LSE001-2016: Diplomado de Sistemas Embebidos para Automatización y Robótica

Dr. Casimiro Gómez González
Facultad de Electrónica, UPAEP
correo: casimiro.gomez@upaep.mx
Tel: 2225 640517

Otoño de 2016

Prólogo

El presente material ha sido elaborado en el laboratorio de sistemas embebidos UPAEP, y se ha desarrollado con la experiencia de estudiantes y profesores que han colaborado en dicho laboratorio. Hay material propio de clases y otro material generado a través de proyectos de vinculación y consultaría. Cualquier comentario o corrección favor de enviarlo por correo al autor.

El autor
Casimiro Gómez González
Doctor en Ingeniería Mecatrónica
correo: casimiro.gomez@upaep.mx

Índice general

Capítulo 1

Conceptos Básicos de Sistemas Embebidos

Los últimos diez años mas o menos, el mundo de la computación se ha movido desde las máquinas de escritorio grandes y estáticas a los dispositivos embebidos pequeños y móviles. Sistemas de software corriendo en redes de móviles, y los dispositivos embebidos deben tener propiedades que no siempre requieren los sistemas tradicionales:

- Rendimiento cerca del óptimo
- Robustez
- Distribución
- Dinamismo
- Movilidad

Una de las diferencias en la ingeniería de software para sistemas embebidos es el conocimiento adicional que el ingeniero tiene de potencia eléctrica y electrónica; interfaces físicas de electrónica analógica y digital con la computadora; y diseño de software para sistemas embebidos y procesamiento digital de señales.

Cerca del 95 % de los sistemas de software son actualmente embebidos. Considera los dispositivos que tienes en casa de uso diario:

- Teléfono Celular, iPOD, microondas

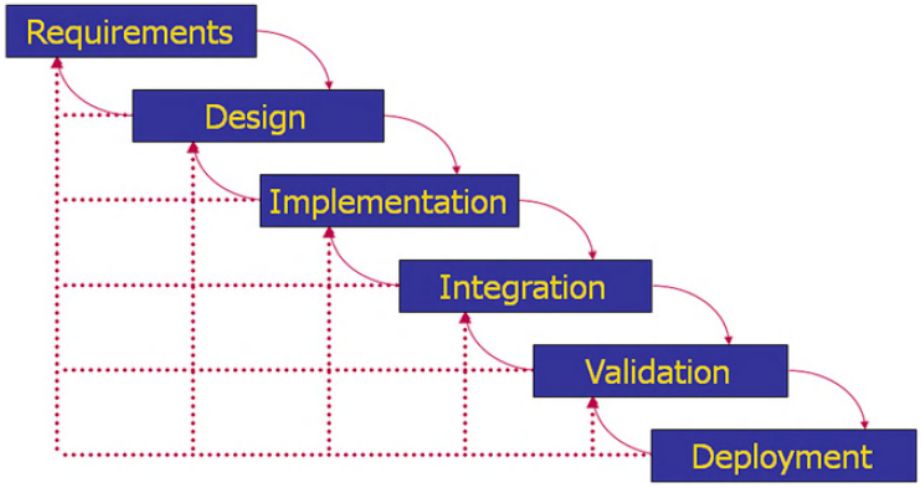

Figura 1.1: Modelo de desarrollo de software *Waterfall*

- receptor satelital de TV, receptor de TV de cable
- Unidad de control de automovil
- Reproductor de DVD

El desarrollo de software embebido usa los mismos modelos de desarrollo de software que las otras técnicas convencionales, incluyendo el modelo *Waterfall*, el modelo en *espiral* y el modelo *Agile*.

La fases principales para el desarrollo de sistemas embebidos se pueden describir:

1. **Definición del problema**: En esta fase se determina exactamente que quiere el cliente y el usuario. Esto incluye el desarrollo de un contrato con el cliente, dependiendo que tipo de producto esta siendo desarrollado. El objetivo de esta fase es especificar que producto de software se hará. Las dificultades incluyen: la solicitud del cliente por el producto incorrecto, el cliente no sabe sobre computación/software, lo cual limita la efectividad de esta fase, las especificaciones por lo regular son ambiguas, inconsistentes e incompletas.

