

### 1. Tipos Básicos

```
const nombre = 'Nombre'   // JavaScript
```

```
const nombre: string = 'Nombre'   // TypeScript
```

### 2. Funciones

```
function suma (a, b) {
  return a + b
}
```

```
function suma (a: number, b: number): number {
  return a + b
}
```

### 3. Interfaces

```
const alumnos = [
  {nombre: 'Ejemplo1', apellido: 'Ejemplo1Ap}
]
```

```
interface Alumno {
  nombre: string
  apellido: string
}
const alumnos: Alumno[] = [
  {nombre: 'Ejemplo1', apellido: 'Ejemplo1Ap
]
```

### 4. Clases

```
class Alumno {
  constructor(nombre) {
    this.nombre = nombre
  }
}
```

```
class Alumno {
  nombre: string
  constructor(nombre) {
    this.nombre = nombre
  }
}
```

- **CSS**

**Cascading Style Sheets (CSS)**: Comúnmente conocido como un lenguaje de diseño gráfico, CSS es el lenguaje empleado para **describir la apariencia visual, presentación y renderizado** de documentos HTML.

Está diseñado para poder separar el contenido del documento de la representación visual. Esta separación nos dota de facilidad de mantenimiento y actualización de estilos, consistencia a la hora de implementar un diseño global, facilidad en adaptación multi-dispositivo, beneficios SEO, mejoras en la accesibilidad...

Junto con HTML y JavaScript constituye la base de la programación web.

*2. Conceptos básicos en el desarrollo web:*

- **SPA**

Single Page Application: Aplicación web contenida en una sóla página cargada al inicio (recursos HTML, JavaScript y CSS), que **va cargando datos e información de forma dinámica** (utilizando AJAX - comunicación asíncrona), normalmente como respuesta a las interacciones del usuario con la aplicación.

Este paradigma ofrece gran velocidad al no requerir carga de ficheros durante la ejecución. Además, se realiza un cacheo eficiente de todos los recursos en el navegador, por la experiencia del usuario en cuanto a velocidad, dinamismo y potencia es muy positiva.

En la otra cara de la moneda, al no disponer de información estática, encontramos problemas de SEO; la carga inicial puede ser costosa y se pueden ocasionar problemas de seguridad (XSS Cross-Site Scripting). No obstante, en un futuro próximo veremos cómo estas desventajas se van poco a poco desvaneciendo, principalmente por el impulso que grandes empresas como Google o FaceBook están dando a esta tecnología en la actualidad.

- **FrontEnd**

En el desarrollo web se denomina FrontEnd a la **parte de la aplicación que interactúa con los usuarios**, también conocida como Client-Side o lado del cliente. El FrontEnd, o **capa de presentación**, Incluye las tecnologías de diseño, desarrollo, testing o debug que se ejecutan en el lado del navegador y que permiten la interacción de los usuarios con la capa de datos o BackEnd.

A modo de resumen podemos afirmar que el FrontEnd es todo lo que el usuario ve o con lo que interactúa en una aplicación web. Es el responsable de la apariencia total de la aplicación y de la experiencia que el usuario tiene en ella.

- **Framework Software**

Un **Framework Software** es un marco de trabajo o estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que puede servir de base para la organización, simplificación y desarrollo de cualquier tarea software. Puede incluir soporte de programas, bibliotecas o soporte de lenguajes de programación, entre otras herramientas, para así ayudar a desarrollar e integrar componentes de un proyecto.

Esta definición, algo compleja, podría resumirse como el entorno pensado para hacer más sencilla la programación de cualquier aplicación o herramienta actual.

Hoy en día encontramos multitud de frameworks que facilitan el desarrollo software y que podemos agrupar en torno a distintas temáticas:

- Frameworks para aplicaciones web: Nos centraremos en ellos posteriormente.
- Frameworks de aplicaciones: .NET Framework
- Frameworks de tecnología AJAX
- Frameworks de gestión de contenidos: WordPress
- Frameworks multimedia

- **Aplicación web**

Denominamos aplicación web a un **programa o software que corre en un servidor web al que se accede a través de un navegador** que actúa como un cliente *ligero*, independiente del sistema operativo. Son aplicaciones que, por lo general, no requieren de una instalación en el equipo cliente, como sí puede hacerlo una aplicación de escritorio.

A modo de resumen: en una aplicación web el cliente recibe servicios a través del navegador desde un servidor alojado por un tercero.

- **MVC**

MVC o Modelo Vista Controlador es un estilo de arquitectura software que separa datos de la aplicación, interfaz de usuario y lógica en 3 componentes distintos: modelo, vista y controlador.

Este patrón se basa en ideas de reutilización de código y separación de conceptos. Estas características posibilitan el encapsulamiento del código por funcionalidad, y por lo tanto una mayor facilidad de desarrollo y mantenimiento.

# AngularJS: framework para el desarrollo de aplicaciones web.


## Introducción

Con la llegada de JavaScript en 1966 se abre una nueva era en el desarrollo web, que previamente estaba formada por contenido puramente estático.

*"The web became not just a place to read things, but to do things."*

Esta revolución pone en el punto de mira a JavaScript, cuya popularidad aumenta notablemente al dotar de dinamismo al contenido web. Los desarrolladores agrupan sus funcionalidades en **librerías** que comparten y reutilizan en distintos proyectos, enriqueciendo cada vez más la web.

Como evolución natural de este proceso de empaquetamiento surgen Frameworks para el desarrollo de aplicaciones JavaScript:

Un Framework JavaScript es una librería que nos ofrece una "guía" sobre cómo construir aplicaciones web. Esta guía otorga previsibilidad y homogeneidad a las aplicaciones, características que permiten potenciar la **escalabilidad, mantenimiento y longevidad** del software. Por ello, es factible afirmar que **los frameworks de JavaScript son una parte esencial del desarrollo web front-end moderno**, ya que proporcionan a los desarrolladores herramientas para crear aplicaciones web interactivas y escalables.

En este marco de Frameworks JavaScript para desarrollo FrontEnd surge en el año 2009 **AngularJS**, cuando Misko Hevery (empleado de Google) y Adam Aborns desarrollan un proyecto para simplificar la creación de aplicaciones web.

La primera versión estable de AngularJS (versión 0.9.0) fue lanzada en octubre de 2010 bajo licencia MIT. La versión 1.0 AngularJS se lanzó en junio de 2012, cuando el Framework ya había alcanzado una gran popularidad dentro de la comunidad de desarrollo web. Desarrolladores y diseñadores podían ahora trabajar juntos para crear SPAs potentes de gran tamaño.

Respaldada e impulsada por Google, AngularJS adquiere gran importancia y se sitúa como el **Framework JavaScript más popular a partir del año 2013.** En el año 2016, sitios web tan importantes como PayPal, YouTube o Netflix se crean utilizando AngularJS.


Figura: Aplicaciones desarrolladas con AngularJS

## AngularJS - ¿Qué es?

*"**AngularJS** (comúnmente llamado **Angular.js** o **AngularJS 1**, no confundir con Angular), es un framework de JavaScript de código abierto, mantenido por Google, que se utiliza para crear y mantener aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles."* (Wikipedia)

AngularJS extiende el lenguaje HTML a través de la creación de **nuevas estructuras HTML** o etiquetas personalizadas, **denominadas directivas.** De esta forma es posible dinamizar el contenido de las vistas a través de un enlace bidireccional que permite la **sincronización automática e inmediata de modelos y vistas**. Como resultado, AngularJS pone menos énfasis en la manipulación del DOM y mejora la testeabilidad y el rendimiento.

En AngularJS el trabajo de relleno de plantillas HTML con datos se traslada del lado del servidor al lado del cliente. El resultado es un sistema estructurado para **actualizaciones de página dinámicas y rápidas**.

El enlace de datos de AngularJS y la inyección de dependencias eliminan gran parte del código que hasta el momento venía siendo necesario escribir. Además, todo esto sucede en el navegador, **liberando notablemente la carga de trabajo en el lado del servidor**.

AngularJS simplifica el desarrollo de aplicaciones al presentar un **mayor nivel de abstracción** al desarrollador. Incluye de forma predeterminada todo lo que necesita para crear una aplicación CRUD en un conjunto cohesivo: vinculación de datos, directivas básicas, validación de formularios, enrutamiento, enlaces profundos, componentes reutilizables e inyección de dependencias, testeabilidad.

Algunos de los puntos clave que brinda AngularJS son:

- **Desacopla la manipulación DOM de la lógica** de la aplicación: Mejora drásticamente la capacidad de prueba del código.

- **Desacopla el lado del cliente de una aplicación del lado del servidor**: Permite que el trabajo de desarrollo avance en paralelo y permite la reutilización de ambos lados.

- **Guía a los desarrolladores** a través de todo el proceso de creación de una aplicación: desde el diseño de la interfaz de usuario, pasando por la escritura de la lógica de negocio, hasta las pruebas.

## Claves del éxito

La enorme popularidad alcanzada por AngularJS se debe en su mayor parte a varios factores:

1. **Reducción de código**: Se reduce entre un 80-90% las líneas de código respecto a su predecesor jQuery, lo que se traduce directamente en un ahorro de tiempo y costes de desarrollo, debugeo y mantenimiento.

2. **Two Way Binding**: El "Two Way Binding" o "enlace bidireccional" representa la sincronización entre los modelos de datos y las vistas de la aplicación.

   Cuando el modelo de datos cambia, los cambios se ven reflejados en la vista; Cuando el usuario modifica los datos en la vista (por ejemplo a través de un input), estos cambios se ven reflejados en el modelo de datos, de ahí el *"two way"* binding.

   Esta sincronización se produce de manera automática e instantánea, siendo un importante avance respecto al manejo de datos que se venía haciendo previamente con jQuery.

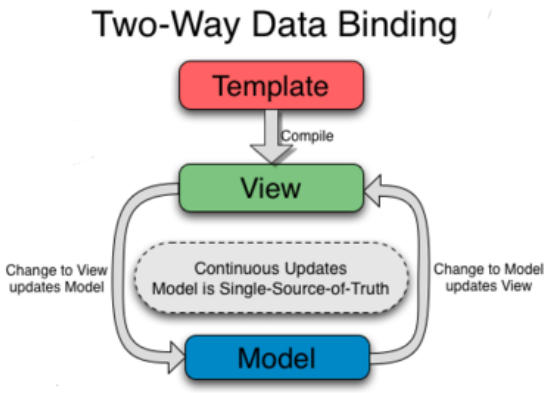



Figura: Binding bilateral: El modelo modifica la vista y viceversa

3. **Solución cohesiva**: La mayor parte de piezas de una solución web de tipo FrontEnd son manejadas consistentemente por AngularJS (enrutamiento, sincronización vistas-modelo de datos, comunicación con servidor, accesibilidad…) sin necesidad de recurrir a librerías de terceros.

   Además, AngularJS proporciona un patrón de desarrollo que hace que aplicaciones totalmente distintas sean similares a nivel de arquitectura código, siendo éstas mucho más entendibles. Este aspecto inevitablemente derivará en una reducción de

tiempos y costes cuando se produzcan cambios en los equipos de desarrollo y sea necesario enfrentarse a código desconocido.

4. **Accesibilidad e internacionalización**: El equipo de AngularJS realizó un importante esfuerzo en facilitar la accesibilidad de las aplicaciones a través de directivas específicas, así como para facilitar la internacionalización de las aplicaciones en lo que a números, fechas y cambios de moneda se refiere.

5. **Popularidad**: Al ser el Framework FrontEnd por excelencia resultaba sencillo aprender AngularJS al existir una enorme cantidad de proyectos de ejemplos, blogs, webs y tutoriales.

6. **MVC**: AngularJS se basa en el **patrón MVC**, consistente en una separación de código entre: vista, modelo y controlador. Este patrón de desarrollo ofrece grandes ventajas, como son: mayor velocidad de desarrollo, facilita la colaboración entre desarrolladores, mayor facilidad de actualización, mayor facilidad de debug y mantenimiento.

7. **Testeabilidad**: AngularJS ofrece excelentes herramientas tanto para el testeo unitario de componentes como para el testeo funcional de la aplicación.

8. **Liberación de tareas**: con AngularJS se libera al desarrollador de labores como:

    - **Registro de callbacks**: eliminar el código repetitivo, como las devoluciones de llamada, reduce enormemente la cantidad de codificación de JavaScript necesaria y facilita ver lo que realmente hace una aplicación.

    - **Manipular el DOM mediante programación**: al describir de forma declarativa cómo debería cambiar la interfaz de usuario a medida que cambia el estado de la aplicación, se libera de las tareas de manipulación de DOM de bajo nivel.

    - **Clasificación de datos hacia y desde la interfaz de usuario**: las operaciones CRUD constituyen la mayoría de las tareas de las aplicaciones AJAX. El flujo de datos entre servidor - objeto interno - formulario HTML crea una gran cantidad de código repetido. AngularJS describe el flujo general de la aplicación en lugar de todos los detalles de implementación, eliminando la mayor parte de código repetitivo.

    - **Escribir mucho código de inicialización para comenzar**: por lo general, es necesario escribir muchas tuberías para hacer funcionar una aplicación básica AJAX. AngularJS permite comenzar a desarrollar funciones rápidamente.

# Arquitectura

## Compilador HTML

El compilador HTML de AngularJS permite que el **navegador sea capaz de interpretar una nueva sintaxis HTML**. Permite asociar un comportamiento específico a cualquier elemento o atributo HTML e incluso crear nuevos elementos o atributos HTML con comportamiento personalizado (directivas).

Toda esta compilación **se realiza en el navegador o cliente**. No existe una pre-compilación, ni se involucra al servidor en este proceso. El compilador es un servicio AngularJS que atraviesa el DOM en busca de directivas. El proceso de compilación ocurre en tres fases.

1. Se atraviesa el DOM y se recopilan todas las directivas. Si el compilador encuentra que un elemento coincide con una directiva, entonces la directiva se agrega a la lista de directivas que coinciden con el elemento DOM.
2. Una vez que se han identificado todas las directivas que coinciden con un elemento DOM, el compilador clasifica las directivas según su prioridad.
   **Se ejecutan las funciones de compilación de cada directiva**. Cada función de compilación tiene la posibilidad de modificar el DOM. Cada función de compilación devuelve una función de enlace. Estas funciones se componen en una **función de enlace combinada**, que **invoca la función de enlace devuelta de cada directiva**.
3. El compilador vincula la plantilla con el $scope llamando a la función de vinculación combinada. Esto, a su vez, llamará a la función de enlace de las directivas individuales, registrando a los oyentes en los elementos y configurando $watchs con el $scope según se configura cada directiva

En la práctica, el compilador realiza la siguiente búsqueda:

***Atributo ng-App:*** En la carga inicial, AngularJS procesa el árbol HTML de arriba a abajo, buscando el elemento del DOM con el atributo *ng-app*. Una vez encuentra el atributo, AngularJS deduce que ese elemento del DOM y sus elementos hijos serán los que formen parte de la aplicación.

***Atributo ng-Controller:*** Una vez encontrado el atributo ng-App, Angular busca el atributo ng-controller. Este atributo indica a AngularJS que esa parte del DOM será controlada por un controlador cuyo nombre viene especificado en el atributo. AngularJS buscará el componente, lo cargará, y continuará procesando el HTML.

***Directivas específicas****:* Una vez identificado el controlador, que será el encargado de la ejecución y control de todos los eventos posteriores, AngularJS procesa el resto del HTML identificando las directivas que vinculan vista y controlador.

## Detección de cambios

La comparación de dos objetos en JavaScript es una operación extremadamente rápida. AngularJS hace uso de esta funcionalidad en el llamado ***Dirty Checking***.

AngularJS cachea una copia de los datos de la aplicación y observa todos los eventos que pueden hacer cambiar los datos del modelo. Cuando se produce uno de estos eventos se lanza el proceso ***Digest***, que no es más que una comparación del modelo de datos actual con la copia de datos cacheada (operación extremadamente rápida).

Si AngularJS detecta cambios en la información, se vuelven a renderizar los datos, de forma que **la vista siempre está actualizada** sin necesidad de actualizarla manualmente con setters, como se venía haciendo en la programación tradicional.

