Desarrollo de Aplicaciones Multiplataforma

Docente: Brian Ducca

Angular

Módulos

Los módulos en angular se declaran como clases Typescript, y tienen estructura de árbol jerárquico.
El módulo App también se conoce como **módulo raíz** porque de él surgen las demás ramas que conforman una aplicación.

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

En declarations se importan los componentes, pipes y directivas.
La asignación de los nodos hijos se realiza en la propiedad imports:[], que es un array de punteros a otros módulos.
En la sección providers, se definen los servicios que fueron decorados con la propiedad “Inyectable”, permitiendo lo llamado Inyección de Dependencias (DI).
Exports se utiliza para poner a disposición componentes a otros módulos. Por defecto, todo lo definido dentro del módulo es privado.
Para crear un módulo por medio del angular-cli, deberemos correr el comando:

```
ng generate module nombreModulo
```

Componentes

Un componente es un bloque básico de construcción de una página web, utiliza algunos conceptos del patrón MVC(Model-View-Controller) y otros del MVVM(model-view-view-model).

Cuando lo generamos automáticamente mediante el angular-cli, nos generará 4 archivos:

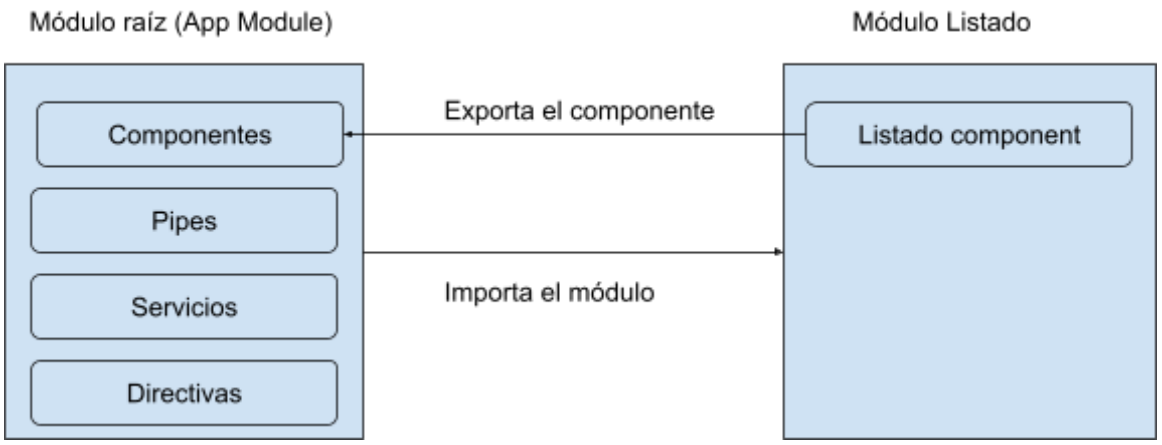
- Vista .html
- Estilo .css
- Controlador .ts
- Prueba unitaria .spec.ts

Si deseamos generar el componente de manera manual, debemos utilizar el decorador `@Component()` en una clase typescript, lo cual se debe importar de `@angular/core`, en el decorador tendremos que completar información sobre cuál será el selector, es decir como lo llamaremos el componente en otro html; el templateUrl que indicaremos de donde el componente tomará el HTML y por último el styleUrls que hará lo mismo para la hoja de estilo.
Los estilos se incrustan durante la compilación en los nodos del *DOM* generado.

```
@Component({
  selector: 'home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
```

Los componentes no deciden por sí mismos su **visibilidad**. Cuando un componente es generado se declara en un módulo contenedor en su propiedad declares:[], esto lo hace visible y utilizable por cualquier otro componente del mismo módulo.
Si queremos usarlo desde fuera tendremos que exportarlo. Eso se hace en la propiedad exports:[] del módulo en el que se crea.

Podemos ver un ejemplo donde tenemos el módulo raíz de nuestra aplicación y el módulo Listado, en el cual este último contiene un componente de listado, el cual lo va a tener que exportar en el atributo “exports” del módulo Listado para que luego nuestro módulo raíz lo pueda importar por medio de la propiedad “imports” para poder usar el componente Listado.