2. **Arquitectura/diseño**: La arquitectura se refiere a la selección de los elementos de arquitectónicos, sus interacciones, y las restricciones de

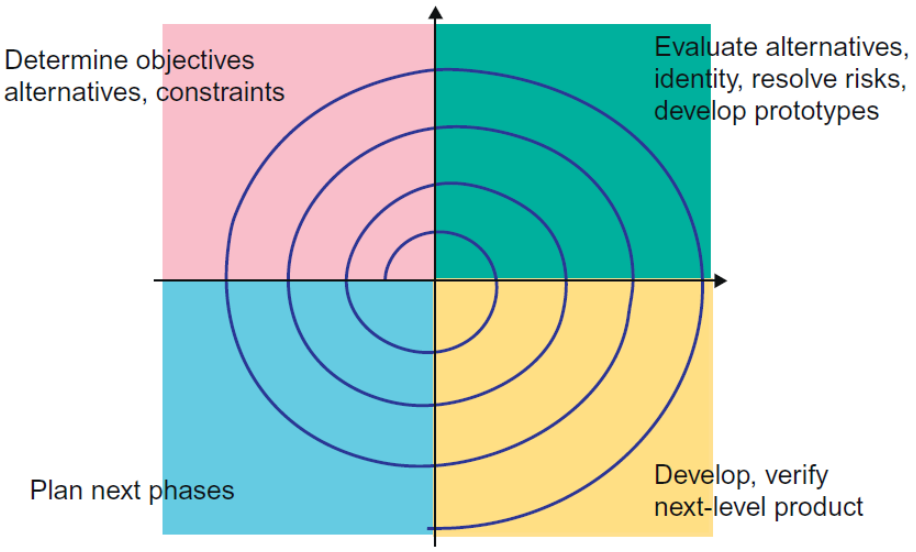

Figura 1.2: Modelo de desarrollo de software *Espiral*

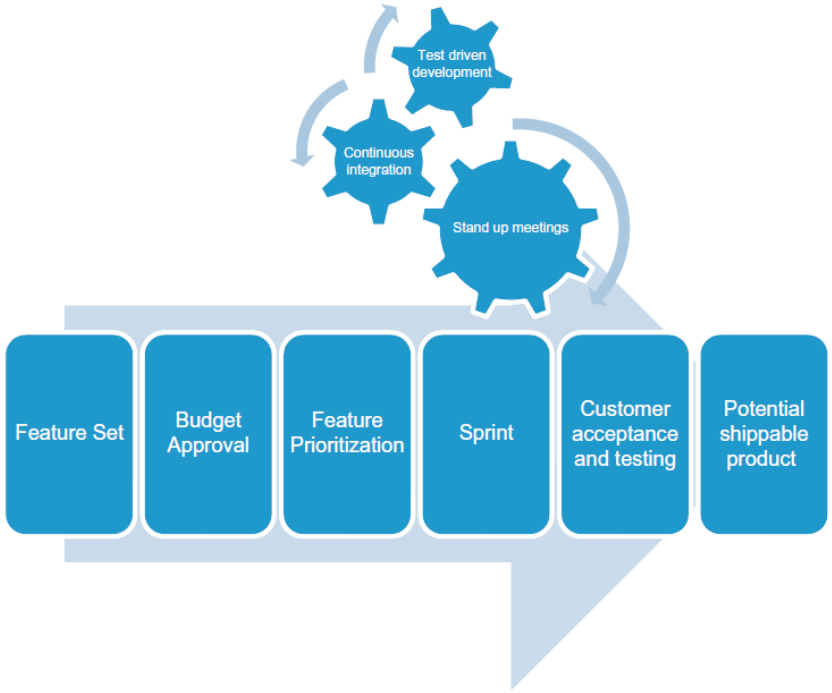

Figura 1.3: Modelo de desarrollo de software *Agile*

aquellos elementos y sus interacciones necesarios para proporcionar un ambiente de trabajo con el cual satisfacer los requerimientos y servir como base para el diseño. El diseño se refiere a la modularización y detalle de interfaz entre los elementos de diseño, sus algoritmos y procedimientos, y los tipos de datos necesarios para soportar la arquitectura y satisfacer los requerimientos. Durante la fase de arquitectura y diseño, el sistema es descompuesto en módulos de software con interfaces. Durante el diseño, el equipo de software desarrolla las especificaciones de los módulos (algoritmos y los tipos de datos), manteniendo un récord de las decisiones de diseño y la trazabilidad, y especifica como los productos de software harán estas tareas. La dificultad principal durante esta fase incluye la falta de comunicación entre los diseñadores de módulos y terminan desarrollando un diseño que puede ser inconsistente, incompleto y ambiguo.