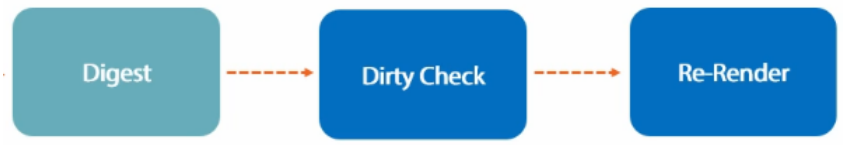



Figura: detección de cambios AngularJS

### Inyección de dependencias

El concepto "Inyección de dependencias" surge de la necesidad de hacer más sencillo el uso de los componentes que AngularJS incluye, y de los componentes personalizados que desarrollan los usuarios.

En el siguiente ejemplo se observan dos funciones JavaScript en las que los parámetros que se pasan en la llamada son componentes (*$http* es un componente incluido en AngularJS, *calculadora* es un componente desarrollado por un usuario cualquiera):

```
function obtenerDatosDeAPI($http) {};
function calcularCosteDeJuego(calculadora){};
```

A modo de resumen, AngularJS nos permite registrar componentes y emplearlos posteriormente en cualquier función, pasando siempre el nombre del componente como parámetro.

## Técnicas de construcción

A continuación se listan los conceptos técnicos más característicos de AngularJS y posteriormente se detallan los más importantes:

| Concepto | Descripción |
|---|---|
| **Plantilla** | HTML con marcado adicional. |
| **Directivas** | Extender HTML con atributos y elementos personalizados. |
| **Modelo** | Datos que se muestran al usuario en la vista y con los que el usuario interactúa. |
| **Scope** | Contexto donde se almacena el modelo para que los controladores, directivas y expresiones puedan acceder a él. |
| **Compilador** | Analiza la plantilla y crea instancias de directivas y expresiones. |
| **Data Binding** | Sincronización automática e inmediata de datos entre el modelo y la vista. |
| **Controlador** | La lógica de negocio tras las vistas. |
| **Inyección de dependencias** | Crea y conecta objetos y funciones. |
| **Módulo** | Contenedor para las diferentes partes de una aplicación, incluidos controladores, servicios, filtros... |
| **Servicio** | Lógica de negocio reutilizable independiente de las vistas. |

### Controlador

El **controlador** en AngularJS no es más que un objeto JavaScript que **contiene la lógica de negocio** de la aplicación necesaria para la representación de las vistas. Su objetivo es gestionar el flujo de la parte cliente. Contiene variables y métodos necesarios para implementar la funcionalidad asociada a la presentación. Corresponde a la parte "controlador" del paradigma MVC.

```
// Ejemplo Controlador.js

var app = angular.module('Demo',[]);

app.controller('ControladorDemo', function($scope) {
       $scope.variablePrueba = "Variable de prueba";
       $scope.funcionPrueba = function() {
            $scope.variablePrueba = "Otro valor";
      };
});
```

Se ha registrado el controlador *ControladorDemo* en el módulo con nombre *Demo*. Dentro del controlador se ha creado una variable (`variablePrueba`) y una función (`funcionPrueba`), que serán accesibles e invocables desde la vista.

### Binding

El **binding** es la forma en la que AngularJS **sincroniza el modelo y la vista**. Cuando el modelo de datos cambia, los cambios se ven reflejados en la vista; cuando el usuario modifica los datos en la vista, estos cambios se ven reflejados en el modelo de datos, de ahí el "*two way*" binding. AngularJS realiza esta operación de forma automática e inmediata.

```
// Ejemplo vista.html con binding en ambos sentidos

<html ng-app="Demo">
   <body ng-controller="ControladorDemo">
      <div>{{ variable }}</div>
      <input ng-model="variable">
   </body>
</html>
```

- Binding Modelo → Vista: El contenido entre los brackets (`{{ variable }}`) es sustituido por el valor de la variable en el controlador. **Si el modelo cambia, la vista cambia.**

- Binding Vista → Modelo: La variable indicada en la directiva ng-model (`ng-model="variable"`) se actualiza con el valor introducido desde la vista. **Si la vista cambia, el modelo cambia.**

### $Scope

La variable ***$scope*** es un contenedor de datos que vincula y sincroniza **variables y funciones entre la vista y el controlador**.


Figura: vínculo entre vista y controlador

```
// Ejemplo Controlador.js

var app = angular.module('Demo',[]);

app.controller('ControladorDemo', function($scope) {
       $scope.título = "Título de prueba";
});
```

Registramos la variable *título* en el `$scope`. Esta variable será ahora accesible desde la vista.

### Directivas

Las **directivas** no son más que elementos DOM (etiquetas HTML personalizadas) que indican al compilador de AngularJS que se debe vincular un comportamiento especial al elemento DOM, o que se debe transformar el elemento DOM y sus hijos. AngularJS incluye directivas propias, aunque también es posible crear directivas personalizadas.

*Ejemplos:*

1. **ng-app**: Directiva de arranque de la aplicación.

```
<div ng-app="myApp"></div>
```

2. **ng-model**: Asocia una propiedad del modelo a la vista.

```
<input type="text" ng-model="propiedad">
```

3. **ng-repeat**: Repite un elemento HTML:

```
<li ng-repeat="empleado in empleados">{{empleado.nombre}}</li>
```

Directivas personalizadas/propias:

```
// HTML
<user-info></user-info>
```

```
// Javascript
(function() {

  angular.module('Demo')

  .directive('informacionEmpleado', [function() {
    return {
      restrict: 'E',
      template:'Nombre: {{empleado.nombre}}, email: <a
href="mailto:{{empleado.email}}">{{empleado.email}}</a>'
    };
  }]);

}());
```

Esta directiva será reemplazada en tiempo de compilación por el HTML definido en la template (plantilla). Las directivas personalizadas tienen varias ventajas respecto a un desarrollo convencional:

- Se pueden desarrollar componentes reutilizables.
- Limpieza y claridad en el código HTML.
- Encapsulamiento.

### Otros elementos

**- Servicio:** Un servicio en AngularJS no es más que un objeto o controlador que contiene lógica de negocio, con el objetivo de ser reutilizable por distintos componentes.

AngularJS ofrece servicios propios, como $http para el manejo de peticiones http, y también permite al usuario crear sus propios servicios.

**- Filtros:** AngularJS permite formatear el valor de una expresión para su representación al usuario a través de filtros propios o customizados.

## Obsolescencia

Factores que llevaron a AngularJS a la extinción

1. **Curva de aprendizaje**

La arquitectura de Angular sigue el patrón MVC (Modelo - Vista - Controlador), más concretamente el patrón MVVM (Modelo - Vista - Modelo de Vista). **Este patrón tiene una curva de aprendizaje de muy lenta evolución** en programadores sin experiencia previa.

A esto hay que sumar que algunos de los **conceptos claves** de su ecosistema **no están claramente definidos** incluso en la documentación oficial (p.e. Existen múltiples formas de utilizar controladores o de inyectar dependencias).

Por otro lado, **el funcionamiento del núcleo de AngularJS también resulta confuso**. El binding y gestión de eventos se gestiona con una versión personalizada del addEventListener (no con el predeterminado del navegador). Esta versión personalizada hace que el orden de actualizaciones del modelo sea difícil de razonar, y que la integración con aplicaciones de terceros requiera de implementaciones específicas de gestión de eventos.

2. **MVC**

**El futuro de la web son los componentes web.** La arquitectura de Angular (MVC) está basada en el uso de directivas, controladores y la variable $scope. A pesar de que en la versión 1.5 **AngularJS incorpora componentes**, su funcionamiento no es óptimo**.**

```
// Ejemplo de uso de componente en AngularJS

<componente setting="{{expresion}}"></componente>
```

En el arranque de la aplicación, y antes de que AngularJS llegue a cargarse, las expresiones de Angular se pasan directamente al componente como si se tratase de un componente web común (p.e. <img>), lo que ocasiona errores.

Para evitarlo se emplea el uso de atributos *ng-*, por lo que **el concepto de componente web se adapta a la arquitectura de AngularJS y no al revés, como debería**.

3. **Javascript:** lenguaje **interpretado no tipado**

**AngularJS utiliza Javascript** como lenguaje de programación. Javascript es un lenguaje interpretado por el navegador. **No incluye tipado estático**, y los **errores** en el código sólo se detectan **en tiempo de ejecución**, lo que complica enormemente el desarrollo en aplicaciones de medio/gran tamaño.

*"Types Provide a Conceptual Framework for the Programmer"*

Un buen diseño se trata de interfaces bien definidas. Y es mucho más fácil expresar la idea de una interfaz en un lenguaje que los admita.

4. **Rendimiento**

El rendimiento de las aplicaciones AngularJS en ocasiones se ve afectado debido a dos factores:

- **Demasiados Bindings o enlaces**: en la detección de cambios AngularJS escanea **recursivamente un árbol completo** de objetos sobre código no procesado u optimizado, lo que deriva en problemas de rendimiento.
- **Dependencias entre módulos**: los módulos no se cargan de forma asíncrona en función de sus dependencias la primera vez que se necesita una dependencia, sino que se cargan todas ellas a la vez.

5. **SEO**

AngularJS es un Framework de desarrollo de aplicaciones web de tipo **SPA** (Single Page Application).

Este tipo de aplicaciones suponen un **problema en cuanto a su posicionamiento** en buscadores (SEO), ya que **no disponen de contenido HTML estático enviado desde el servidor**. El HTML no presenta el contenido del sitio, sino que generalmente será JavaScript el que reciba el contenido y genere las vistas dinámicamente en el cliente.

Cuando un motor de búsqueda accede a la SPA sólo podrá obtener un marcado muy elemental, sin nada de contenido y mucho JavaScript.

## Actualidad

Debido a los puntos comentados en el punto anterior, y ante el surgimiento de nuevos Frameworks que no sólo evitaban esos problemas, sino que además introducían novedades técnicas y mejoras de rendimiento, **AngularJS comienza a perder importancia rápidamente** en el panorama del desarrollo web.

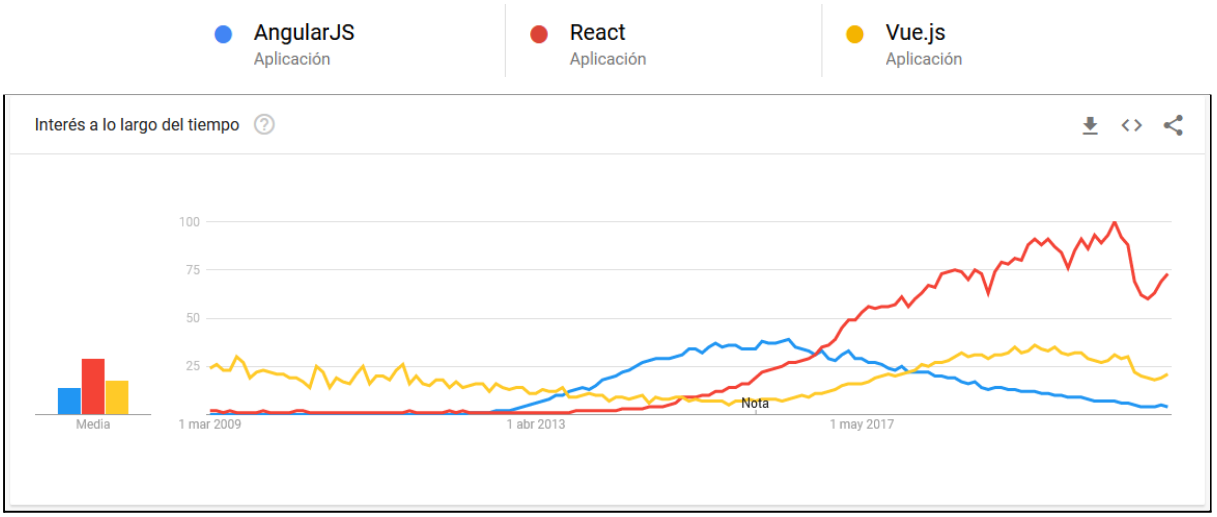



Figura: interés de búsqueda en Google de frameworks Javascript

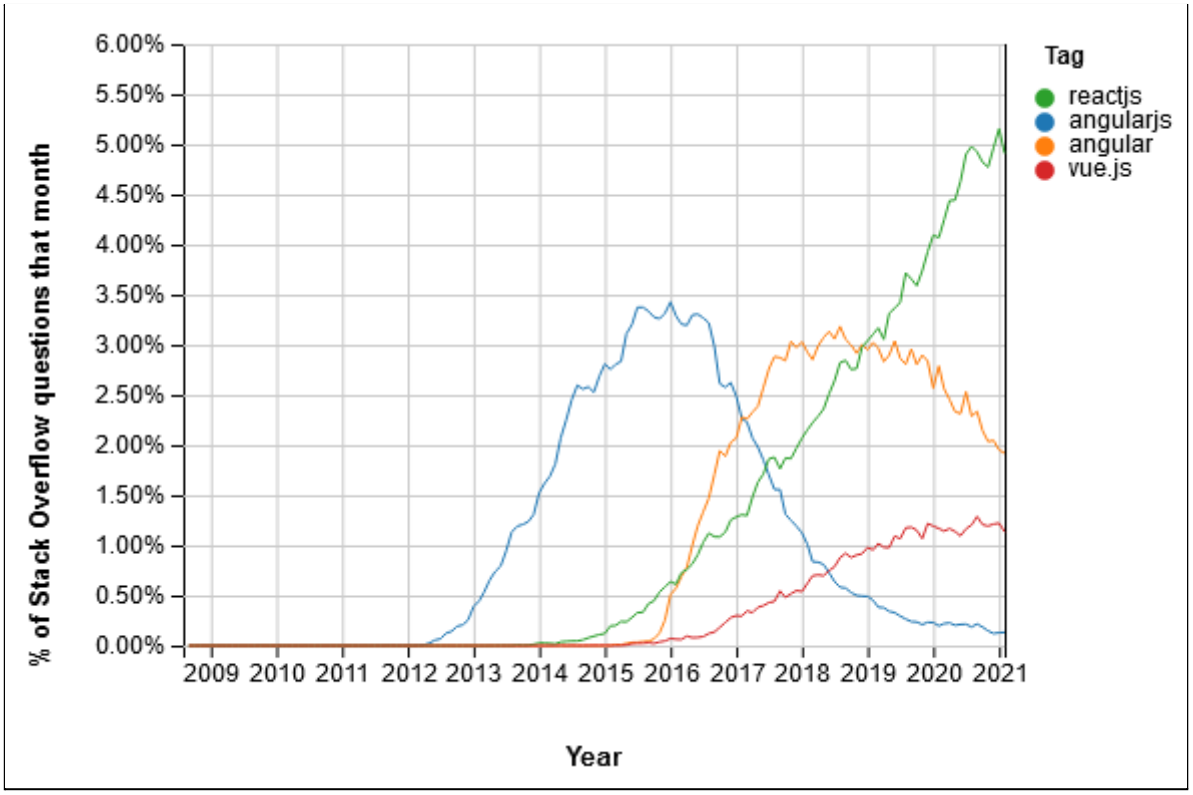



Figura: porcentaje de preguntas por año de frameworks Javascript en StackOverflow

Ante esta situación, Google apuesta por otro framework no continuista (Angular), y el 1 de Julio de 2018 anuncia que AngularJS entraría en un periodo de Long Term Support[1] hasta el **1 de Julio de 2021**, fecha en la que **el Framework dejaría de recibir soporte**, quedando expuestas todas las aplicaciones que hiciesen uso de esta tecnología.

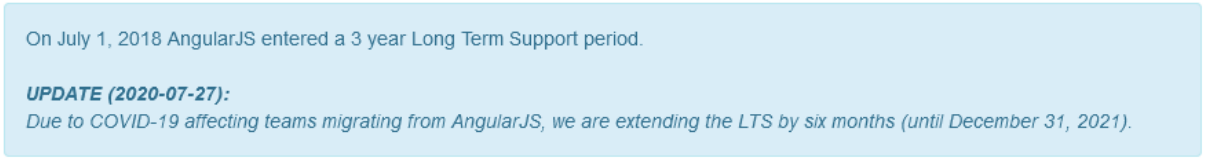



Figura: anuncio de Google de prórroga de periodo LTS de AngularJS

Debido a la situación del Covid-19, Google decidió **ampliar el plazo de LTS hasta el 31 de Diciembre de 2021,** al afectar a posibles migraciones en curso.