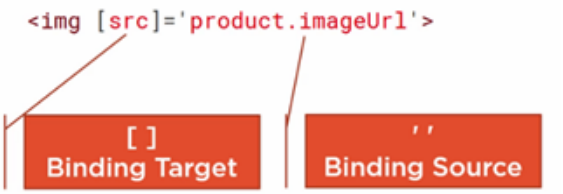
Para crear un componente por medio del angular-cli, deberemos utilizar la siguiente sentencia:

```
ng generate component nombreComponente
```

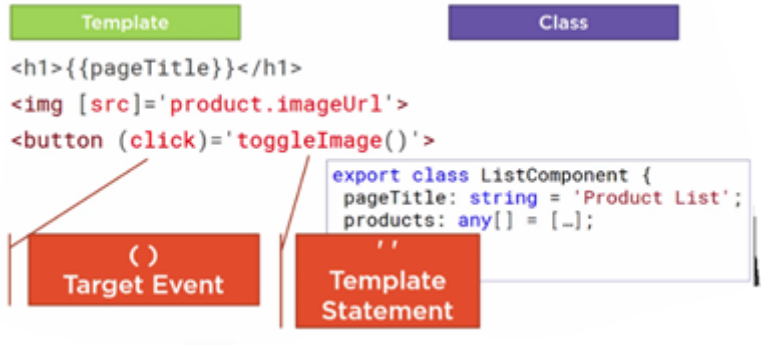
Bindings

Los bindings son los encargados de coordinar la comunicación entre la class y el template de un componente, para pasar datos.
Se utilizan para:

- Vincular propiedades (property binding): Este caso se da cuando queremos pasar datos desde la Class al Template. La propiedad a modificar en la vista se encierra en corchetes y la propiedad que se pasa por valor se encierra en comillas.



- Manejar eventos (event binding): aquí vamos a escuchar eventos, el cual se va a encerrar el nombre en paréntesis (Por ejemplo, el evento (click)), y se incluye el nombre del método a ejecutar entre comillas

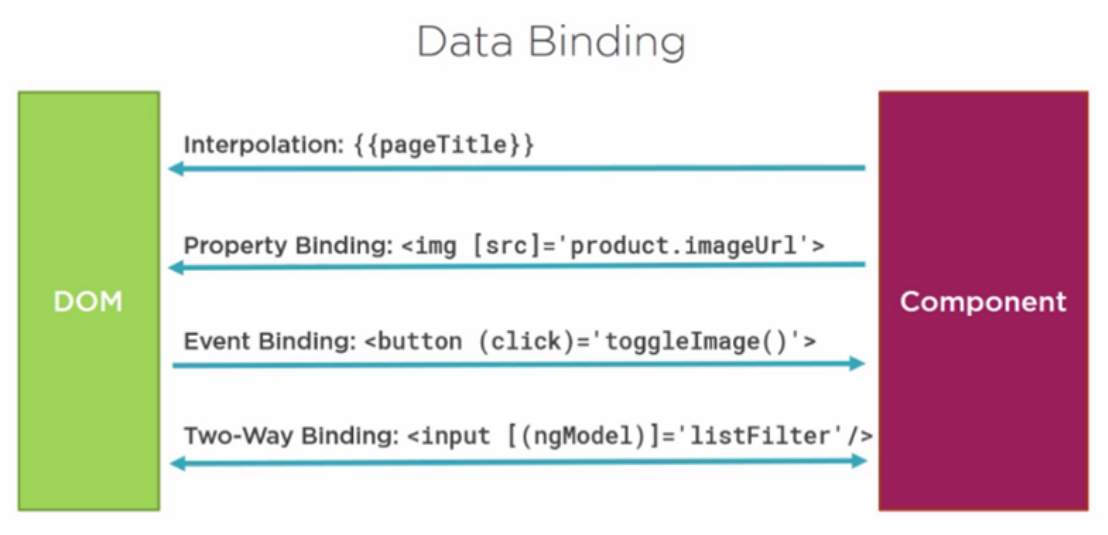


One Way Binding

Crea un canal de comunicación en una sola dirección, de la class a la vista o template, se encierra entre llaves. Se lo usa generalmente para vincular propiedades, cuando tenemos algún valor de un objeto en nuestra clase Typescript y lo queremos mostrar en la vista. Esto es conocido como Interpolación.