3. **Implementación**: Durante esta fase el equipo de desarrollo implementa los módulos y componentes y verifica que cumplan las especificaciones. Los módulos se combinan de acuerdo con el diseño. Las implementaciones especifican como los productos del software hacen su trabajo. Algunas de las dificultades principales incluyen los errores de interacción entre los módulos y los errores de integración que pueden influenciar la calidad y la productividad.

4. **Verificación y Validación(V&V)**: Hay varias formas de V&V. Una forma es el "análisis". El análisis puede ser en la forma de verificación estática, científica y formal y en revisiones informales. Las pruebas es una forma dinámica de V&V. Esta forma de pruebas viene en la forma de una caja blanca (se tiene acceso al código) y de caja negra (no hay acceso al código). Las pruebas pueden ser estructurales o de comportamiento.Hay pruebas estándares para realizar dependiendo del producto a desarrollar.

Mas y mas desarrollo de software para sistemas embebidos se esta moviendo al desarrollo basado en componentes.Este tipo de desarrollo es aplicado generalmente para componentes de tamaño razonable y reutilización a través del sistema, que es una tendencia creciente en sistemas embebidos. Los desarrolladores aseguran que estos componentes son adaptables a contextos variantes y extienden la idea mas allá del código a otros artefactos desarrollados también. Esta técnica cambia la ecuación desde "Integración, entonces

Bajar al Hardware" a "Bajar al Hardware, entonces Implementar".

1.1. Definición formal de Sistemas Embebidos

Se trata de un sistema de computación diseñado para realizar una o algunas funciones dedicadas frecuentemente en un sistema de computación en tiempo real. Al contrario de lo que ocurre con los ordenadores de propósito general (como por ejemplo una computadora personal o PC) que están diseñados para cubrir un amplio rango de necesidades, los sistemas embebidos se diseñan para cubrir necesidades específicas.

1.2. Historia

Capítulo 2

Aplicaciones con Angular 2

Desde su creación hace ya más de 4 años, Angular ha sido el framework preferido por la mayoría de los desarrolladores Javascript. Este éxito ha provocado que los desarrolladores quieran usar el framework para más y más cosas.

De ser una plataforma para la creación de Web Apps, ha evolucionado como motor de una enorme cantidad de proyectos del ámbito empresarial y de ahí para aplicaciones en la Web Mobile Híbrida, llevando la tecnología al límite de sus posibilidades.

Es el motivo por el que comenzaron a detectarse problemas en Angular 1, o necesidades donde no se alcanzaba una solución a la altura de lo deseable. Son las siguientes.

- Javascript.- Para comenzar encontramos problemas en la creación de aplicaciones debido al propio Javascript. Es un lenguaje con carácter dinámico, asíncrono y de complicada depuración. Al ser tan particular resulta difícil adaptarse a él, sobre todo para personas que están acostumbradas a manejar lenguajes más tradicionales como Java o C#, porque muchas cosas que serían básicas en esos lenguajes no funcionan igualmente en Javascript.

- Desarrollo del lado del cliente.- Ya sabemos que con Angular te llevas al navegador mucha programación que antes estaba del lado del servidor, comenzando por el renderizado de las vistas. Esto hace que surjan nuevos problemas y desafíos. Uno de ellos es la sobrecarga en el navegador, haciendo que algunas aplicaciones sean lentas usando Angular 1 como motor.

Por otra parte tenemos un impacto negativo en la primera visita, ya que se tiene que descargar todo el código de la aplicación (todas las páginas, todas las vistas, todas las rutas, componentes, etc), que puede llegar a tener un peso de megas. A partir de la segunda visita no es un problema, porque ya están descargados los scripts y cacheados en el navegador, pero para un visitante ocasional sí que representa un inconveniente grande porque nota que la aplicación tarda en cargar inicialmente.