[1] Long Term Support: periodo el equipo de desarrollo se centra **exclusivamente en solucionar bugs críticos** de la última versión que puedan afectar a la seguridad, compatibilidad con nuevas versiones de jQuery y/o compatibilidad con navegadores.

# Angular: alternativa natural a AngularJS


## Motivación

El **14 de septiembre de 2016** se lanza una nueva versión de AngularJS conocida como **Angular 2** (a partir de ahora: Angular). Los objetivos de la nueva versión podrían agruparse en los siguientes:

- Razonamiento más sencillo
- Mejorar el rendimiento
- Mayor modularidad
- Mejorar el sistema de inyección de dependencias
- Adoptar el patrón de componentes web
- Incorporar soporte nativo para móviles
- Incorporar soporte para renderizado en servidor (SEO)
- Mejorar la testeabilidad

Esta versión fue una **reescritura completa de la anterior, no compatible** con las versiones previas de AngularJS. Esta **decisión fue muy criticada** ya que AngularJS era un Framework ampliamente utilizado en el mundo de desarrollo web, y la discontinuidad del Framework supondría tener que rehacer o migrar los proyectos existentes a una nueva tecnología.

A continuación se exponen los principales **motivos por los que se decidió crear un nuevo Framework** y no se siguió una línea continuista con respecto al Framework ya existente:

1. **Razonamiento más sencillo**

Se modifica el **lenguaje de programación a Typescript**, lenguaje que permite el **tipado estático y el uso de clases**. Al tratarse de un **lenguaje compilado,** se recuperarán los errores en tiempo de ejecución.

Se **estandarizan y unifican mecanismos de creación de elementos** como los componentes, que previamente podían ser definidos de múltiples formas, haciendo la curva de aprendizaje mucho más pronunciada.

El equipo de Angular extrae del núcleo de Angular los **mecanismos de parcheado** de todos los puntos de interacción asincrónica (bindings, eventos). Los **unifica y los hace reutilizables para evitar confusiones** acerca del funcionamiento interno de los bindings.

2. **Mejorar el rendimiento y la modularidad:**

Se incorporan varios mecanismos de mejora:

- **Mecanismo de detección de cambios** (binding): se modifica para que la VM de Javascript pueda **optimizar el código en código nativo**. Con esto, en lugar de escanear recursivamente un árbol de objetos, se crea una función optimizada en el inicio de Angular para ver si el enlace ha cambiado.
- Se incorporan **sistemas que evitan escanear partes del árbol** en la detección de cambios (observable, inmutable).
- Se incorpora de forma nativa la carga diferida (**lazy load**) de dependencias entre módulos.

3. **Adoptar el patrón de componentes web**

Se modifica la sintaxis de las plantillas para evitar vincularse a atributos simples. En ningún lugar habrá expresiones de Angular en atributos simples (para evitar problemas de interoperabilidad con componentes web).

**La versión 2.0 trae tantos cambios importantes a nivel arquitectónico, manejo de canalización HTTP, ciclo de vida de la aplicación, administración del estado, etc, que transferir el código antiguo al nuevo era casi imposible: a pesar de mantener el nombre, la nueva versión Angular era un marco completamente nuevo con poco o nada en común con el anterior.**

## Angular - ¿Qué es?

*"Angular" es una plataforma de desarrollo de aplicaciones web construida sobre TypeScript. Como plataforma, Angular incluye:*

- *Un marco basado en componentes para crear aplicaciones web escalables*
- *Una colección de bibliotecas bien integradas que cubren una amplia variedad de características, que incluyen enrutamiento, administración de formularios, comunicación cliente-servidor y más.*
- *Un conjunto de herramientas para desarrolladores que lo ayudarán a desarrollar, compilar, probar y actualizar su código" [angular.io - what-is-angular]*

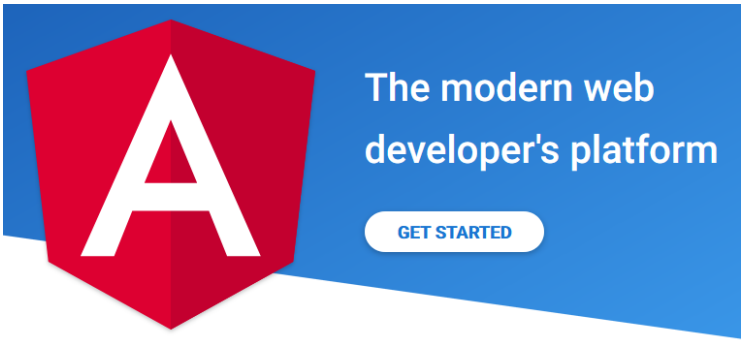



Figura: banner de Angular anunciado por Google

Las ideas clave que residen en el núcleo de Angular se pueden resumir en 3 apartados:

1. **Componentes**

Los componentes son el bloque de construcción principal en las aplicaciones Angular. Cada componente consta de:

- **Plantilla HTML:** representa la interfaz visual/vista del componente.
- **Selector CSS:** identifica el componente en las plantillas. Los elementos HTML que coinciden con el selector de un componente se convierten en instancias del componente.
- **Clase TypeScript:** define el comportamiento, actúa de controlador.
- **Estilos CSS:** que se aplican a la plantilla.

2. **Plantillas**

Angular extiende el lenguaje HTML permitiendo insertar valores dinámicos desde el componente: directivas. Angular actualiza automáticamente el DOM renderizado cuando el estado de un componente se modifica.

Al igual que en AngularJS, las directivas en Angular **permiten separar la lógica de su aplicación de su presentación**. Las plantillas se basan en HTML estándar, por lo que son fáciles de crear, mantener y actualizar.

### 3. Inyección de dependencias

La inyección de dependencias permite declarar dependencias de sus clases de TypeScript sin ocuparse de su instanciación, que es manejada exclusivamente por Angular. Este patrón de diseño permite escribir código más comprobable y flexible.

## Angular CLI

Una de las novedades que trae Angular es la creación de Angular Command Line Interface: una **herramienta de interfaz de línea de comandos** que se utiliza para inicializar, desarrollar, andamiaje y mantenimiento de aplicaciones de Angular directamente desde un shell de comandos.

Angular CLI facilita infinidad de tareas haciendo uso del comando *ng*, entre ellas:

- `ng build`: compila la aplicación Angular.
- `ng serve`: crea un servidor local de desarrollo y lo re-compila con cada cambio en el proyecto.
- `ng test`: ejecuta los test unitarios de un proyecto.
- `ng generate`: automatiza la generación de componentes.

## Arquitectura

La arquitectura de una aplicación Angular se basa en varios conceptos fundamentales. Los bloques de construcción **básicos del marco Angular son componentes Angular** organizados en módulos *ngModule*.

Los módulos de Angular recopilan el código relacionado en conjuntos funcionales; una aplicación Angular está definida por un conjunto de NgModules. **Cada aplicación tendrá siempre al menos un módulo raíz** que permite el arranque y, por lo general, tiene muchos más módulos de funciones.

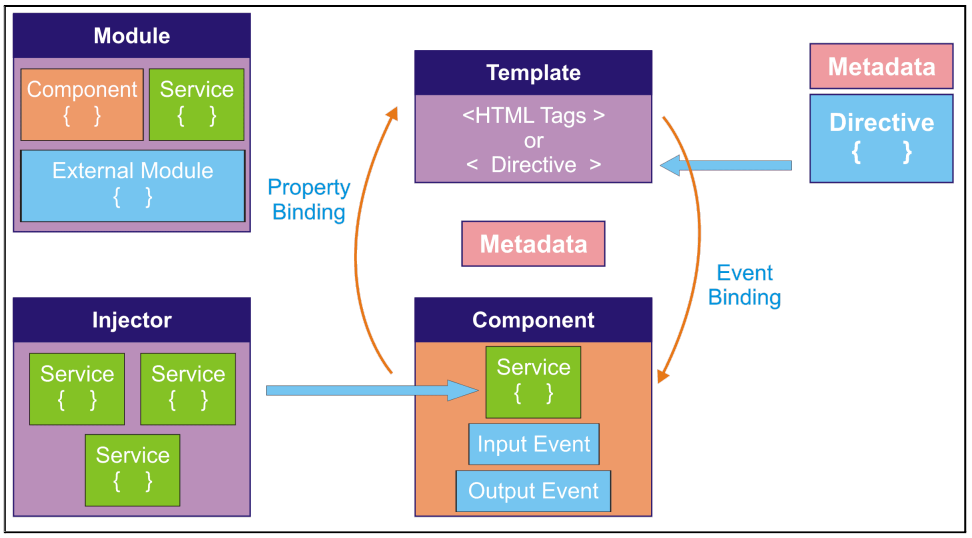



Figura: arquitectura de Angular

Los **componentes son el elemento principal** sobre el que se construyen las aplicaciones en Angular. Los componentes **definen vistas HTML**, vistas que se modifican de acuerdo con la lógica y los datos de la aplicación.

Los componentes utilizan **servicios que proporcionan una funcionalidad específica y limitada** que no está directamente relacionada con las vistas. Los proveedores de servicios pueden inyectarse en componentes como dependencias, lo que hace que el código sea modular, reutilizable y eficiente.

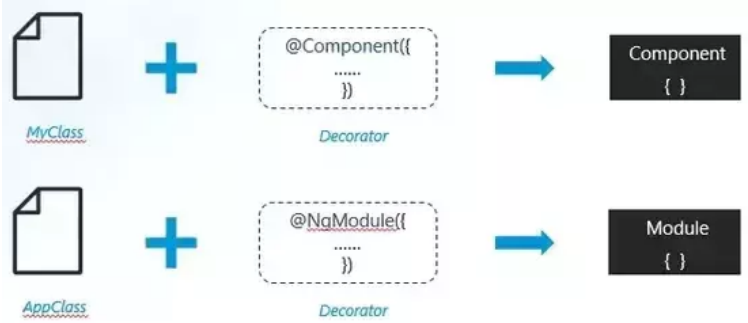



Figura: decoradores en Angular

**Los módulos, componentes y servicios son clases que utilizan decoradores**. Estos decoradores marcan su tipo y proporcionan metadatos que le dicen a Angular cómo usarlos.

- Los metadatos de una clase de componente la asocian con una plantilla que define una vista.
- Una plantilla combina HTML con directivas Angular y marcado vinculante que permiten a Angular modificar el HTML antes de renderizarlo para su visualización.
- Los metadatos para una clase de servicio proporcionan la información que Angular necesita para ponerla a disposición de los componentes a través de la inyección de dependencias.

## Técnicas de construcción

A continuación se listan los conceptos técnicos fundamentales en la construcción de cualquier aplicación Angular y posteriormente se detallan los más importantes:

| Concepto | Descripción |
|---|---|
| **Componentes** | Bloque de construcción principal que incluye la lógica de negocio de la aplicación. |
| **Plantillas** | Fragmento de HTML extendido que representa la vista de un componente. |
| **Directivas** | Clases que agregan comportamiento adicional a los elementos de las aplicaciones. |
| **Inyección de dependencias** | Patrón de diseño en el que una clase solicita dependencias de fuentes externas en lugar de crearlas. |
| **Data Binding** | Sincronización automática e inmediata de datos entre el modelo y la vista * |
| **Módulo** | Contenedor de diferentes partes de una aplicación. |
| **Servicio** | Lógica de negocio reutilizable independiente de las vistas. |

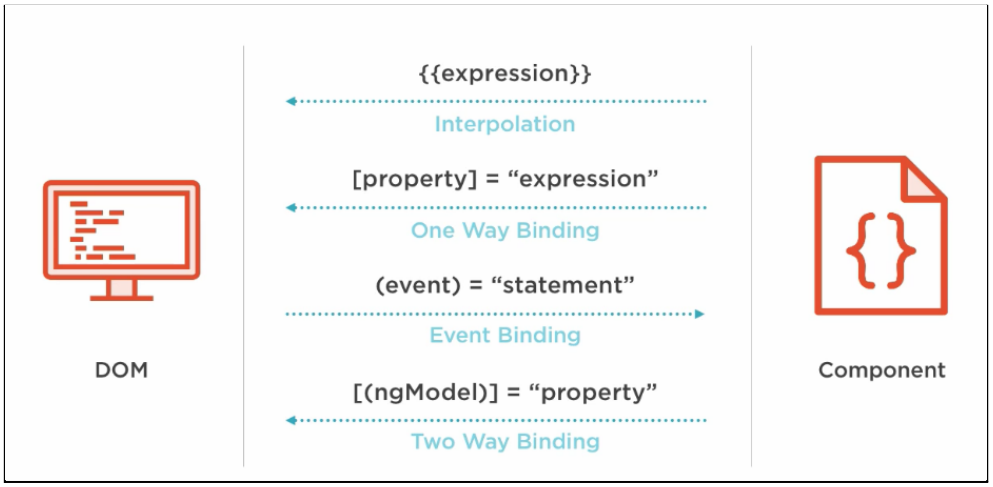



*Figura: funcionamiento del binding en Angular

### Componente

Un componente en Angular es una clase Typescript que **contiene la lógica de negocio** de una parte de la aplicación, que es **representada en la interfaz a través de la vista/plantilla HTML** correspondiente. **La clase interactúa con la vista** a través de una API de propiedades y métodos.

```
// Ejemplo de componente

@Component({
  selector: 'alumnos-listado',
  templateUrl: './alumnos-listado.component.html',
  styleUrls: ['./alumnos-listado.component.css']
})
export class AlumnosListado implements OnInit {

  alumnos: Alumno[];

  constructor(private service: ServicioAlumnos) { }

  ngOnInit() {
    this.alumnos = this.service.getAlumnos();
  }
}
```

El **decorador @Component** indica a Angular dónde se encuentra la vista correspondiente, el fichero de estilos asociados al componente, y el selector que se utilizará para inyectar el componente en otras vistas.

La propia clase del componente incorpora un **constructor en el que se inyectan las dependencias**. Los métodos y variables de la clase serán accesibles desde la vista.

### Plantilla

Una plantilla HTML Angular **representa una vista en el navegador** empleando una extensión de HTML que otorga mayor dinamismo y mucha más funcionalidad.

Angular **permite ampliar el vocabulario HTML** de sus aplicaciones. Por ejemplo, permite obtener y establecer valores DOM dinámicamente con características como funciones de plantilla integradas, variables, escucha de eventos y enlace de datos (binding).

Debido a que una plantilla angular es parte de una página web general no es necesario incluir elementos como <html>, <body> o <base>. Basta con **centrarse exclusivamente en la parte de la página** que está desarrollando.

```
// Ejemplo de plantilla

<h2>Alumnos</h2>
<div *ngFor="let alumno of alumnos">
  <h3>
    <a [title]="alumno.nombre + ' detalle">
      {{ alumno.nombre }} {{ alumno.apellido }}
    </a>
  </h3>
  <p *ngIf="alumno.asignatura">
    Description: {{ alumno.asignatura.descripcion }}
  </p>
  <button (click)="añadirAsignatura()">
    Añadir
  </button>
</div>
```

### Directiva

Las directivas son clases que extienden el comportamiento original de los elementos de las aplicaciones. Con las directivas integradas de Angular, puede administrar formularios, listas, estilos, interfaces...

Existen tres tipos de directivas:

- **Directivas de componente**

Los componentes en Angular son un subconjunto de directivas asociadas con una plantilla. A diferencia de otras directivas, sólo se puede crear una instancia de un componente para un elemento dado en una plantilla.

La sintaxis de la directiva viene definida en el decorador del componente (`selector: 'alumnos-listado'`) y permitirá inyectar la plantilla del componente allí donde se añada.

```
<h1>Listado de alumnos</h1>
<alumnos-listado></alumnos-listado>
```

En tiempo de ejecución la directiva `<alumnos-listado></alumnos-listado>` será sustituida por su plantilla correspondiente.

- **Directivas de atributo**

Las directivas de atributos escuchan y modifican el comportamiento de otros elementos, atributos, propiedades y componentes HTML.

```
<div [ngClass]="alumno == seleccionado ? 'claseSeleccion' : 'clasePredeterminada'">Div
de ejemplo</div>
```

*ngClass* permite aplicar una clase css u otra en función del resultado que arroje la expresión `alumno == seleccionado`

Angular ofrece multitud de directivas de atributo incorporadas, pero también es posible desarrollar directivas de atributo personalizadas.