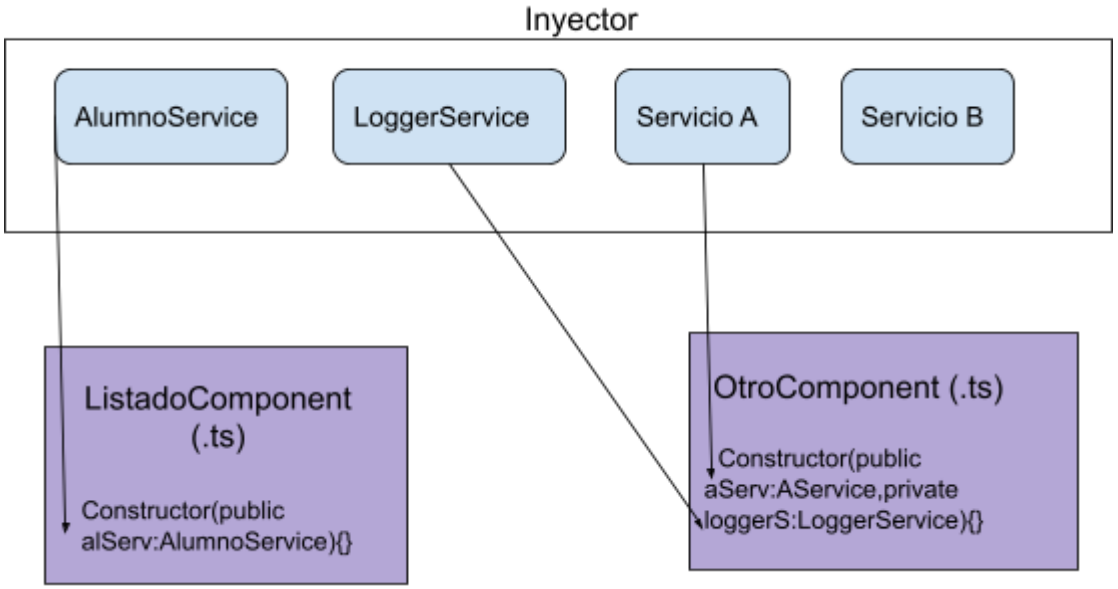
Two Way Binding

Crea un canal de comunicación en las dos direcciones, de la class a la vista y de la vista a la class. Se encierra primero entre corchetes para indicar el property binding (de la class al template) y luego se encierra entre paréntesis para expresar el event binding (del template a la class).
Esto se usa en casos en donde en un campo de texto nos interesa que pase el contenido escrito de la vista a los datos de nuestro objeto en la clase, como también, luego de ser procesado, la clase transmita los resultados para que sean visibles en la vista.



Servicios

Los servicios son clases typescript y su propósito yace en contener la lógica de negocio, clases para acceso a datos. Su particularidad es que lleva el decorador `@Injectable()` que proviene de `@angular/core` e indica que puede ser inyectada dinámicamente a quien la demande. Con este decorador, Angular reconoce que es un servicio y va a observar en cuales componentes tienen dependencia con este servicio y lo va a inyectar (Para ello, angular mira los constructores de la clase typescript de los componentes)
Los servicios se auto-proveen en el módulo raíz mediante la configuración providedIn:'root' de su decorador, para que luego el Inyector de Angular se encargue de cuando exista una dependencia con algún servicio, inyectarlo.
En el siguiente ejemplo veremos de forma gráfica cómo funciona el Inyector de Angular:



Http Service

El servicio Http nos lo brinda angular mediante la librería @angular/common/http la cual contiene el módulo HttpClientModule con el servicio inyectable HttpClient.