Los intentos de implementar Lazy Load, o carga perezosa, en el framework en su versión 1.x no fueron muy fructíferos. Lo ideal sería que no fuese necesario cargar toda tu aplicación desde el primer instante, pero es algo muy difícil de conseguir en la versión precedente por el propio inyector de dependencias de Angular 1.x.

Otro de los problemas tradicionales de Angular era el impacto negativo en el SEO, producido por un renderizado en el lado del cliente. El contenido se inyecta mediante Javascript y aunque se dice que Google ha empezado a tener en cuenta ese tipo de contenido, las posibilidades de posicionamiento de aplicaciones Angular 1 eran mucho menores. Nuevamente, debido a la tecnología de Angular 1, era difícil de salvar.

Todos esos problemas, difíciles de solucionar con la tecnología usada por Angular 1, han sido los que han impulsado a sus creadores a desarrollar desde cero una nueva versión del framework. La nueva herramienta está pensada para dar cabida a todos los usos dados por los desarrolladores, llevar a Javascript a un nuevo nivel comparable a lenguajes más tradicionales, siendo además capaz de resolver de una manera adecuada las necesidades y problemas de la programación del lado del cliente.

2.1. Aportaciones de Angular 2

En la imagen de la figura 2.1 puedes ver algunas de las soluciones aportadas en Angular 2.

2.1.1. TypeScript / Javascript

Como base hemos puesto a Javascript, ya que es el inicio de los problemas de escalabilidad del código. Ayuda poco a detectar errores y además produce con facilidad situaciones poco deseables.

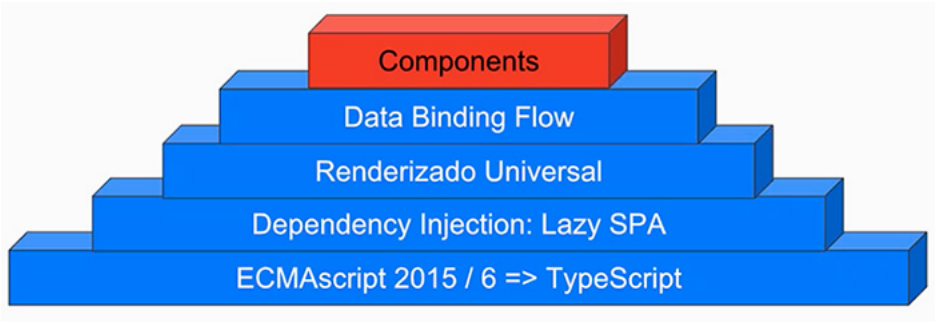

Figura 2.1: Soluciones aportadas por Angular 2

Con ECMAscript 6 ya mejora bastante el lenguaje, facilitando la legibilidad del código y solucionando diversos problemas, pero todavía se le exige más. Ya puestos a no usar el Javascript que entienden los navegadores (ECMAscript 5), insertando la necesidad de usar un transpilador como Babel, podemos subir todavía un poco de nivel y usar TypeScript. Angular 2 promueve el uso de TypeScript a sus desarrolladores. El propio framework está desarrollado en TypeScript, un lenguaje que agrega las posibilidades de ES6 y el futuro ES7, además de un tipado estático y ayudas durante la escritura del código, el refactoring, etc. pero sin alejarte del propio Javascript (ya que el código de Javascript es código perfectamente válido en TypeScript).

La sugerencia de usar TypeScript para desarrollar en Angular es casi una imposición porque la documentación y los generadores de código están pensados en TypeScript. Se supone que en futuro también estarán disponibles para Javascript, pero de momento no es así. De todos modos, para la tranquilidad de muchos, TypeScript no agrega más necesidad de procesamiento a las aplicaciones con Angular 2, ya que este lenguaje solamente lo utilizas en la etapa de desarrollo y todo el código que se ejecuta en el navegador es al final Javascript, ya que existe una transpilación previa.