- **Directivas estructurales**

Las Directivas Estructurales son directivas que cambian la estructura del DOM agregando o quitando elementos.

```
<div *ngIf="alumno == seleccionado">Detalle de alumno</div>
```

**ngIf* mostrará u ocultará el elemento div y sus hijos en función del resultado que arroje la expresión `alumno == seleccionado`

Al igual que sucede con las directivas de atributo, Angular ofrece multitud de directivas estructurales incorporadas, y también es posible desarrollar directivas estructurales personalizadas.

## Actualidad

Debido a su **enorme potencial y al respaldo de Google**, hoy en día Angular se encuentra en el **top 3 de Frameworks Javascript más populares** (2º posición en el momento) y sigue siendo el perfil más demandado del sector en portales de empleo junto con React.js:

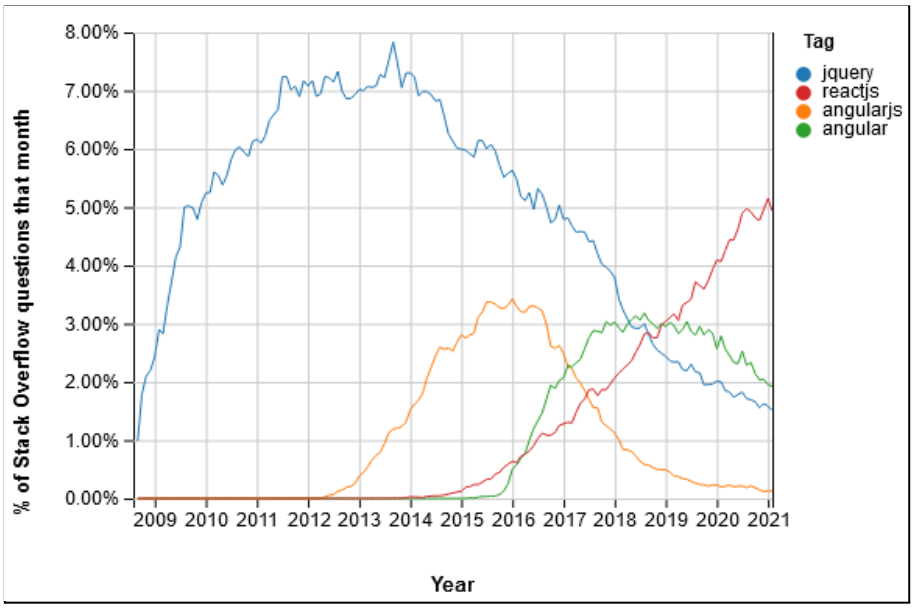



Figura: Porcentaje de preguntas por tema en StackOverflow (En verde Angular)

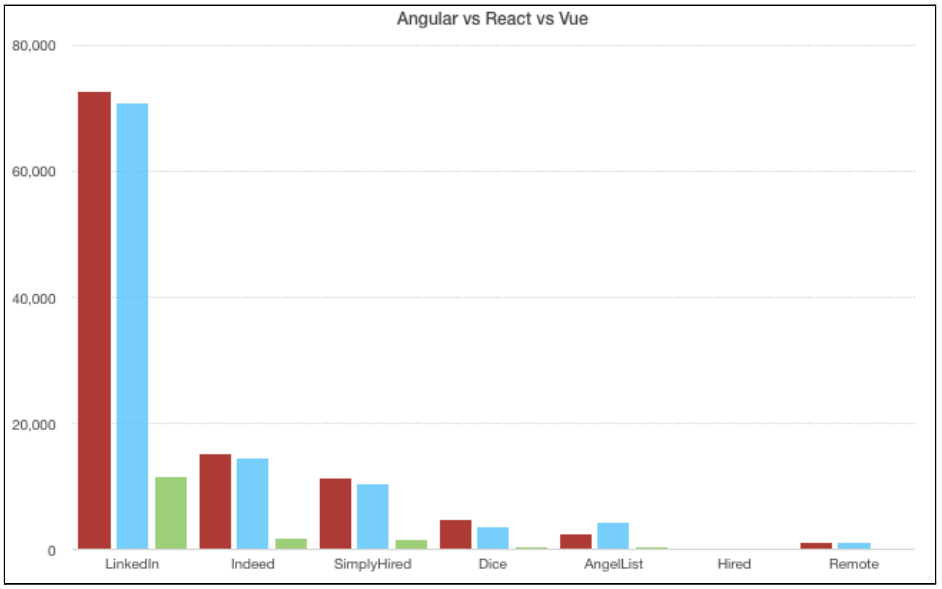



Figura: Ofertas de trabajo en distintas plataformas de contratación

| | stars | issues | updated | created | size |
|---|---|---|---|---|---|
| angular | 59.590 | 468 | Mar 9, 2021 | Jan 6, 2010 | Minified + gzip package size for angular in KB |
| react | 165.971 | 685 | Mar 26, 2021 | May 24, 2013 | Minified + gzip package size for react in KB |
| vue | 181.063 | 563 | Mar 26, 2021 | Jul 29, 2013 | Minified + gzip package size for vue in KB |

Figura: Número de descargas de Frameworks Javascript en NPM

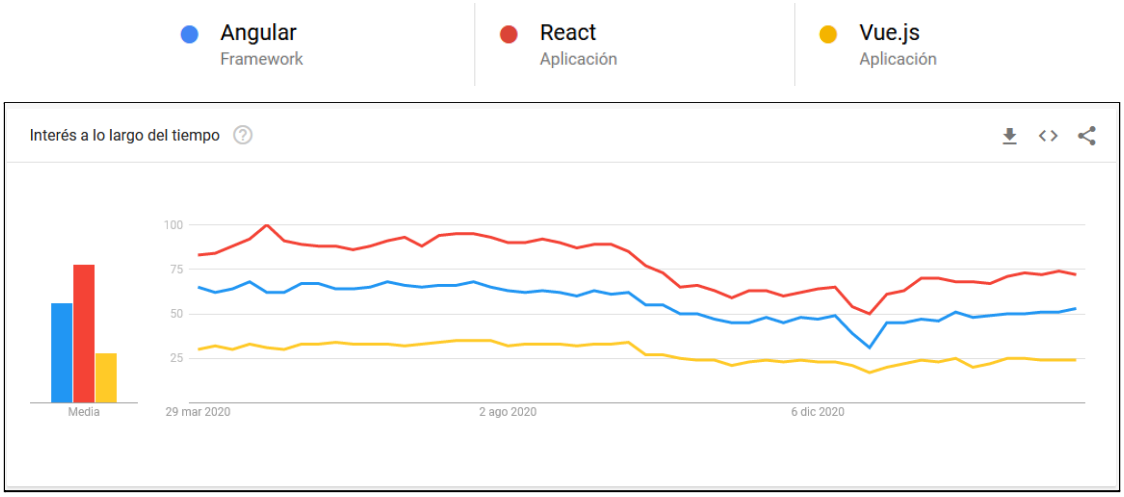



Figura: Búsquedas en Google en el último año de los Frameworks Javascript más populares

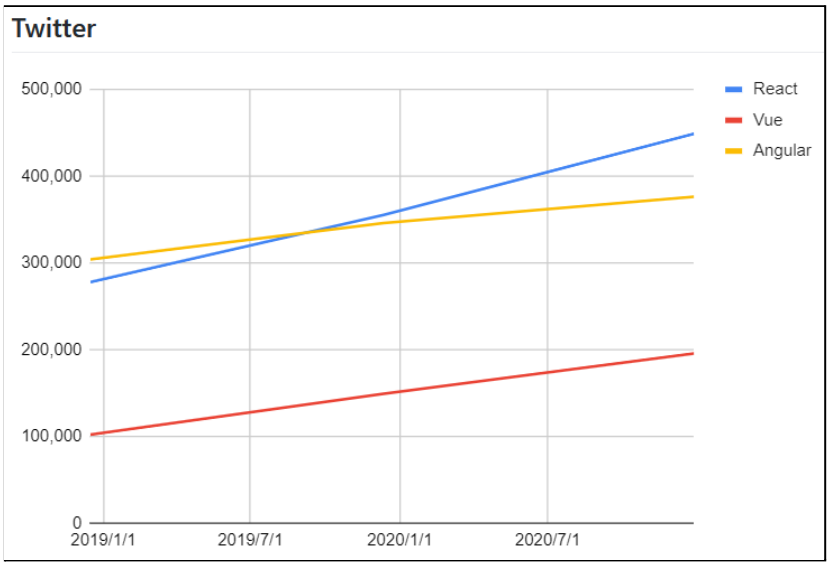



Figura: Número de seguidores en Twitter de los Frameworks Javascript más populares

El equipo de Angular sigue realizando mejoras y lanzando versiones con la siguiente periodicidad:

- ★ **Un lanzamiento importante cada 6 meses**.
- ★ 1-3 lanzamientos menores para cada lanzamiento principal.
- ★ Una versión de parche y una versión preliminar prácticamente cada semana.

Todas las versiones principales suelen tener soporte durante 18 meses: 6 meses de soporte activo, durante los cuales se lanzan actualizaciones y parches programados regularmente; 12 meses de soporte a largo plazo (LTS), durante los cuales solo se publican parches de seguridad y correcciones críticas.

| VERSION | STATUS | RELEASED | ACTIVE ENDS | LTS ENDS |
|---|---|---|---|---|
| ^11.0.0 | Active | Nov 11, 2020 | May 11, 2021 | May 11, 2022 |
| ^10.0.0 | LTS | Jun 24, 2020 | Dec 24, 2020 | Dec 24, 2021 |
| ^9.0.0 | LTS | Feb 06, 2020 | Aug 06, 2020 | Aug 06, 2021 |

Figura: Versiones de Angular a fecha Abril 2021

# Guía práctica de migración: AngularJS a Angular

En esta guía se describen los **pasos y herramientas necesarias para llevar a cabo una migración exitosa a Angular** de una aplicación desarrollada en AngularJS, minimizando los riesgos y maximizando la productividad y calidad del código.

Este proceso de migración constará de **dos fases claramente diferenciadas**:

## 1. Fase de preparación

Se adaptará la aplicación para que sea más fácil de mantener, que esté más desacoplada y mejor alineada con las herramientas de desarrollo modernas. Además de facilitar la migración, también **mejorará las aplicaciones AngularJS** existentes.

## 2. Fase de migración

Se realizará de forma híbrida e incremental, ejecutando los dos frameworks paralelamente en la misma aplicación y migrando los componentes de AngularJS a Angular uno a uno.

**Los conceptos teóricos de la migración vendrán representados con un ejemplo práctico de migración real a partir de una aplicación real desarrollada en AngularJS.** La aplicación pasará por varios estados durante todo el proceso. Existirá un repositorio por cada estado con el código fuente:

| Estado | Lenguaje | Repositorio |
|---|---|---|
| Original | AngularJS | https://github.com/rezusta/DemoAngularJS |
| Adaptada | AngularJS | https://github.com/rezusta/DemoReadyAngularJS |
| Híbrida | AngularJS + Angular | https://github.com/rezusta/DemoAngularHibrid |
| Migrada | Angular | https://github.com/rezusta/DemoAngularMigrated |

## Aplicación de ejemplo

*Aplicación para la gestión de alumnos y asignaturas.*

Se ha implementado una aplicación sencilla empleando AngularJS para el desarrollo de la parte FrontEnd. Se trata de una aplicación de **gestión de asignaturas y alumnos** sencilla que **cubre un amplio espectro de las funcionalidades** anteriormente descritas, y será la aplicación que se utilice como ejemplo para el proceso de migración.

- Enlace: https://demo-frontend-angularjs.herokuapp.com/

Los detalles técnicos como casos de uso, arquitectura o despliegue de la aplicación se detallan en el Anexo 1: Aplicación de Ejemplo - AngularJS.

---

### Estructura y componentes de la aplicación

A continuación se detalla la jerarquía de código empleada y los componentes desarrollados:

```
lib/
├─ controllers/
├─ services/
├─ views/
├─ app.js
├─ styles.css
index.html
```

- **Controladores**

| | |
|---|---|
| *ControladorAlumnos.js* | Implementa la lógica de la sección de alumnos (listado, detalle…) |
| *ControladorAsignaturas.js* | Implementa la lógica de la sección de asignaturas (listado, detalle…) |
| *ControladorSidenav.js* | Implementa la lógica del menú lateral |
| *MainControlador.js* | Implementa la lógica global |

- **Servicios**

| | |
|---|---|
| *ServicioAlumnos.js* | Realiza llamadas CRUD al backend de alumnos vía API REST HTTP |
| *ServicioAsignaturas.js* | Realiza llamadas CRUD al backend de asignaturas vía API REST HTTP |

- **Vistas**

| | |
|---|---|
| *crearAlumno.html* | Código HTML de la vista de creación de alumnos |
| *crearAsignatura.html* | Código HTML de la vista de creación de asignaturas |
| *detalleAlumno.html* | Código HTML de la vista de detalle de alumno |
| *detalleAsignatura.html* | Código HTML de la vista de detalle de asignaturas |
| *listadoAlumnos.html* | Código HTML de la vista del listado de alumnos |
| *listadoAsignaturas.html* | Código HTML de la vista del listado de asignaturas |

- **Enrutador**

| | |
|---|---|
| *app.js* | Vincula las URLs del navegador a vistas de la aplicación. |

- **Estilos**

| | |
|---|---|
| *styles.css* | Incluye los estilos CSS globales de la aplicación. |

- **Recursos**

| | |
|---|---|
| */assets* | Incluye los recursos estáticos de la aplicación (imágenes). |

De aquí en adelante todas las modificaciones se aplicarán sobre esta aplicación, estos cambios aparecerán en el documento subrayados en verde.

## Fase de preparación

Como paso previo a la migración se requiere que la aplicación a migrar cumpla una serie de prerrequisitos. En este apartado se expone cómo preparar nuestra aplicación para migrarla a Angular minimizando los riesgos y priorizando la facilidad y velocidad de todo el ciclo.

| Paso | Obl./Opt. | Descripción |
|---|---|---|
| **1** | *Opcional* | Seguir la Guía de estilos. |
| **2** | *Opcional* | Actualizar AngularJS a su última versión. |
| **3** | *Opcional* | Transformar Controladores en Componentes. |
| **4** | *Opcional* | Eliminar características incompatibles de Directivas. |
| **5** | *Opcional* | Transformar Directivas de componentes en Componentes. |
| **6** | *Obligatorio* | Implementar Bootstrapping manual. |
| **7** | *Opcional* | Añadir Typescript y Compilación. |
| **8** | *Opcional* | Controladores a clases ES6. |
| **9** | *Opcional* | Servicios a clases ES6. |

### **Paso 1:** Seguir la guía de estilos

AngularJS ofrece una guía de estilos que recopila patrones y prácticas que se ha demostrado que dan como resultado aplicaciones AngularJS más limpias y fáciles de mantener. Contiene una gran cantidad de **información sobre cómo escribir y organizar el código AngularJS**.

Una de las claves de la Guía de estilos consiste en estructurar el código siguiendo el principio **Folders-By-Feature**: carpetas con el **nombre de la función que representan**.