Para poder utilizarlo, debemos importar dicho módulo en nuestro app.module.ts, con esto ya formaría parte de nuestra aplicación, ahora para utilizarlo creamos nuestro propio servicio y en ese archivo typescript lo importaremos y lo declararemos dentro del constructor y quedará algo parecido a lo siguiente:

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

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

En el método get, su primer parámetro será la *url* a la que invocar.

El método *get* retorna un objeto observable. Los observables *http* han de consumirse mediante el método *subscribe* para que realmente se lancen.

Dicho método *subscribe* admite hasta tres *callbacks*: susbcribe(data, err, end) para que se ejecuten en respuesta a eventos

Luego tenemos el método post al que se le pasará la ruta del *end point* y el objeto *payload* que se enviará al servidor.

Pipe

Se utilizan para transformar los datos antes de mostrarlos, muchos de ellos vienen predefinidos en el framework, pero también podemos crear nuestros propios pipes.

```
{{ product.productCode | lowercase }}
```

Pueden recibir parámetros como por ejemplo el pipe por defecto “currency”

```
{{ product.price | currency:'USD':true:'1.2-2' }}
```

Los argumentos se indican mediante : y en este caso expresamos que queremos que el símbolo de la moneda sea USD, true implica que sí queremos que se muestre la moneda, y ‘1.2–2’, el primer dígito, el 1, indica el número mínimo de cifras enteras, el siguiente, el .2 indica el número mínimo de cifras decimales, y el último -2 indica el máximo de cifras decimales.

Los pipes tienen un decorador @Pipe donde debemos poner el valor en el atributo “name” para que sea identificable en el template del componente. Cuando creamos nuestro propio pipe por medio del angular-cli, se generará un archivo con la siguiente forma con el siguiente comando:

```
ng generate pipe nombrePipe
```

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
 name: 'nombrePipe'
})
export class CustomPipe implements PipeTransform {

 transform(value: any, args?: any): any {
   return null;
```

```
  }
}
```

En el método transform tendremos que cambiar la firma de los parámetros que recibe por los que queramos y luego su implementación para que devuelva lo que deseamos. Para poder utilizarla luego, tendremos que incluirla en el módulo en el atributo de “declarations”.
Por ejemplo, un pipe que eleva un número a la potencia que le pasemos por parámetro

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * El exponente por default es 1.
 * Uso:
 *   valor| elevar:exponent
 * Ejemplo:
 *   {{ 2 | elevar:10 }}
 *   devuelve: 1024
*/
@Pipe({name: 'elevar'})
export class ElevarPipe implements PipeTransform {
  transform(value: number, exponente?: number): number {
    return Math.pow(value, isNaN(exponente) ? 1 : exponente);
  }
}
```

Pipes default que vienen en angular: https://angular.io/api?type=pipe

Directivas

Las directivas de Angular son funciones que son invocadas cuando el DOM (Document Object Model) es compilado por el framework de Angular. Se podría decir que las directivas están vinculadas a sus correspondientes elementos del DOM cuando el documento es cargado. La finalidad de una directiva es modificar o crear un comportamiento totalmente nuevo.

```
<div *ngIf=”variable”>
    <p>Este es un párrafo</p>
</div>
```

Acción de la directiva *ngIf

Tipos de Directivas

- **Directivas estructurales:** Corresponden a elementos en el HTML que permiten añadir, manipular o eliminar elementos del DOM. Estos elementos, en forma de atributos, se aplican a *elementos huéspedes*. Al hacer esto, la directiva hace lo que debe hacer sobre el elemento huésped y sus elementos hijos. Estas directivas son fácilmente reconocibles debido a que están antecedidas por un asterisco (*) seguido del nombre de la directiva.
- **Directivas de atributo:** Una directiva de atributo cambia la apariencia o comportamiento de un elemento, componente u otra directiva.