2.1.2. Lazy SPA

Ahora el inyector de dependencias de Angular no necesita que estén en memoria todas las clases o código de todos los elementos que conforman una aplicación. En resumen, ahora con Lazy SPA el framework puede funcionar sin conocer todo el código de la aplicación, ofreciendo la posibilidad de cargar más adelante aquellas piezas que no necesitan todavía.

2.1.3. Renderizado Universal

Angular nació para hacer web y renderizar en HTML en el navegador, pero ahora el renderizado universal nos permite que no solo se pueda renderizar una vista a HTML. Gracias a ésto, alguien podría programar una aplicación y que el renderizado se haga, por ejemplo, en otro lenguaje nativo para un dispositivo dado.

Otra cosa que permite el renderizado universal es que se use el motor de renderizado de Angular del lado del servidor. Es una de las novedades más interesantes, ya que ahora podrás usar el framework para renderizar vistas del lado del servidor, permitiendo un mejor potencial de posicionamiento en buscadores de los contenidos de una aplicación. Esta misma novedad también permite reducir el impacto de la primera visita, ya que podrás tener vistas "precocinadas.[en] el servidor, que puedes enviar directamente al cliente.

2.1.4. Data Binding Flow

Uno de los motivos del éxito de Angular 1 fue el data binding, pero éste tenía un coste en tiempo de procesamiento en el navegador, que si bien no penalizaba el rendimiento en todas las aplicaciones sí era un problema en aquellas más complejas. El flujo de datos ahora está mucho más controlado y el desarrollador puede direccionarlo fácilmente, permitiendo optimizar las aplicaciones. El resultado es que en Angular 2 las aplicaciones pueden llegar a ser hasta 5 veces más rápidas.

2.1.5. Componentes

La arquitectura de una aplicación Angular ahora se realiza mediante componentes. En este caso no se trata de una novedad de la versión 2, ya que en la versión de Angular 1.5 ya se introdujo el desarrollo basado en componentes.

Sin embargo, la componetización no es algo opcional como en Angular 1.5, sino es una obligatoriedad. Los componentes son estancos, no se comunican con el padre a no ser que se haga explícitamente por medio de los mecanismos disponibles, etc. Todo esto genera aplicaciones más mantenibles, donde se encapsula mejor la funcionalidad y cuyo funcionamiento es más previsible. Ahora se evita el acceso universal a cualquier cosa desde cualquier parte del código, vía herencia o cosas como el “Root Scope”, que permitía en versiones tempranas de Angular modificar cualquier cosa de la aplicación desde

cualquier sitio.

2.2. Creando y configurando un proyecto

En este paso nosotros:

- Crearemos una carpeta del proyecto
- Agregar archivos de configuración y de definición de paquetes
- Instalar Paquetes

2.2.1. Crear una carpeta de proyecto

```
$ mkdir angular2-quickstart
$ cd    angular2-quickstart
```

2.2.2. Agregar un archivo de definición de paquetes y de configuración

Agregar los siguientes archivos de definición de paquetes y configuración a la carpeta del proyecto:

- *package.json* . - Lista los paquetes de los cuales el proyecto depende y define algunos scripts útiles.
- *tsconfig.json* .- Es el archivo de TypeScript de configuración del compilador.
- *typings.json* .- Identifica los archivos de definición de TypeScript.
- *systemjs.config.js* .- archivo de configuración de SystemJS.

2.2.3. Instalar Paquetes

Se instalan los paquetes listados en *package.json* usando npm. Introducir el siguiente comando:

```
$ npm install
```

Si la carpeta typings no se muestra después de ejecutar npm install. Entonces, lo que tiene que ejecutarse es:

```
$ npm run typing install
```

2.2.4. Agregando las librerías y paquetes necesarios con *npm*

Los desarrolladores de aplicaciones utilizan el manejador de paquetes *npm* para instalar librerías y paquetes que sus aplicaciones requieren. El equipo de Angular recomienda un conjunto inicial de paquetes especificado en las secciones **dependencies** y **devDependencies**. Se han incluido un número de scripts *npm* en el archivo propuesto **package.json** de la sub sección 2.2.6.