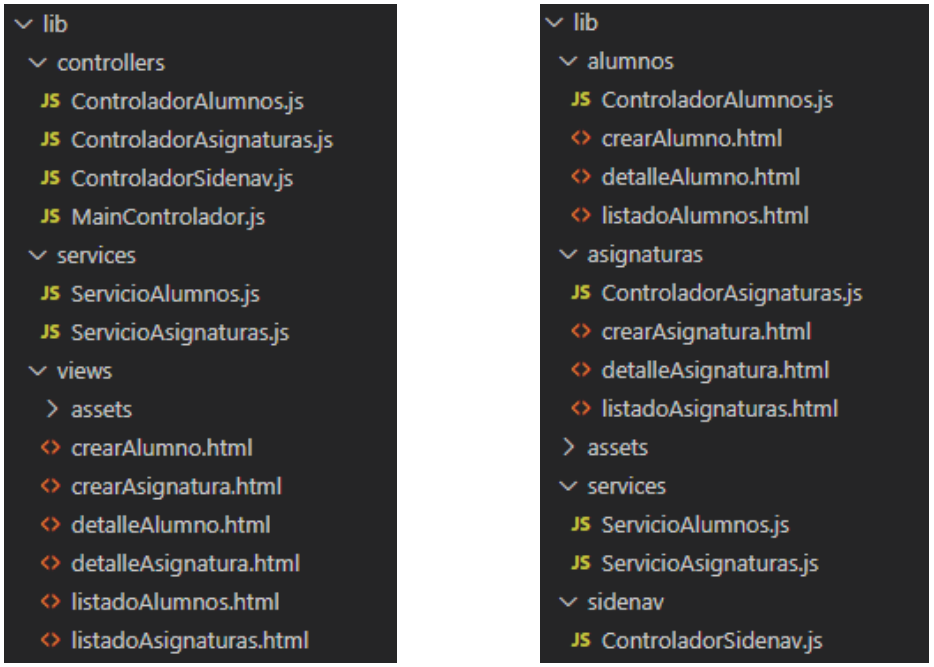
La segunda clave consiste en **definir un único componente por cada fichero**. Cada servicio, controlador o componente se deberá declarar en un fichero independiente.

La tercera y última clave consiste en **utilizar *Controller As* en el enrutamiento** para reemplazar el uso del $scope por *this* en los controladores, e incluir el prefijo $ctrl en los bindings de las vistas.

Lo aplicamos así:

La estructura original agrupa los ficheros por tipo, por lo que nos vemos obligados a modificarla y agrupar por funcionalidad:

*Estructura por tipo* → *Estructura por funcionalidad*



## **Paso 2:** Actualizar AngularJS a su última versión

Para poder emplear todo el potencial que nos ofrece AngularJS es necesario emplear la última versión del Framework. Puede obtenerse desde aquí.

Para comprobar qué versión de AngularJS tenemos, bastará con abrir el fichero angular.min.js con cualquier editor de texto. Encontraremos la versión en la primera línea:

```
/*
 AngularJS v1.8.2
 (c) 2010-2020 Google LLC. http://angularjs.org
 License: MIT
*/
```

Lo aplicamos así:

Opción 1: descargamos la versión 1.8.2 de AngularJS de https://code.angularjs.org/1.8.2/ y la incorporamos en el directorio raíz de la aplicación.

Opción 2: referenciamos a la última versión de AngularJS desde el propio index.html vía script:

```
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"><
/script>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular-route.js"
></script>
```

## **Paso 3:** Transformar Controladores en Componentes de AngularJS.

Convertir los Controladores en Componentes será un punto clave para la posterior migración a los componentes de Angular.

Lo aplicamos así:

Los componentes en AngularJS son **bloques de construcción** que incluyen una vista y su lógica asociada. En nuestro caso, un único controlador gestiona hasta 3 vistas (*p.e: controladorAlumno.js - Listado, Detalle, Creación*), por lo que dividiremos cada controlador en tantos componentes como sea necesario:

| **Controlador** | **Componentes** |
|---|---|
| ControladorAlumnos.js | AlumnosListado.js |
| | AlumnosCreacion.js |
| | AlumnosDetalle.js |
| ControladorAsignaturas.js | AsignaturasListado.js |
| | AsignaturasCreacion.js |
| | AsignaturasDetalle.js |

Ejemplo de actualización de ControladorAlumos.js a Componente AngularJS alumnosListado.js:

```
// Controlador

(function() {
    var app = angular.module("demoAngularJS");

    var ControladorAlumnos = function() {};

    app.controller("ControladorAlumnos", ControladorAlumnos);
  }());
```

```
// Componente (formato camelCase)

(function() {
    var app = angular.module("demoAngularJS");

    var alumnosListado = {
        templateUrl: 'lib/alumnos/listadoAlumnos.html',
        bindings: {},
        controller: function() {
        }
    }
```

```
    app.component("alumnosListado", alumnosListado);
  }());
```

1. **alumnosListado** pasa de ser una función a ser un objeto con las siguientes propiedades:

    - templateUrl: plantilla HTML del componente.
    - bindings: enlaces de datos uni/bidireccionales (anterior $scope) externos al componente.
    - controller: función controlador. Idéntica a la función previa.

2. **Es necesario modificar el enrutador** para que apunte al Componente y no al Controlador. Para ello se eliminará la etiqueta *controller* y se añadirá como **template** (no templateUrl) una directiva customizada con el nombre del nuevo componente:

    *Componente*: `alumnosListado`
    *Template*: `<alumnos-listado><alumnos-listado>`

```
// Antes

.when("/alumnos", {
    templateUrl: "lib/views/listadoAlumnos.html",
    controller: "ControladorAlumnos",
    controllerAs: '$ctrl'
})
```

```
// Después

.when("/alumnos", {
    template: "<alumnos-listado><alumnos-listado>",
    controllerAs: '$ctrl'
})
```

Como último paso, habrá que **referenciar el nuevo componente en el index.html** para que sea accesible por la aplicación.

### **Paso 4:** Eliminar características incompatibles de Directivas.

En AngularJS existen 4 características de las directivas que es necesario eliminar para poder realizar la migración a AngularJS:

1. Método *Compile*
2. Propiedades *Terminal, Priority, Replace*

### **Paso 5:** Transformar Directivas de componentes en Componentes.

Este paso consiste en reemplazar las directivas de componentes en componentes para facilitar su posterior migración a componentes AngularJS.

Lo aplicamos así:

En el proyecto de ejemplo no existen directivas por lo que este paso no es necesario. Sin embargo, se muestra un ejemplo de transformación de *directiva de componente* a *componente*:

```
// Directiva de componente

angular.module('app').directive('panelDetalle', function() {
    return {
        restrict: 'E',
        transclude: true,
        templateUrl: '/components/detallePanel.html',
        scope: {
            titulo: '@'
        },
        bindToController: true,
        controller: function() {}
    }
})
```

```
// Componente

angular.module('app').component('panelDetalle', {
    transclude: true,
    templateUrl: '/components/detallePanel.html',
    bindings: {
        titulo: '@'
    },
    controller: function() {}
})
```

## **Paso 6:** Implementar Bootstrapping manual.

La mayoría de aplicaciones AngularJS utilizan el atributo o directiva *ng-app* para el proceso de bootstrapping o arranque de la aplicación. El objetivo de este punto es implementar un bootstrap manual o programático.

Para ello es suficiente con realizar 2 pasos:

1. Eliminar atributo ng-app.
2. Modificar el fichero root y añadir una función para que AngularJS cargue cuando la página esté cargada.

Lo aplicamos así:

Una vez eliminado el atributo ng-app del index.html, modificamos el fichero app.js, incluyendo la siguiente instrucción al final:

```
angular.element(document).ready(function() {
     angular.bootstrap(document.body, ['demoAngularJS']);
 })
```

Cuando el documento esté listo, AngularJS se iniciará para el módulo `demoAngularJS` en el body de nuestro index.html.

## **Paso 7:** Añadir Typescript y Compilación.

Incorporar el compilador de TypeScript incluso antes de que comience la actualización trae consigo un doble beneficio:

- Nos deshacemos de una tarea más que inevitablemente hemos de acometer durante la fase de migración.
- Se puede comenzar a usar funciones de TypeScript en el código AngularJS, mejorando la aplicación existente.

Lo aplicamos así:

**1. Instalación de Typescript y dependencias:** ejecutando en consola

```
npm install typescript @types/angular @types/core-js @types/jasmine --save-dev
```

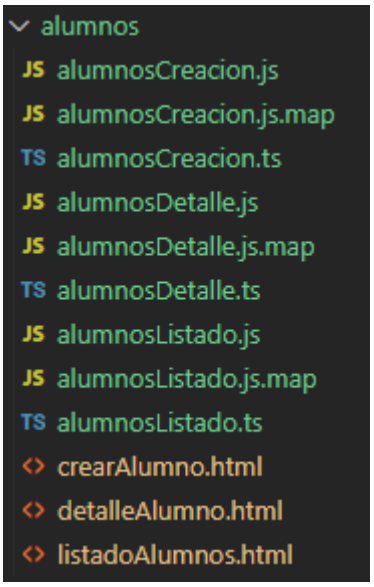
**2. Creamos un comando de compilación typescript** en el fichero package.json: `"tsc": "tsc -p . -w"`

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "tsc": "tsc -p . -w"
},
```

**3. Creamos un fichero de configuración de Typescript** en la raíz del proyecto: tsconfig.json, con la siguiente configuración:

```
{
    "compilerOptions": {
        "target": "ES5",
        "sourceMap": true,
        "emitDecoratorMetadata": true,
        "experimentalDecorators": true,
        "removeComments": false,
        "lib": ["es2017", "dom"]
    },
    "exclude": [
        "node_modules"
    ]
}
```

**5. Convertimos todos los ficheros Javascript a Typescript** cambiando su extensión a .ts, y ejecutamos la compilación con: `npm run tsc`




Por cada fichero .ts, la compilación generará un .js compilado y un .js.map.

Opcionalmente para evitar confusión de ficheros compilados/no compilados, en nuestro editor podremos ignorar los ficheros map, e ignorar .js si existe el correspondiente .ts utilizando reglas customizadas en sus preferencias.

**Nota: A partir de ahora será necesario compilar el código Typescript cada vez que realicemos un cambio.**

## **Paso 8:** Controladores a clases ES6.

En el paso 3 se migran los Controladores AngularJS a Componentes. Los controladores residen ahora como una función dentro del componente. En Angular no existe el término controlador, por lo que será necesario convertir los controladores de los componentes en clases ES6.

Lo aplicamos así:

Funciones a clases ES6:

```
// Antes

controller: function(auth) {}
```

```
// Después

controller: class AlumnosListadoCtrl {
        auth: any;
      constructor(auth) {
             this.auth = auth;
      }
}
```

Se incorpora en el cuerpo de la clase un constructor.

Los servicios inyectados a través del constructor y las variables internas del controlador será obligatorio declararlas. A estas variables se accede con **this.**variable en el .ts, y con **$ctrl**.variable en el .html.

**Importante:** Las funciones ahora seguirán el formato de declaración **Arrow function:**

```
// Antes
onErrorCreacion = function(reason){}
```

```
// Después
onErrorCreacion = (reason) => {}
```

## **Paso 9:** Servicios a clases ES6.

En Angular los servicios son siempre clases, por lo que convertir los servicios existentes de AngularJS en clases ES6 facilitará el proceso de migración, al igual que pasa con los controladores.

Lo aplicamos así:

### 1. Convertir las funciones de callbacks a arrow functions:

```
// Antes

var getAsignaturasAlumno = function(NIA) {
    return $http.get(ApiURL + "/alumno/" + NIA + "/asignaturas")
        .then(function(response){
            return response.data;
        });
};
```

```
// Después

var getAsignaturasAlumno = function(NIA) {
    return $http.get(ApiURL + "/alumno/" + NIA + "/asignaturas")
        .then((response) => {
            return response.data;
        });
};
```

### 2. Modificamos la declaración del servicio:

```
// Antes

app.factory("servicioAlumnos", servicioAlumnos);
```

```
// Después

app.service("servicioAlumnos", servicioAlumnos);
```

### 3. De función a clase ES6:

```
// Antes

var servicioAlumnos = function($http) {}
```

```
// Después

var servicioAsignaturas = class servicioAsignaturas {
        $http: any;
        constructor($http) {
            this.$http = $http
        }
}
```

El funcionamiento interno de la clase será igual que en el caso de los controladores. Declararemos las variables y accederemos a ellas con **this.**variable.

### 4. Eliminar exports

Por último eliminaremos los *exports* del interior de la clase. Ya no serán necesarios dado que los métodos de la clase serán accesibles desde el exterior al instanciarla.

## Fase de migración

Una vez hemos realizado todos los pasos del apartado de preparación, nuestra aplicación AngularJS está lista para ser migrada a Angular.

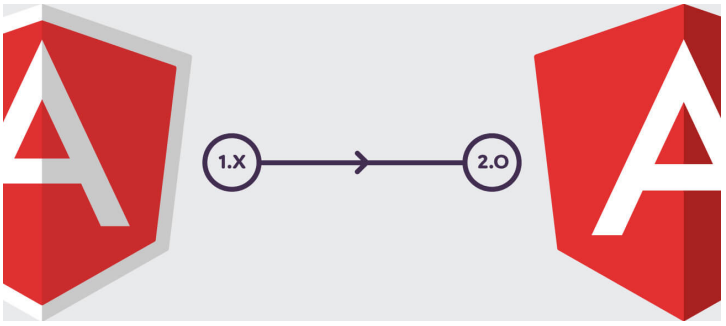



Figura: banner promocional de la migración de AngularJS a Angular

## Proceso híbrido

El proceso de migración será un **proceso gradual**. Durante este proceso se irán migrando partes de la aplicación a Angular mientras el resto de elementos AngularJS siguen funcionando: **ambos Frameworks trabajarán conjuntamente** para facilitar y simplificar el proceso de migración.

Este proceso consta de varios pasos:

1. **Añadir al proyecto Angular** y el resto de dependencias necesarias.
2. **Utilizar un cargador de módulos** o empaquetador: Webpack
3. **Migrar la aplicación:** servicios, componentes, enrutador.
4. **Eliminar AngularJS** del proyecto migrado.

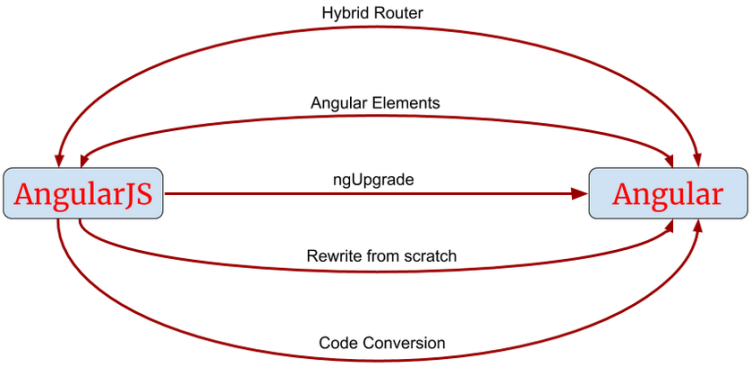



Figura: representación de funcionamiento de una aplicación híbrida

- **Paso 1: Añadir Angular a nuestro proyecto**

Para realizar la migración de la aplicación a Angular es necesario descargar tanto Angular como otras dependencias que serán imprescindibles para la generación de módulos, realización de llamadas HTTP, o upgrades, entre otras cosas.