Se pueden aplicar muchas *directivas de atributo* a un elemento huésped. Sin embargo, solo es posible aplicar una *directiva estructural* a un elemento huésped

Angular posee 2 directivas estructurales:
- *ngIf: Permite mostrar / ocultar elementos del DOM.
- *ngFor: Permite ejecutar bucles sobre elementos del DOM.

La tercera directiva, ngSwitch, es una directiva de atributo (pero que depende de otras directivas estructurales para funcionar):
- *ngSwitch: Permite ejecutar casos condicionales sobre elementos del DOM.

El prefijo * que antecede a las directivas permite a Angular envolver al elemento huésped en una etiqueta <ng-template> durante el proceso de transpilación de Angular CLI.

```
<ng-template [ngIf]="variable">
    <div>
        <p>Este es un párrafo</p>
    </div>
</ng-template>
```

Elemento generado con la directiva

Al utilizar una directiva estructural, el transpilador agrega un elemento <ng-template> que envuelve el elemento huésped

Ocurren dos cosas:

- La directiva (ngIf en este caso) pasa al elemento padre <ng-template>.
- El elemento huésped (incluyendo todos sus atributos), pasan a ser hijos de ng-template.

*ngIf

Esta directiva toma una expresión booleana y hace que toda la porción del DOM aparezca o desaparezca dada esa condición.

```
Evaluación Booleana
<p *ngIf="variable">Este es un párrafo</p>
```

La directiva *ngIf hace que aparezca o desaparezca el elemento del DOM. Esto es totalmente distinto a ocultar un elemento con CSS (ej.: display: none;), ya que el que elemento esté ausente indica que todo el *event binding* correspondiente a ese elemento estará ausente, mejorando así la performance de nuestra aplicación.
Adicionalmente, es posible utilizar operadores lógicos para evaluar la condición booleana de *ngIf, tales como &&, ||, etc.

*ngFor

Permite generar iteraciones de elementos HTML. Posee partes obligatorias y opcionales. Las partes obligatorias:

- Declaración de la variable que contiene el valor de la iteración
- Utilización de palabra of.
- Variable a iterar.

Las partes opcionales:

- Índice de la iteración
- Imprimir la variable que contiene el valor de la iteración con data binding.

```
var miArray = [1,2,3,4,5,6];
```

```
<div *ngFor="let elemento of miArray; let i = index">
    <p>El valor de párrafo es {{elemento}}</p>
</div>
```

Declaración de variable que contiene el valor de la iteración

Variable a iterar

Índice de la iteración

Valor de la iteración

*ngSwitch

Esta directiva corresponde a una serie de directivas que cooperan entre sí para generar un resultado. Estas directivas son ngSwitch, ngSwitchCase y ngSwitchDefault. La directiva ngSwitch es una directiva de atributo, mientras que las directivas ngSwitchCase y ngSwitchDefault corresponden a directivas estructurales.

```
<div [ngSwitch]="evaluacion">
    <p *ngSwitchCase="1">Muestra este párrafo genial</p>
    <p *ngSwitchCase="2">Muestra este otro párrafo genial</p>
    <p *ngSwitchDefault>Muestra este párrafo por defecto</p>
</div>
```

Evaluación del valor de una variable

Casos posibles para el valor

Caso por defecto para el valor

Directivas Custom

Para crear nuestras propias directivas, lo haremos por medio del angular-cli y correremos el siguiente comando:

```
ng generate directive nombreDirectiva
```

Con ello tendremos generado nuestra directiva, que en este caso implementaremos. Para este ejemplo elegimos una directiva de atributo que modifique el color del fondo de un elemento. Nuestro archivo directiva-atributo.directive.ts quedará de la siguiente manera:

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
 selector: '[appDirectivaAtributo]'
})
export class DirectivaAtributoDirective {

 constructor(private el:ElementRef) {
    el.nativeElement.style.backgroundColor = 'red';
 }
```

En donde el atributo que recibe en el constructor (**ElementRef**) hace referencia a un elemento del DOM, en el cual aplicaremos la directiva.
En el selector que nos genera Angular automáticamente ([appDirectivaAtributo]) lo encierra entre corchetes [] dado que lo reconoce como un selector de atributos, es decir que Angular va a buscar entre los atributos del html algo de nombre (en este ejemplo) appDirectivaAtributo, al cual le va a aplicar la lógica de la directiva.

También podremos capturar eventos de mouse enter por ejemplo para cambiar el estilo y darle un poco más de complejidad a la directiva (gracias a HostListener), quedando por ejemplo de la siguiente manera:

```
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
 selector: '[appDirectivaAtributo]'
})
export class DirectivaAtributoDirective {

 constructor(private el:ElementRef) {}

 @HostListener('mouseenter') onMouseEnter(){
   this.cambiar('yellow');
 }

 @HostListener('mouseleave') onmouseleave(){
   this.cambiar(null);
```

```
 }
 private cambiar(color:string){
   this.el.nativeElement.style.backgroundColor=color;
 }
}
```

Con esto, cada vez que pasemos con el mouse sobre el elemento al que esté aplicado la directiva, cambiará de color a amarillo y luego cuando salgo, volverá a no tener color en su background.

Para crear una directiva estructural la forma es la misma, pero debemos importar otras clases que serán necesarias para cualquier directiva estructural, estos son ViewContainerRef y TemplateRef.

TemplateRef tal como el nombre lo sugiere es una referencia a la plantilla <ng-template> donde es aplicada la directiva estructural.

Por esta misma razón y a diferencia de las directivas de atributo, no es posible aplicar más de una directiva estructural a un elemento.

Angular nos provee de un mecanismo para generar vistas e inyectarlas en un lugar específico del árbol de componentes. De esto se encarga ViewContainerRef, que representa un contenedor donde una o más vistas pueden ser adjuntadas.

Al inyectar **ViewContainerRef** en nuestra directiva estructural a través de constructor, le estamos dando la capacidad de generar una vista y modificar su contenido en el lugar donde es aplicada.

Entonces, nuestra directiva estructural nos quedaría de la siguiente manera:

```
import { Directive,TemplateRef, ViewContainerRef,Input } from
'@angular/core';


@Directive({
 selector: '[appDirectivaEst]'
})
export class DirectivaEstDirective {


 constructor(private templateRef: TemplateRef<any>, private viewContainer:
ViewContainerRef) {}
```

```
@Input() set appDirectivaEst(numero:number){
  for(let i =0;i<numero;i++){
    this.viewContainer.createEmbeddedView(this.templateRef);
  }
 }
}
```

En esta directiva, por medio del decorador @Input vamos a recibir en el setter de nuestra directiva (appDirectivaEst) un número que nos va a servir para en este caso crear n cantidad de elementos.

Reactive Forms

Reactive Forms es un módulo que nos provee Angular para definir formularios de una forma inmutable y reactiva. Por medio de este módulo podemos construir controles dentro de un formulario y asociarlos a las etiquetas HTML del template sin necesidad de usar explícitamente un ngModel.

Los formularios reactivos nos permiten crear e inicializar los objetos de control del formulario en nuestra clase de componentes, y estos escuchan los cambios en los valores de control de entrada y reflejan el estado del objeto, es decir, vamos a trabajar a nivel de componente y las validaciones estarán en nuestro archivo .typescript del componente.

El desarrollo utilizando formularios manejados por plantillas es bastante sencillo: básicamente consiste en enlazar los datos (data binding) desde el componente (código typescript) hasta la plantilla (código html) a través de la sintaxis [(ngModel)] en cada elemento html a enlazar.

En formas reactivas, crea el árbol de control de formulario completo en el código. Podemos actualizar inmediatamente un valor o desglosar los descendientes del formulario principal porque todos los controles están siempre disponibles.

Los formularios basados en plantillas delegan la creación de sus controles de formulario a las directivas. Para evitar errores " modificados después de comprobados ", estas directivas tardan más de un ciclo en construir todo el

árbol de control. Eso significa que debe esperar un tic antes de manipular cualquiera de los controles desde dentro de la clase de componentes.

Para implementar reactive forms en angular, vamos a necesitar importar el módulo `ReactiveFormsModule` de `@angular/forms` y añadirlo en nuestro app.module.ts.

Vamos a necesitar un `FormBuilder` que es un servicio para construir formularios creando un `FormGroup` que no es más que un grupo de controles (los cuales serán los necesarios para el formulario que necesitemos). Este último es el encargado de realizar un seguimiento del valor, estado de cambio y validez de los datos.

Cuando creamos el FormGroup mediante el FormBuilder, en cada control del grupo que especifiquemos deberemos colocar un valor default y el tipo de validación que aplicará; por ejemplo `nombre:['',Validators.required]` .

Entonces, el formulario se define como un grupo de controles, en el cual, cada control tendrá un nombre y una configuración. Esta definición nos permitirá también establecer un valor inicial en el control.

Acá vemos un ejemplo:

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from
'@angular/forms';

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

  public formGroup:FormGroup;

  constructor(private formBuilder: FormBuilder) { }

  ngOnInit(): void {
```

```
    this.crearForm();
  }

  private crearForm(){
    this.formGroup=this.formBuilder.group({
     nombre:['',Validators.required],
     ubicacion:['',Validators.required],
    });
  }

  public guardarDispositivo(){
    console.log(this.formGroup.get("nombre").value);
    console.log(this.formGroup.get("ubicacion").value);
  }

}
```