Muchos de los scripts de *npm* se ejecutan de la siguiente manera: **npm run** seguido por el **nombre-script**. Algunos comandos (como el comando **start**) no necesitan la palabra clave **run**. Lo que los script realizan es:

- **npm start** .- ejecuta el compilador y el servidor al mismo tiempo, ambos en "*watch mode*"
- **npm run tsc** .- ejecuta el compilador TypeScript una vez.
- **npm run tsc:w**.- ejecuta el compilador TypeScript en *watch mode*; el proceso se mantiene corriendo, esperando cambios del archivo TypeScript y recompilando cuando ve un cambio
- **npm run lite** .- ejecuta el servidor lite, un servidor de archivos ligero con excelente soporte para aplicaciones Angular usando *routing*.
- **npm run typings**.- ejecuta las herramientas *typings* separadamente.
- **npm run postinstall**.- llamado por *npm* automáticamente después de instalar los paquetes completamente. Este script instala los archivos de definición TypeScript definidos en *typings.json*

2.2.5. Archivo systemjs.config.js

```
1 /**
2 * System configuration for Angular 2 samples
3 * Adjust as necessary for your application needs.
4  */
5 (function(global) {
6   // map tells the System loader where to look for things
7   var map = {
8     'app':                          'app', // 'dist',
9     '@angular':                     'node_modules/@angular',
```

```
10      'angular2-in-memory-web-api': 'node_modules/angular2-in-
          memory-web-api',
11      'rxjs':                       'node_modules/rxjs'
12    };
13
14    // packages tells the System loader how to load when no
        filename and/or no extension
15    var packages = {
16      'app':                        { main: 'main.js',
          defaultExtension: 'js' },
17      'rxjs':                       { defaultExtension: 'js' },
18      'angular2-in-memory-web-api': { main: 'index.js',
          defaultExtension: 'js' },
19    };
20
21    var ngPackageNames = [
22      'common',
23      'compiler',
24      'core',
25      'forms',
26      'http',
27      'platform-browser',
28      'platform-browser-dynamic',
29      'router',
30      'router-deprecated',
31      'upgrade',
32    ];
33
34    // Individual files (~300 requests):
35    function packIndex(pkgName) {
36      packages['@angular/'+pkgName] = { main: 'index.js',
          defaultExtension: 'js' };
37    }
38    // Bundled (~40 requests):
39    function packUmd(pkgName) {
40      packages['@angular/'+pkgName] = { main: '/bundles/' +
          pkgName + '.umd.js', defaultExtension: 'js' };
41    }
42    // Most environments should use UMD; some (Karma) need the
        individual index files
43    var setPackageConfig = System.packageWithIndex ? packIndex
        : packUmd;
44
45
46     // Add package entries for angular packages
```

```
47   ngPackageNames.forEach(setPackageConfig);
48   var config = {
49     map: map,
50     packages: packages
51   };
52   System.config(config);
53 })(this);
```

2.2.6. Archivo package.json

```
1
2 {
3
4   "name": "angular2-quickstart",
5   "version": "1.0.0",
6   "scripts": {
7     "start": "tsc && concurrently \"npm run tsc:w\" \"npm run
        lite\" ",
8     "lite": "lite-server",
9     "postinstall": "typings install",
10    "tsc": "tsc",
11    "tsc:w": "tsc -w",
12    "typings": "typings"
13  },
14  "license": "ISC",
15  "dependencies": {
16    "@angular/common": "2.0.0-rc.5",
17    "@angular/compiler": "2.0.0-rc.5",
18    "@angular/core": "2.0.0-rc.5",
19    "@angular/forms": "0.3.0",
20    "@angular/http": "2.0.0-rc.5",
21    "@angular/platform-browser": "2.0.0-rc.5",
22    "@angular/platform-browser-dynamic": "2.0.0-rc.5",
23    "@angular/router": "3.0.0-rc.1",
24    "@angular/router-deprecated": "2.0.0-rc.2",
25    "@angular/upgrade": "2.0.0-rc.5",
26    "systemjs": "0.19.27",
27    "core-js": "^2.4.0",
28    "reflect-metadata": "^0.1.3",
29    "rxjs": "5.0.0-beta.6",
30    "zone.js": "^0.6.12",
31    "angular2-in-memory-web-api": "0.0.15",
32    "bootstrap": "^3.3.6"
33  },
34  "devDependencies": {
```

```
35     "concurrently": "^2.0.0",
36     "lite-server": "^2.2.0",
37     "typescript": "^1.8.10",
38     "typings": "^1.0.4"
39   }
40 }
```