```
// Instalación de Angular

npm i @angular/common
npm i @angular/compiler
npm i @angular/core
npm i @angular/forms
npm i @angular/http
npm i @angular/platform-browser
npm i @angular/platform-browser-dynamic
npm i @angular/router
npm i @angular/upgrade
```

```
// Instalación de otras dependencias

npm i core-js@2.4.1
npm i rxjs@5.0.1
npm i zone.js
```

```
// Dependencias de desarrollo, deben ir en devDependencies del packages.json

npm i @angular/compiler-cli
npm i @ngtools/webpack
npm i @types/node
npm i angular2-template-loader
npm i awesome-typescript-loader
npm i html-loader
npm i html-webpack-plugin
npm i karma-mocha-reporter
npm i karma-sourcemap-loader
npm i karma-webpack
npm i null-loader
npm i raw-loader
npm i rimraf
npm i webpack
npm i webpack-bundle-analyzer
```

- **Paso 2: Utilizar un cargador de módulos: Webpack**

Cuando se divide el código de la aplicación en un componente por archivo, la estructura del proyecto resulta en una gran cantidad de archivos pequeños. De esta forma los proyectos están mucho más ordenados, pero no funciona tan bien si tiene que cargar todos esos archivos en la página HTML con etiquetas <script>. Utilizar un cargador de módulos es una práctica recomendada que evita esta problemática.

**Webpack** es un empaquetador que toma módulos con dependencias y genera archivos estáticos que representan esos módulos. Es **necesario realizar unos pasos intermedios** para ajustar nuestro código **AngularJS para que Webpack pueda realizar la compilación**.

### 1. Modificar las rutas de las plantillas: *de rutas completas a rutas relativas.*

```
// Antes
templateUrl: 'lib/alumnos/crearAlumno.html'
```

```
// Después
templateUrl: './crearAlumno.html'
```

### 2. Eliminar variables globales

### 3. Modificar fichero tsconfig.json: *añadir en las opciones del compilador:*

***tsconfig.json***

```
"module": "commonjs",
"moduleResolution": "node",
```

### 4. Crear fichero index.ts en el directorio raíz: *incluiremos en el fichero las referencias a los ficheros AngularJS y que están definidas en el index.html:*

***index.html***

```
<script type="text/javascript" src="lib/app.js"></script>
...
<script type="text/javascript" src="lib/sidenav/ControladorSidenav.js"></script>
```

***index.ts***

```
import ".lib/app";
...
import ".lib/sidenav/ControladorSidenav";
```

### ➔ Configurar Webpack

**1. Creamos un directorio de configuración en la raíz**: *./config*
**2. Copiamos en */config* el fichero index.html de la aplicación:** *sin las referencias a los ficheros que AngularJS utiliza.*
**3. Incluimos un fichero de configuración de desarrollo**: *webpack.dev.js*
**4. Incluimos un fichero auxiliar:** *helpers.js*


***helpers.js***

```
const path = require('path');
var _root = path.resolve(__dirname, '..');

function root(args) {
    args = Array.prototype.slice.call(arguments, 0);
    return path.join.apply(path, [_root].concat(args));
}

exports.root = root;
```

***webpack.dev.js***

```
const webpack = require('webpack');
const helpers = require('./helpers');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const ENV = process.env.NODE_ENV = process.env.ENV = 'development';

module.exports = {
    entry: {
        'ng1' : './public/index.ts',
        'app' : './public/main.ts'
    },

    output: {
        path: helpers.root('dist/dev'),
        publicPath: '/',
        filename: '[name].bundle.js',
        chunkFilename: '[id].chunk.js'
    },

    resolve: {
```

```
        extensions: ['.ts', '.js']
    },

    module: {
        rules: [
            {
                test: /\.ts$/,
                use: [
                    {
                        loader: 'awesome-typescript-loader'
                    },
                    {
                        loader: 'angular2-template-loader'
                    }
                ]
            },
            {
                test: /\.html$/,
                use: [
                    {
                        loader: 'html-loader',
                        options: {
                          esModule: false
                        }
                    }
                ]
            }
        ]
    },

    plugins: [
        new webpack.optimize.SplitChunksPlugin({
            minChunks: Infinity
        }),
        new webpack.SourceMapDevToolPlugin({
            "filename": "[file].map[query]",
            "moduleFilenameTemplate": "[resource-path]",
            "fallbackModuleFilenameTemplate": "[resource-path]?[hash]",
            "sourceRoot": "webpack:///"
        }),
        new HtmlWebpackPlugin({
            template: 'config/index.html',
        }),
        new webpack.DefinePlugin({
            'process.env': {
                'ENV': JSON.stringify(ENV)
            }
        }),
        new webpack.ContextReplacementPlugin(
            /angular(\\|\/)core(\\|\/)@angular/,
            helpers.root('./src'),
            {}
        )
    ]
}
```

Es importante que nuestro servidor de la aplicación sirva el contenido estático de los directorios que no están incluidos en el bundle.

En nuestro caso, servimos la aplicación demo con un servidor Node.js + Express, por lo que añadimos la siguiente instrucción para que se sirva el fichero estático de estilos (styles.css):

***servidor (en nuestro caso Node.js)***

```
app.use(express.static(rootPath + '/dist/dev'));
app.use('/public/styles.css', express.static(rootPath + '/public/styles.css'));
app.use('/assets', express.static(rootPath + '/public/assets'));
```

### ➔ Ejecutar Webpack

Creamos los scripts de ejecución en packages.json:

***packages.json***

```
"start": "npm run clean-dev && webpack && --config config/webpack.dev.js --watch
--profile --progress",
"clean-dev": "rimraf dist/dev/*",
```

Una vez se ejecuta el comando *npm run start* (`webpack`), se genera un módulo ng1.bundle.js que incluye los .js de toda la aplicación. Si servimos el index.html generado en el directorio *dist/dev,* **la aplicación se reduce únicamente al uso del index.html y del ng1.bundle.js** (además del contenido estático):

# Inicializar Angular

## 1. Proceso de inicio/bootstrap

Es necesario modificar el arranque de la aplicación para que utilice la versión híbrida. Para ello, desactivaremos la inicialización de AngularJS en favor de Angular. Eliminaremos la siguiente línea del código del fichero de declaración del módulo principal:

***app.ts***

```
angular.element(document).ready(function() {
    angular.bootstrap(document.body, ['demoAngularJS']);
})
```

Ahora activamos el inicio de Angular. Creamos un fichero *main.ts* en el directorio public e incluimos:

***main.ts***

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { UpgradeModule } from '@angular/upgrade/static';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule).then(platformRef => {
    const upgrade = platformRef.injector.get(UpgradeModule) as UpgradeModule;
    upgrade.bootstrap(document.documentElement, ['demoAngularJS'])
});
```

Este punto de entrada requiere de un módulo Angular. Lo creamos en un nuevo directorio */app/app.module.ts:*

***app.module.ts***

```
import { Injectable, NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import { BrowserModule } from '@angular/platform-browser';
import { UpgradeModule } from '@angular/upgrade/static';
import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    UpgradeModule,
  ],
  declarations: [
    AppComponent
  ],
  providers: [
  ],
  bootstrap: [
      AppComponent
  ],
```

```
  entryComponents: [

  ]
})
export class AppModule {}
```

Este módulo requiere a su vez de un componente Angular, que será el que contenga la aplicación. Lo creamos en el mismo directorio y establecemos como vista/plantilla lo siguiente: * *, para que nuestra aplicación AngularJS se renderice en el nuevo componente.

***app.component.ts***

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

@Component({
    selector: 'my-app',
    template: `<div class="ng-view"></div>`
})
export class AppComponent {}
```

Por último debemos eliminar del config/index.html el código AngularJS e incluir la nueva directiva de la aplicación híbrida:

***index.html***

```
<!--<ng-view></ng-view>-->
<my-app></my-app>
```

Como paso final, debemos configurar Webpack para que compile la aplicación Angular junto con la compilación de AngularJS. Modificamos el fichero de configuración webpack.dev.js y añadimos un nuevo entry point:

***webpack.dev.js***

```
entry: {
    'ng1' : './public/index.ts',
    'app' : './public/main.ts'
},
```

y añadimos también el siguiente plugin:

```
new webpack.ContextReplacementPlugin(
    /angular(\\|\/)core(\\|\/)@angular/,
    helpers.root('./src'),
    {}
)
```

Si ejecutamos *npm run start* veremos cómo se crean los compilados de ambas versiones:


Y ambos se referencian en el index.html:

```
<script defer src="/ng1.bundle.js"></script><script defer
src="/app.bundle.js"></script>
```

## 2. Ficheros Polyfills y Vendor

Para optimizar el paso a producción y futuras etapas, incluimos en nuestro directorio *public* estos dos ficheros: polyfills.ts y vendor.ts

- Polyfills: Eliminamos los imports del main.ts y los añadimos en este fichero.
- Vendor: Añadimos los imports clásicos de paquetes Angular:

```
import '@angular/common';
import '@angular/platform-browser-dynamic';
import '@angular/platform-browser';
import '@angular/core';
import '@angular/http';
import '@angular/router';
import '@angular/forms';
import '@angular/upgrade/static';
```

Modificamos los entries del webpack.dev.ts y añadimos:

```
entry: {
    'polyfills': './public/polyfills.ts',
    'vendor': './public/vendor.ts',
    'ng1' : './public/index.ts',
    'app' : './public/main.ts'
},
```

## Paso 3: Migración básica de Angular

En esta sección comenzaremos la migración propiamente descrita. Re-escribiremos algunas piezas de código de AngularJS a Angular, manteniendo la aplicación operativa y funcionando en su versión híbrida.

Todos los elementos migrados se incluirán en el directorio */app* de la aplicación de ejemplo.

### ngUpgrade

Para realizar esta migración híbrida se va a hacer uso de la biblioteca ngUpgrade de Angular. Esta biblioteca es una herramienta útil para migrar aplicaciones AngularJS. Permite **mezclar y combinar componentes AngularJS y Angular en la misma aplicación** y hacer que interoperen sin problemas. Eso significa que **no es necesario realizar todo el trabajo de actualización de una vez**, ya que existe una coexistencia natural entre los dos marcos durante el período de transición.

#### Funcionamiento de ngUpgrade

Una de las herramientas principales proporcionadas por ngUpgrade se llama UpgradeModule. Este es un módulo que contiene utilidades para **arrancar y administrar aplicaciones híbridas que admiten tanto código Angular como AngularJS**. Cuando usa ngUpgrade, realmente **se ejecuta AngularJS y Angular al mismo tiempo**.

Todo el código Angular se ejecuta en el marco Angular y el código AngularJS en el marco AngularJS. **No hay emulación**, por lo que puede esperar tener todas las características y el comportamiento natural de ambos marcos. Lo que sucede además de esto es que los componentes y servicios administrados por un marco pueden interoperar con los del otro marco. Esto ocurre en tres áreas principales: **inyección de dependencia, DOM y detección de cambios.**

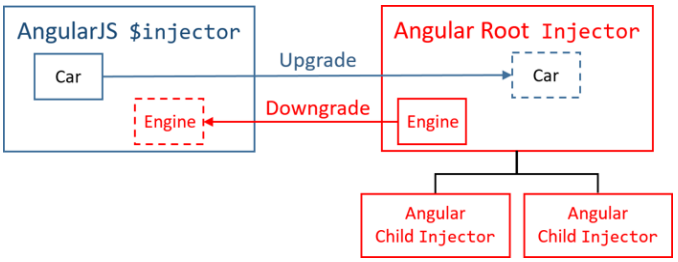



Figura: Inyección de dependencias ngUpgrade

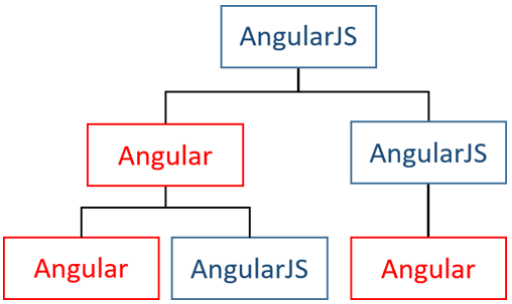



Figura: DOM ngUpgrade

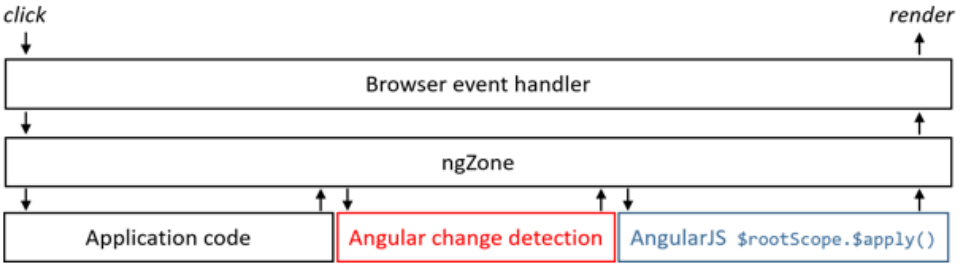



Figura: Detección de cambios ngUpgrade

- **Migrar servicios**

**1. Convertimos el servicio a una clase ES6:** *Eliminamos del servicio toda la información de AngularJS y lo dejamos como clase ES6*

```
// Antes
(function(){
    var app = angular.module("demoAngularJS");
    var servicioAlumnos  = class servicioAlumnos {
```

```
// Después
export class servicioAlumnos {
```

**2. Añadimos decorador `@Injectable`**

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

@Injectable()
export class servicioAlumnos {
```

**3. Renombramos el servicio:** *cambiamos su nombre a alumnos**.service.ts** (guía de estilos, no es obligatorio).*

**4. Copiamos el servicio migrado en el directorio /app***: movemos el servicio a nuestro directorio migrado /app. Creamos en su interior el directorio /services.*

**5. Eliminamos el *import* del index.ts**: *ahora el servicio irá incluido en el paquete app (Angular, no AngularJS).*

**6. App.module.ts: añadimos servicioAlumnos como provider.**

```
providers: [
  servicioAlumnos
],
```

**Este servicio es ahora un servicio Angular**. El problema es que este servicio de Angular es inyectado en un componente de AngularJS. Habrá que registrarlo en el sistema de inyección de dependencias de AngularJS.

Será necesario hacer un **downgrade del servicio para que pueda ser utilizado por AngularJS**.

***Downgrade de Servicio Angular:***

***main.ts***

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { downgradeInjectable, UpgradeModule } from '@angular/upgrade/static';
import { AppModule } from './app/app.module';
import { servicioAlumnos } from './app/services/alumnos.service';

declare var angular: angular.IAngularStatic;

platformBrowserDynamic().bootstrapModule(AppModule).then(platformRef => {
    //Downgrades:
    angular.module('demoAngularJS')
    .factory('servicioAlumnos', downgradeInjectable(servicioAlumnos));

    const upgrade = platformRef.injector.get(UpgradeModule) as UpgradeModule;
    upgrade.bootstrap(document.documentElement, ['demoAngularJS'])
});
```

## 7. Emplear servicio HTTP de Angular: *hasta ahora se hace uso del servicio HTTP de AngularJS.*

***alumnosCreacion.component.ts***

```
import { Http } from "@angular/http";
```

```
constructor(private http: Http) {
    this.http = http
}
```

```
// Antes
getAlumno = (NIA) => {
    return this.$http.get(this.ApiURL + "/alumno/" + NIA)
        .then(function(response){
            return response.data;
        });
};
```

```
// Después
getAlumno = (NIA) => {
    return this.http.get(this.ApiURL + "/alumno/" + NIA)
        .pipe(map((rsp: Response) => {
            let data = rsp.json();
            return data;
        })).toPromise();
};
```

- **Migrar componentes**

Los componentes, al igual que los servicios, serán una clase ES6 con decoradores Angular. El primer paso a realizar en su migración será cambiar la estructura del componente de AngularJS a la nueva estructura:

### 1. Cambios en el controlador

***alumnosCreacion.component.ts***

```
// Antes
var app = angular.module("demoAngularJS");

var alumnosCreacion = {
    templateUrl: './crearAlumno.html',
    bindings: {},
    controller: class AlumnosCreacionCtrl {
```

```
// Después
import { Component } from "@angular/core";

@Component({
    selector: 'alumnos-creacion',
    templateUrl: './crearAlumno.html'
})
export class AlumnosCreacionCtrl {
```

**2. Bindings:** En caso de tener bindings, se añadirán como variables de la propia clase. Dependiendo del tipo de binding las crearemos como variables Input u Output:

```
// INPUTS
@Input() nombreVariable: any;
```

```
// OUTPUTS
@Output() nombreVariable = new EventEmitter():
```

➔ Será necesario realizar el import de los servicios que ya estén migrados, y añadir el tipo en el constructor para que el mecanismo de inyección de dependencias de Angular funcione:

```
import { servicioAlumnos } from "../services/alumnos.service";
```

```
constructor(servicioAlumnos: servicioAlumnos) {
    this.servicioAlumnos = servicioAlumnos;
```

```
}
```

## 3. Upgrade Servicios de AngularJS / terceros

La mayoría de componentes AngularJS emplean servicios. Estos servicios pueden ser propios y, por lo tanto, se pueden migrar a Angular para ser empleados posteriormente por componentes Angular.

Sin embargo, existen servicios como location de AngularJS, o externos como toast, que no pueden ser migrados y que deben de ser utilizados por los componentes migrados. Para conseguir que estos servicios AngularJS funcionen en Angular, haremos que sean compatibles con Angular empleando el módulo Upgrade:

***Upgrade de Servicio AngularJS:***

1. Registramos en *providers* los servicios AngularJS que emplearemos.

***app.module.ts***

```
providers: [
  servicioAlumnos,
  servicioAsignaturas,
  {provide: '$location', useFactory: getLocation, deps: ['$injector']}
],
```

2. Creamos la función `getLocation`:

***app.module.ts***