Y en la vista:

```
<form [formGroup]="formGroup" (submit)="guardarDispositivo()">
    <label for="nombre">Nombre:</label>
    <input name="nombre" formControlName="nombre" type="text" />
    <div class="text-danger"
*ngIf="formGroup.controls.nombre.invalid &&
formGroup.controls.nombre.touched">
        El nombre es requerido
    </div>
    <label for="ubicacion">Ubicación:</label>
    <input name="ubicacion" formControlName="ubicacion"
type="text" />
```

```
    <div class="text-danger"
*ngIf="formGroup.controls.ubicacion.invalid &&
formGroup.controls.ubicacion.touched">
        La ubicación es requerida
    </div>
    <br>
    <button type="submit" class="btn btn-primary"
[disabled]="formGroup.invalid">Guardar</button>

</form>



<p>Form value {{ formGroup.value | json }} </p>
<p> Form status {{ formGroup.status | json}} </p>
```

Tal como vemos en el ejemplo, luego de que se realiza un submit en el formulario, para obtener el valor de cada control, necesitaremos acceder a dicho control por medio del método “get” del FormGroup pasándole el nombre del control y luego ahí podremos acceder a su atributo “valor”, por ejemplo `this.formGroup.get("nombre").value`

Validación y estados

Los formularios y controles reactivos están gestionados por **máquinas de estados** que determinan en todo momento la situación de cada control y del formulario en sí mismo.

La máquina de estados de validación contempla los siguientes estados mutuamente excluyentes:

- VALID: el control ha pasado todas las validaciones
- INVALID: el control ha fallado al menos en una regla.
- PENDING: el control está en medio de un proceso de validación
- DISABLED: el control está desactivado y exento de validación

Cuando un control incumple con alguna regla de validación, estas se reflejan en su propiedad **errors** que será un objeto con una propiedad por cada regla insatisfecha y un valor o mensaje de ayuda guardado en dicha propiedad.

Los controles, y el formulario, se someten a otra máquina de estados que monitoriza el valor del control y sus cambios.

La máquina de estados de cambio contempla entre otros los siguientes:

- PRINSTINE: el valor del control no ha sido cambiado por el usuario
- DIRTY: el usuario ha modificado el valor del control.
- TOUCHED: el usuario ha tocado el control lanzando un evento blur al salir.
- UNTOUCHED: el usuario no ha tocado y salido del control lanzando ningún evento blur.

Como en el caso de los estados de validación, el formulario también se somete a estos estados en función de cómo estén sus controles.