2.2.7. Archivo tsconfig.json

```
1
2 {
3   "compilerOptions": {
4     "target": "es5",
5     "module": "commonjs",
6     "moduleResolution": "node",
7     "sourceMap": true,
8     "emitDecoratorMetadata": true,
9     "experimentalDecorators": true,
10    "removeComments": false,
11    "noImplicitAny": false
12  }
13 }
```

2.2.8. Archivo typings.json

```
1
2 {
3   "globalDependencies": {
4     "core-js": "registry:dt/core-js#0.0.0+20160602141332",
5     "jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
6     "node": "registry:dt/node#6.0.0+20160807145350"
7   }
8 }
```

2.3. Diseñando el primer componente en Angular

Primero vamos a crear una carpeta para colocar nuestra aplicación y agregar un componente Angular simple.

2.3.1. Crear una subcarpeta de la aplicación

Crear una carpeta en el directorio raíz para allí colocar todos los archivos de la aplicación.

```
1 $ mkdir app
```

2.3.2. Crear el archivo de componente

Se crea el archivo **app/app.component.ts** en el nuevo directorio creado con el siguiente contenido:

```
1 import { Component } from '@angular/core';
2 @Component({
3   selector: 'my-app',
4   template: '<h1>My First Angular 2 App</h1>'
5 })
6 export class AppComponent { }
```

Cada aplicación angular tiene al menos un componente raíz, convencionalmente llamado ***AppComponent***, que contiene la experiencia del usuario. Los componentes son los bloques fundamentales en las aplicaciones Angular. Un componente controla una parte de la pantalla- una vista- a través de los template asociados.

La estructura de cada componente contiene lo siguiente:

- Una o más sentencias ***import*** para la referencia a las librerías u objetos de lo que se necesite
- Un ***@Component decorator*** que le dice a Angular que template usar y como crear el componente
- A ***component class*** que controla la apariencia y comportamiento de una vista a través de su template.

Import

Las aplicaciones Angular son modulares. Consisten de muchos archivos cada uno dedicado a un propósito. Angular en si mismo en modular. Es una colección de módulos de librerías cada uno hecho de varios módulos a su vez.

Cuando necesitamos algo de un módulo o librería, la importamos. En este ejemplo importamos *Angular 2 core* a nuestro componente y se puede acceder con el decorador ***@Component***.

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

@Component decorator

Component es una función decoradora que toma un objeto *metadata* como argumento. Aplicamos esta función a la clase componente colocándole de prefijo a la función el símbolo @ invocándola con el objeto *metadata*.

@Component es un decorador que nos permite asociar *metadata* con una clase componente. Los *metadata* le dicen a Angular como crear y usar este componente.

```
1 @Component({
2   selector: 'my-app',
3   template: '<h1>My First Angular 2 App</h1>'
4 })
```

Este objeto metadata particular tiene dos campos, un *selector* y un *template*.

El **selector** especifica un simple **selector** CSS para un elemento HTML que representa el componente. Los elementos para este componente se llaman *my-app*. Angular crea y despliega una instancia de nuestro ***AppComponent*** en donde encuentra un elemento *my-app* en el archivo HTML.

El template especifica los formatos de los componentes, escrito en una forma mejorada de HTML que le dice a Angular como desplegar este componente. En este ejemplo nuestro template es una simple linea de HTML que anuncia "My First Angular 2 App". Un template mas avanzado puede contener enlaces de datos a las propiedades de los componentes y puede identificar otros componentes de aplicaciones los cuales tiene sus propios templates. Estos templates pueden identificar todavía otros componentes. De esta forma una aplicación Angular se convierte a un árbol de componentes.

2.3.3. Component class

En el fondo del archivo hay una clase vacía que no hace nada llamada **AppComponent**

```
1 $ export class AppComponent { }
```

Cuando estamos listos para construir una aplicación, podemos expandir esta clase con propiedades y aplicaciones lógicas. Nuestra ***AppComponent***

esta vacía porque no se necesita nada particular. Se exporta ***AppComponent*** de tal forma que podemos importarlo en cualquier lugar de nuestra aplicación, como se verá cuando se cree el archivo **app.module.ts**.