```
function getLocation(i: any) { return i.get('$location') }
```

3. Inyectamos el servicio en el componente Angular:

***alumnosCreacion.component.ts***

```
constructor(servicioAlumnos: servicioAlumnos, @Inject('$location') private $location) {
    this.servicioAlumnos = servicioAlumnos;
}
```

Al añadir la clave **private**, no será necesario asignarlo a una variable interna de la clase.

## 4. Cambios en la vista:

1. Eliminaremos todas las instancias de `$ctrl`

***crearAlumno.component.html***

```
// Antes
<input ng-model="$ctrl.NIA" type="text" id="NIA" name="NIA">
```

```
// Después
<input ng-model="NIA" type="text" id="NIA" name="NIA">
```

2. Convertimos las directivas *ng-* a directivas Angular:

```
// Antes
<md-button class="md-raised md-warn botonDerecha" style="color: white"
ng-click="cancelar()">
```

```
// Después
<button class="md-raised md-warn botonDerecha" style="color: white"
(click)="cancelar()">
```

En casos especiales, como en el uso de AngularJS Material, será necesario reescribir los elementos y adecuarlos a la librería Material de Angular.

## 5. Últimos cambios:

- **Copiar componente a /app**: copiamos el nuevo componente y la plantilla en el directorio migrado */app/alumnos* (renombrando ambos ficheros según ).
- **Eliminar componente antiguo de index.ts**
- **app.module.ts**: añadir componente en declarations y entryComponents.
- **main.ts**: downgradear componente migrado para que pueda ser empleado por AngularJS:

***main.ts***

```
.directive('alumnosCreacion', downgradeComponent({component: AlumnosCreacionCtrl}));
```

- **Migrar enrutador**

Hasta ahora el **enrutamiento ha sido gestionado completamente por AngularJS**, por lo que es necesario migrar este elemento a Angular. Este será el último paso y requerirá que el resto de elementos estén migrados. Para llevar a cabo **la migración de cada ruta** habrá que seguir los siguientes pasos:

**1. Eliminar la ruta del enrutador AngularJS.** *Se eliminará además la ruta por defecto (otherwise).*

**2. Añadir módulo de enrutamiento en app.module.ts**: *se irán añadiendo las rutas y sus correspondientes componentes.*

***app.module.ts***

```
@NgModule({
  imports: [
    ...
    RouterModule.forRoot([
      { path: 'alumnos/nuevo', component: AlumnosCreacionCtrl}
    ], {useHash: true})
```

**3. Indicar al enrutador Angular que sólo gestione las URLs indicadas en el punto anterior.** *Para ello, crearemos una clase en el mismo app.module.ts con la siguiente información:*

***app.module.ts***

```
class Ng1Ng2UrlHandlingStrategy implements UrlHandlingStrategy {
  shouldProcessUrl(url: UrlTree): boolean {
    return url.toString().startsWith('/alumnos/nuevo');
  }
  extract(url: UrlTree): UrlTree { return url; }
  merge(newUrlPart: UrlTree, rawUrl: UrlTree): UrlTree { return newUrlPart; }
}
```

Añadimos esta clase como Provider para que Angular lo tenga en cuenta en la gestión de las URLs. Añadimos otro provider necesario para la correcta ejecución:

***app.module.ts***

```
  providers: [
    ...
    {provide: UrlHandlingStrategy, useClass: Ng1Ng2UrlHandlingStrategy },
    {provide: '$scope', useExisting: '$rootScope'}
  ],
```

**4. Añadimos el router-outlet en app.component.ts**

***app.component.ts***

```
@Component({
    selector: 'my-app',
    template: `
    <router-outlet></router-outlet>
    <div class="ng-view"></div>
    `
})
export class AppComponent {}
```

**5. Eliminamos dependencia location, añadimos router.** *En los componentes migrados, eliminamos la dependencia al servicio $location de AngularJS, e inyectamos el Router de Angular:*

```
constructor(servicioAlumnos: servicioAlumnos, private router: Router) {
    this.servicioAlumnos = servicioAlumnos;
}
```

```
this.router.navigate(['/alumnos']);
```

**6. Sustituir:** los `<a href="/ruta">`, por `[routerLink]="['/ruta']"`

**7. HashRouting:** Además se recomienda eliminar el HashRouting (# en la URL) para el correcto funcionamiento del router. Para ello:

***app.module.ts***

```
RouterModule.forRoot([
  { path: 'alumnos/nuevo', component: AlumnosCreacion},
  { path: 'alumnos/:NIA', component: AlumnosDetalle},
], {useHash: false})
```

***app.ts***

```
app.config(['$locationProvider', function($locationProvider) {
    $locationProvider.html5Mode(true);
}])
```

***main.ts***

```
upgrade.bootstrap(document.documentElement, ['demoAngularJS']);
setUpLocationSync(upgrade);
```

- **Completar migración / Eliminar AngularJS**

Una vez tenemos nuestra aplicación totalmente migrada a Angular, eliminaremos completamente AngularJS de la aplicación siguiendo los pasos que se muestran a continuación:

### 1. app.module.ts:

- Eliminamos gestor de URLs: `Ng1Ng2UrlHandlingStrategy`
- Eliminamos $scope de providers.
- Eliminamos $location de providers.
- Eliminamos contenido de *entryComponents*

### 2. app.component.ts: *Eliminamos ng-view:*

```
@Component({
    selector: 'my-app',
    template: `
    <router-outlet></router-outlet>
    <!--<div class="ng-view"></div>-->
    `
})
```

### 3. main.ts: *eliminamos el then() con los downgrades:*

```
platformBrowserDynamic().bootstrapModule(AppModule).then(platformRef => {
    //Downgrades:
    angular.module('demoAngularJS')
    .factory('servicioAlumnos', downgradeInjectable(servicioAlumnos))
    .factory('servicioAsignaturas', downgradeInjectable(servicioAsignaturas))

    const upgrade = platformRef.injector.get(UpgradeModule) as UpgradeModule;
    upgrade.bootstrap(document.documentElement, ['demoAngularJS'])
});
```

### 4. package.json: *eliminamos la compilación de AngularJS: solo se generarán los ficheros Angular.*

### 5. Eliminar referencias a AngularJS en el index.html.

```
<script src="https://ajax.googleapis.com/../1.8.2/angular.min.js"></script>
<script src="https://ajax.googleapis.com/../1.8.2/angular-route.js"></script>
<script src="https://ajax.googleapis.com/../1.2.2/angular-material.js"></script>
<script src="https://ajax.googleapis.com/../1.8.2/angular-animate.min.js"></script>
<script src="https://ajax.googleapis.com/../1.8.2/angular-aria.min.js"></script>
```