2.4. creando app.module.ts

Se crean aplicaciones Angular estrechamente relacionadas con bloques de funcionalidad con módulos angular. Cada aplicación requiere al menos un módulo, el módulo raíz, que se llama **AppModule** por convención.

Crear un archivo app/app.module.ts con el siguiente contenido:

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

Se pasan metadata a la función decoradora NgModule

- imports .- los otros módulos que exportan material que necesitamos en este módulo. Casi cada aplicación módulo raíz debe importar el **BrowserModule**.
- declarations .- Componentes y directivas que pertenecen a este módulo
- bootstrap .- identifica el componente raíz que angular. La palabra inglesa bootstrapping es generalmente un término utilizado para describir el arranque, o proceso de inicio de cualquier ordenador. Suele referirse al programa que arranca un sistema operativo como por ejemplo GRUB, LiLo (utilizados en sistemas GNU/Linux, por ejemplo), BCD o NTLDR (utilizados en sistemas Windows). Se ejecuta tras el proceso POST (power-on-self-test) del BIOS. También es llamado «Boot loader» (cargador de inicio).

Importamos nuestra **app.component.ts** y se agrega tanto a **declarations** y al arreglo **bootstrap**.

Agregamos el BrowserModule desde @angular/platform-browser al arreglo import. Este es el modulo Angular que contiene todo lo que necesitamos para correr nuestra aplicación en el navegador.

2.5. Agregar main.ts

Ahora es necesario decirle a Angular que cargue el modulo de aplicación. Para ello se crea el archivo **app/main.ts** con el siguiente contenido:

```
1 import { platformBrowserDynamic } from '@angular/platform-
    browser-dynamic';
2
3 import { AppModule } from './app.module';
4
5 platformBrowserDynamic().bootstrapModule(AppModule);
```

Se importan dos cosas necesarias para cargar la aplicación:

- La función Navegador Angular **platformBrowserDynamic**
- El módulo de aplicación **AppModule**

Entonces se llama **platformBrowserDynamic().bootstrapModule** con **AppComponent**

2.6. El Bootstraping es específico de la plataforma

Es importante puntualizar que importamos la función **platformBrowserDynamic** desde **@angular/platform-browser-dynamic** y no desde **@angular/core**. Bootstrapping no esta en el core porque no hay una manera simple para arrancar la aplicación. Muchas aplicaciones que corren en un navegador llaman a la función bootstrap desde esta librería. Pero es posible cargar un módulo en un ambiente diferente. Es posible cargar desde un dispositivo con Apache Cordova o NativeScript.

2.7. Agregar un archivo index.html

En la carpeta raíz del proyecto se crea un archivo index.html con el siguiente contenido:

```
<html>
  <head>
    <title>Angular 2 QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="styles.css">
    <!-- 1. Load libraries -->
     <!-- Polyfill(s) for older browsers -->
    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <!-- 2. Configure SystemJS -->
    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>
  <!-- 3. Display the application -->
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

El archivo index.html define la pagina web que alojará la aplicación. Las secciones del archivo HTML son:

- Las librerias de JavaScript
- El archivo de configuración para **SystemJS**, y el script donde se importa y ejecuta el módulo de la *app* que llama al archivo *main*
- El tag <**my-app**> en el <**body**> que es donde nuestro *app* vive.

2.7.1. Agregando algo de estilo

Los estilos no son esenciales pero son agradables, y el archivo **index.html** supone que tenemos un archivo de estilos llamado **style.css**.

Crear el archivo **style.css** en la carpeta raíz del proyecto para poder cargarlo con el código mínimo que se muestra a continuación:

```
/* Master Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
```

2.8. Construir y ejecutar la app

En una terminal ejecutar el siguiente comando:

```
npm start
```

El comando anterior ejecuta dos procesos en paralelo:

- El compilador TypeScript en **watch mode**
- Un servidor estático llamado **lite-server** que carga **index.html** en un navegador y refresca el navegador cuando el archivo cambia