```
<script src="https://ajax.googleapis.com/../1.8.2/angular-messges.min.js"></script>
<link rel="stylesheet"
href="https://ajax.googleapis.com/.../1.2.2/angular-material.css">
```

# Conclusiones

La elaboración de este proyecto y su implementación práctica sobre un proyecto real han permitido extraer varias conclusiones tanto positivas como negativas sobre los frameworks AngularJS, Angular y la migración entre ambos:

### Positivas

- Angular es un **framework potente, con proyección de futuro, respaldado por Google y con características fiables** (Typescript - Microsoft).
- La **migración híbrida** permite tener parte de la aplicación en AngularJS y la otra parte ya migrada en Angular, **funcionando conjuntamente**: esto permitirá ir migrando la aplicación poco a poco y reduciendo la complejidad y el esfuerzo.
- AngularJS es un **framework obsoleto**: la falta de novedad tecnológica (componentes, SEO, móvil…), los problemas de rendimiento, la ausencia de soporte y los consiguientes posibles agujeros de seguridad hacen que sea **necesario migrar** las aplicaciones que empleen esta tecnología.

### Negativas

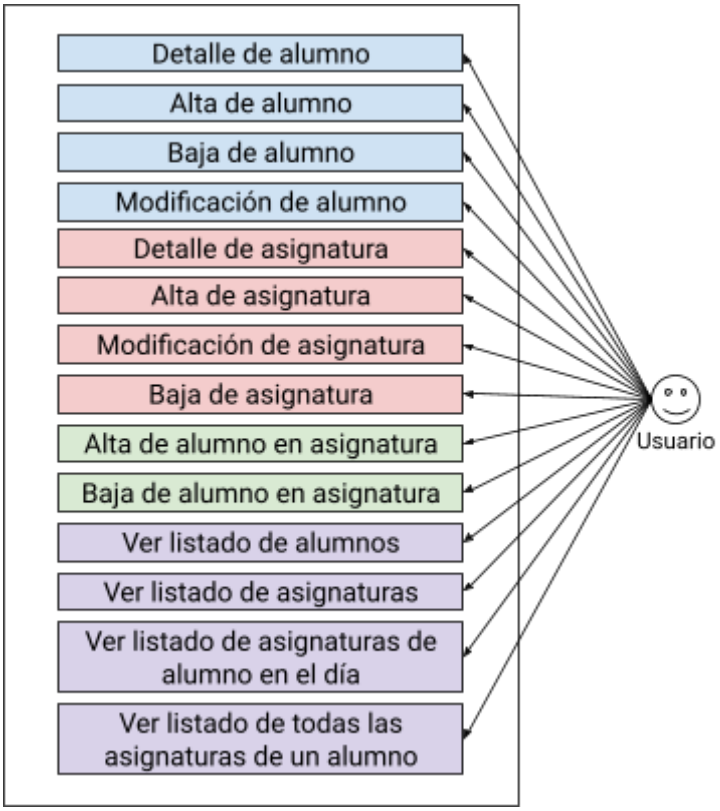
- **La migración de aplicaciones AngularJS a Angular es un proceso muy costoso y de alta dificultad**: se requiere tener amplios conocimientos sobre ambos frameworks, sobre Webpack, y sobre todo sobre la estructura de la aplicación a migrar.

- **Existe poca información sobre el proceso de migración**: la documentación oficial de Angular es limitada y muy escasa.
- La migración de aplicaciones AngularJS a Angular **es un proceso muy manual**: la generación de módulos con Webpack, reescritura de elementos, etc. se realizan de forma manual. Esta estrategia no incluye Angular CLI en la migración de los componentes del proyecto.
- Esta estrategia de migración híbrida está **basada en versiones específicas de muchos componentes**: la utilización de las últimas versiones de estos componentes (rxjs, core-js…) implica cambios en ficheros de configuración, estructuras Webpack… complicando más si cabe el proceso de migración.
- Esta estrategia de migración **no se integra con Angular CLI.** Para hacerlo es necesario crear una aplicación Angular desde el CLI y copiar la aplicación **ya migrada,** por lo que el CLI no es útil en el proceso de migración.

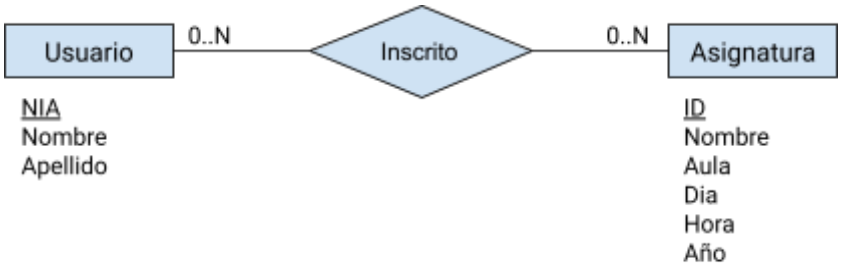
Por todo lo anterior conviene estudiar en detalle cada proyecto a migrar, para decidir si la mejor opción es realizar todo este proceso de migración, o si bien la reescritura completa de la aplicación a otro Framework puede ser una opción más rápida que aporte más valor.

# Anexo 1: Aplicación de Ejemplo - AngularJS

## Casos de uso




## Diagrama Entidad-Relación




## Modelo Relacional

1. **Usuario** (NIA, Nombre, Apellido)

   NIA: Número de identificación único del usuario en la universidad.
   Nombre: Nombre del usuario.
   Apellido: Apellido del usuario.

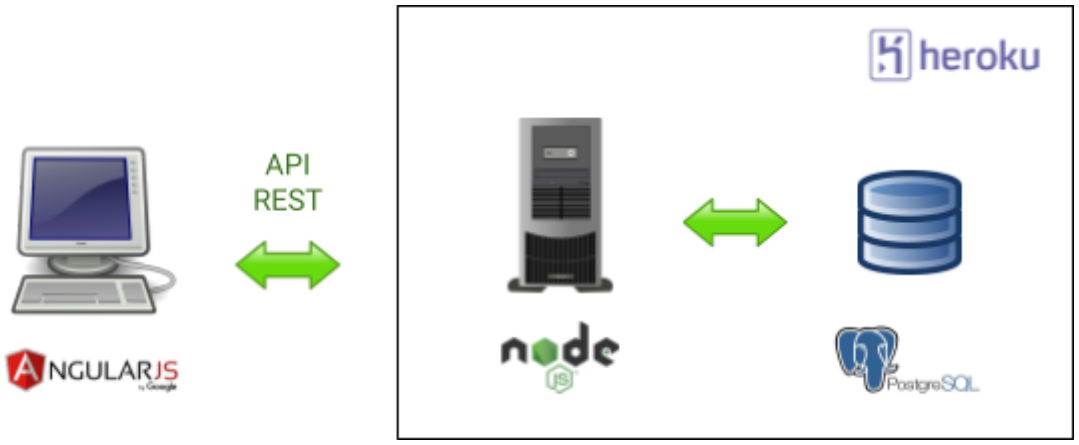
2. **Asignatura** (ID, Nombre, Aula, Día, Hora, Año)

   ID: Identificador único de la asignatura.
   Nombre: Nombre identificativo de la asignatura.
   Aula: Número de aula en el que se imparte la asignatura.
   Día: Número de la semana en el que se imparte la asignatura (1-7).
   Hora: Hora del día en la que se imparte la asignatura.
   Año: Año de impartición de la asignatura.

3. **Alumno_Asignatura** (NIA, ID)

   NIA: Identificador del alumno.
   ID: Identificador de la asignatura.

### Arquitectura




En sintonía con el trabajo, **en la parte BackEnd se ha implementado un API REST programado en lenguaje Javascript** utilizando NodeJS + Express.

Como **sistema gestor de bases de datos** se ha empleado PostgreSQL.

Con la idea de poder reutilizar el BackEnd posteriormente en este análisis con otros frameworks, ambos componentes se han publicado en internet empleando Heroku, plataforma en la nube que permite desplegar de manera sencilla y gratuita proyectos web.

- Resultado Aplicación DEMO
- Código fuente FrontEnd AngularJS
- Código fuente BackEnd NodeJS


Figura : Listado de alumnos

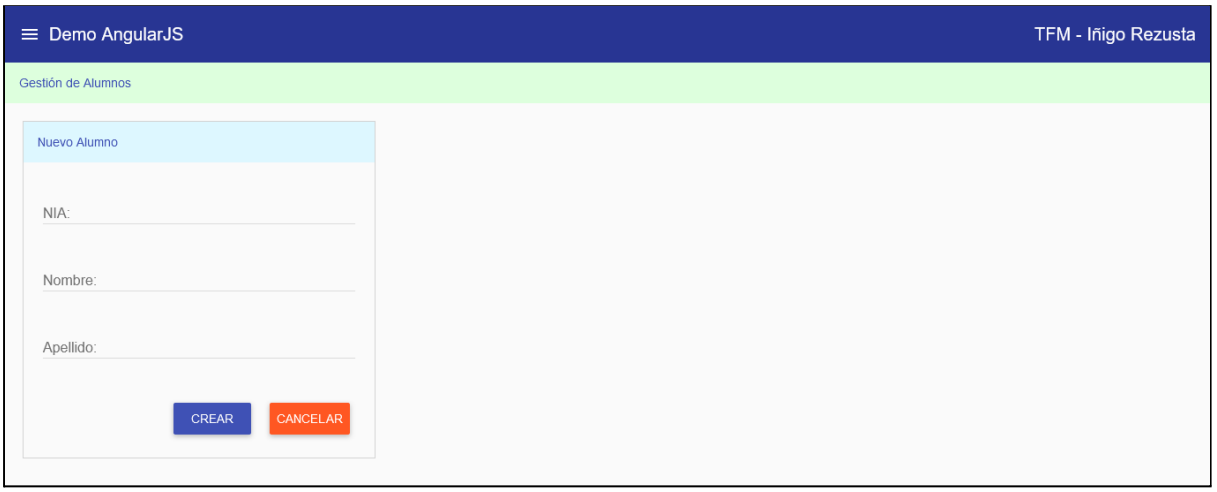



Figura : Creación de alumno

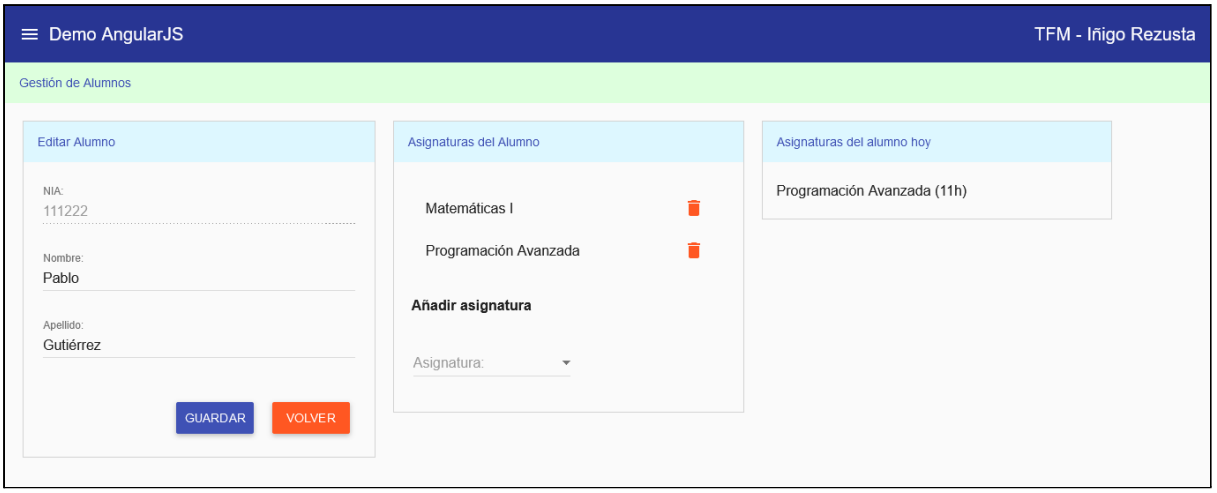



Figura : Edición/Visualización de alumno


Figura : Listado de asignaturas

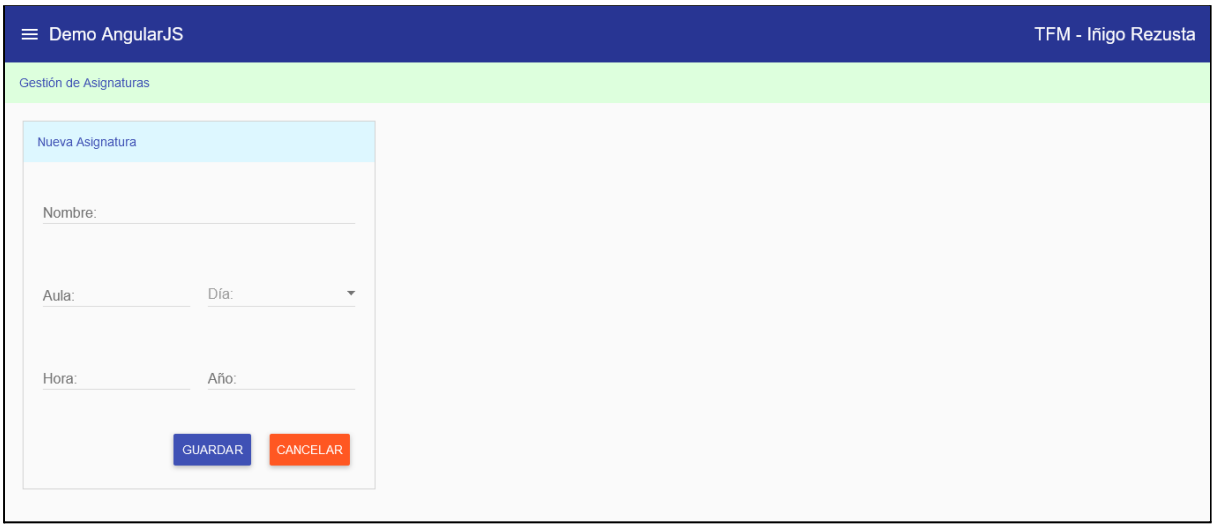



Figura : Creación de asignatura

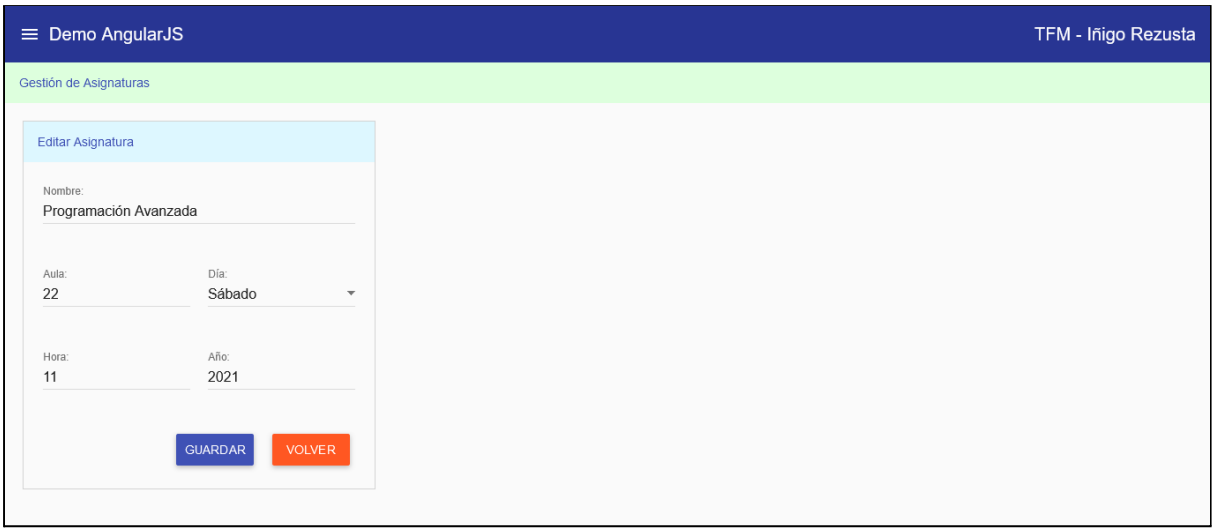
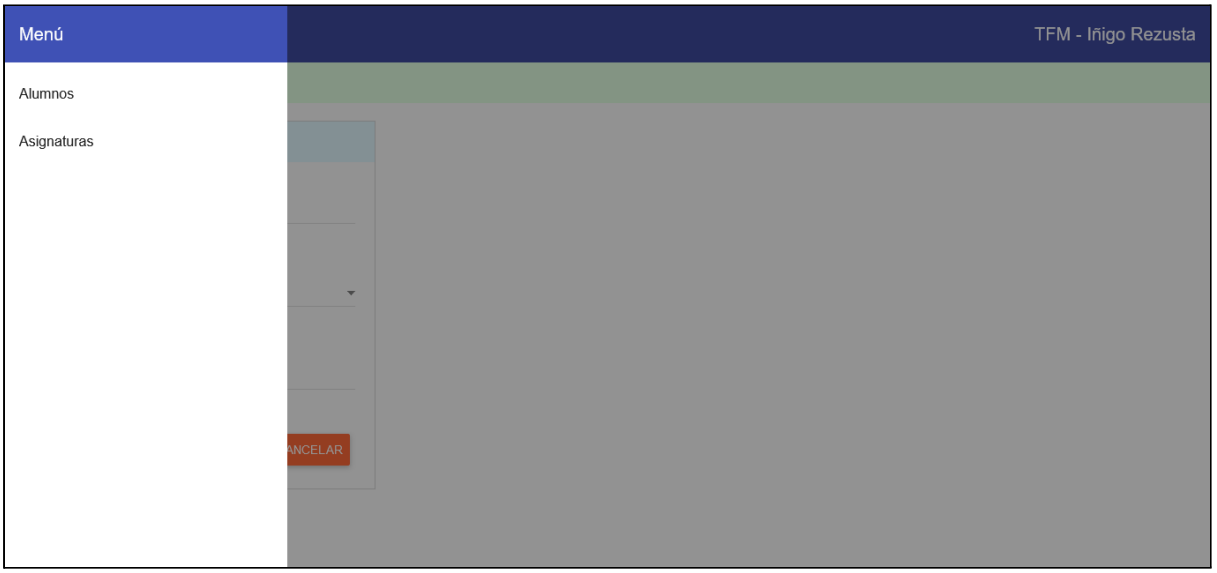



Figura : Detalle/Edición de asignatura


Menú
TFM - Iñigo Rezusta
Alumnos
Asignaturas
ANCELAR

# Anexo 2: Diferencias de construcción entre ambos Frameworks.

Las **principales diferencias** que encontramos entre ambos Frameworks pueden resumirse en los siguientes puntos:

1. Lenguajes de programación
2. Controladores vs. Componentes
3. Sintaxis de directivas estructurales
4. Directivas vs. Propiedades de elementos y eventos del DOM
5. Binding
6. Inyección de dependencias
7. Routing
8. Rendimiento y versatilidad.

## 1. Lenguajes de programación

Angular extiende los lenguajes de programación en los que se pueden desarrollar las aplicaciones, añadiendo TypeScript:

| Lenguaje | AngularJS | Angular |
|---|---|---|
| *ES5* | ✔ | ✔ |
| *ES6* | ✔ | ✔ |
| *Dart (Google)* | ✔ | ✔ |
| *TypeScript* | ❌ | ✔ |

**El uso de Typescript supone un gran avance**: lenguaje de programación que extiende JavaScript y añade **tipado estático y objetos basados en clases**. Dado que extiende JavaScript, cualquier código JavaScript funcionará sin problemas. Sin embargo, no es un lenguaje interpretado y requiere de compilador, que traducirá el código a JavaScript original.

## 2. Controladores, $scope y componentes

AngularJS basa su estructura en el uso de controladores que contienen la lógica de la aplicación y vinculan vista y modelo a través de la variable scope.

Angular sustituye el uso de controladores por el uso de componentes. Las aplicaciones componentizadas son un estándar web de futuro.

### 3. Sintaxis de directivas estructurales

Angular emplea ahora el "*" como prefijo para directivas estructurales; "*in*" se sustituye por "*of*" y se usa la sintaxis *camelCase*.

| AngularJS | Angular |
|---|---|
| `ng-repeat = "alumno in alumnos"` | `*ngFor = "alumno of alumnos"` |

### 4. Directivas vs. Propiedades de elementos y eventos del DOM

Angular emplea las propiedades de los elementos y los eventos estándar del DOM incorporando paréntesis, mientras que AngularJS disponía de directivas específicas para obtener los mismos resultados.

| AngularJS | Angular |
|---|---|
| `<button ng-click="eventoClick()">` | `<button (click)="eventoClick()">` |

### 5. Binding

El enlace de datos en **una y dos direcciones** entre vista y lógica se modifica:

| Tipo de binding | AngularJS | Angular |
|---|---|---|
| One way binding | `ng-bind="alumno.NIA"` | `[value]="alumno.NIA"` |
| Two way binding | `ng-model="alumno.NIA` | `[(ngModel)]="alumno.NIA"` |

### 6. Inyección de dependencias

En AngularJS se inyectan las dependencias en la declaración del controlador, mientras que Angular, al emplear clases, realiza la inyección de dependencias en el constructor:

| | |
|---|---|
| **AngularJS** | `.controller("alumnosController", function($scope, $http) {` |
| **Angular** | `constructor(private _http: Http)` |

### 7. Enrutamiento

Se modifica tanto el sistema de creación de rutas como la inyección del enrutador en la aplicación:

| | Enrutamiento | Inyección en la vista |
|---|---|---|
| **AngularJS** | `$routeProvider.when()` | `ng-view` |
| **Angular** | `@RouteConfig{(...})` | `<router-outlet>` |

### 8. Rendimiento y versatilidad

Angular utiliza un sistema de inyección de dependencias jerárquico **impulsando su rendimiento** notoriamente. También incorpora un sistema de **detección de cambios basado en árboles unidireccionales**, que también incrementa el rendimiento. Se estima que Angular puede llegar a ser **5 veces más rápido que AngularJS**.

Por otro lado, AngularJS no tiene en cuenta de forma nativa el **soporte para dispositivos móviles**. Sin embargo, Angular se ha creado teniendo en cuenta una arquitectura orientada a móviles, permitiendo el uso de bibliotecas específicas y renderizando el contenido de forma distinta para optimizar el rendimiento.

# Bibliografía


- https://www.pluralsight.com/
- https://es.wikipedia.org/wiki/Single-page_application
- https://developer.mozilla.org/es/docs/Web/CSS
- https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada
- https://vanseodesign.com/css/benefits-of-cascading-style-sheets/
- https://developer.mozilla.org/es/docs/Glossary/Prototype-based_programming
- https://developer.mozilla.org/es/docs/Web/JavaScript
- https://es.wikipedia.org/wiki/JavaScript
- https://developer.mozilla.org/es/docs/Web/HTML
- https://developer.mozilla.org/es/docs/Learn/Getting_started_with_the_web/HTML_basics
- https://www.typescriptlang.org/
- https://es.wikipedia.org/wiki/TypeScript
- https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58
- https://descubrecomunicacion.com/que-es-backend-y-frontend/
- https://platzi.com/blog/que-es-frontend-y-backend/
- https://airfocus.com/glossary/what-is-a-front-end/
- https://neoattack.com/neowiki/framework/
- https://rockcontent.com/es/blog/framework/
- https://es.wikipedia.org/wiki/Aplicaci%C3%B3n_web
- https://edu.gcfglobal.org/es/informatica-basica/que-son-las-aplicaciones-web/1/
- https://en.wikipedia.org/wiki/Web_application
- https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction
- https://developer.mozilla.org/es/docs/Learn/Herramientas_y_pruebas/Lado-del-cliente_JavaScript_frameworks
- https://www.angularjswiki.com/angular/history-of-angularjs/
- https://www.ryadel.com/en/angular-angularjs-history-through-years-2009-2019/
- https://insights.stackoverflow.com/trends?tags=angularjs
- https://trends.google.es/trends/explore?date=2009-06-01%202021-02-09&geo=ES&q=%2Fm%2F0j45p7w
- https://www.peerbits.com/blog/reasons-behind-growing-popularity-of-angularjs.html
- https://blog.logrocket.com/history-of-frontend-frameworks/
- https://www.grapecity.com/blogs/angular-roadmap-the-past-present-and-future-of-angular
- https://www.w3schools.com/angular/angular_databinding.asp
- https://docs.angularjs.org/guide
- https://guru99.es/angularjs-controller/
- https://www.w3schools.com/angular/angular_controllers.asp
- http://blog.enriqueoriol.com/2016/03/diferencias-servicios-angularjs.html
- https://guru99.es/angularjs-controller/
- https://www.adictosaltrabajo.com/2015/07/16/introduccion-a-las-directivas-en-angularjs/
- https://www.c-sharpcorner.com/UploadFile/17e8f6/creating-custom-directives-in-angularjs/
- https://es.wikipedia.org/wiki/AngularJS
- https://angular.io/guide/what-is-angular
- https://angular.io/guide/component-overview
- https://angular.io/cli
- https://www.campusmvp.es/recursos/post/las-10-principales-diferencias-entre-angularjs-y-angular.aspx
- https://blog.angular-university.io/angularjs-vs-angular-an-in-depth-comparison/
- https://www.quora.com/Why-is-Angular-2-not-backwards-compatible-with-AngularJS
- https://www.linkedin.com/jobs/search/?geoId=92000000&keywords=ANGULAR&location=En%20todo%20el%20mundo
- https://medium.com/zerotomastery/tech-trends-showdown-react-vs-angular-vs-vue-61ffaf1d8706
- https://trends.google.es/trends/explore?q=%2Fg%2F11c6w0ddw9,%2Fm%2F012l1vxv,%2Fg%2F11c0vmgx5d
- https://angular.io/guide/releases