Comunicación entre componentes

@Input & @Output

Son directivas para intercambiar datos entre componentes, @Input se encargará de recibir datos y @Output se encargará de enviar datos, pero este último enviará datos por medio de eventos por medio de objetos EventEmitter, entonces si tenemos algo como esto:

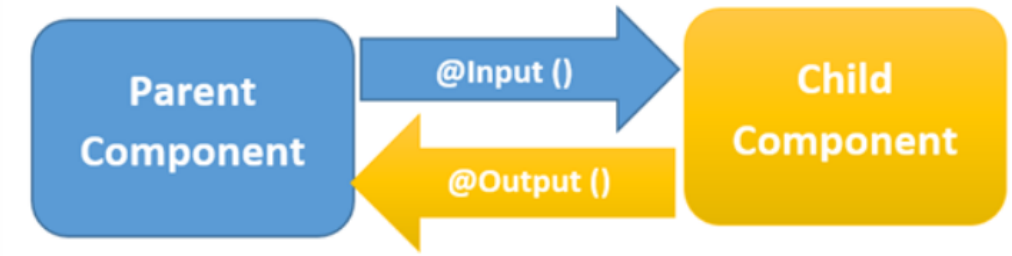
```
@Component({
 selector: 'alumno',
 ...
})
export class AlumnoComponent {
 @Input()  alumno
 @Output() onChange = new EventEmitter()
}
```

Esto quiere decir, que espera recibir datos en la propiedad “alumno” y que, a su vez, este componente producirá y enviará datos por medio de la propiedad “onChange”.
Entonces si tenemos un listado de alumnos, en su .html del componente del listado tendremos algo así:

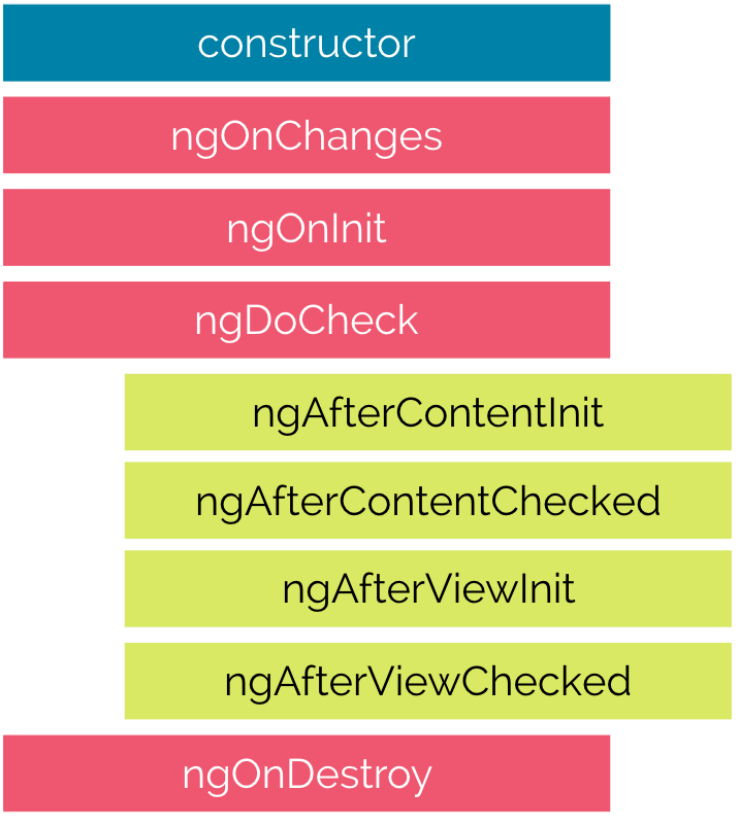
```
...
<alumno
 [alumno]="alumnoListado"
 (onChange)="manejarCambios($event)"
```

```
</alumno>
```
…

Esto significa que nuestra propiedad alumno, le va a llegar el valor “alumnoListado” de nuestro componente ListadoAlumno y que los cambios que se realicen en el componente Alumno, serán manejados en el evento “manejarCambios” del componente ListadoAlumno.



Ciclo de vida Angular

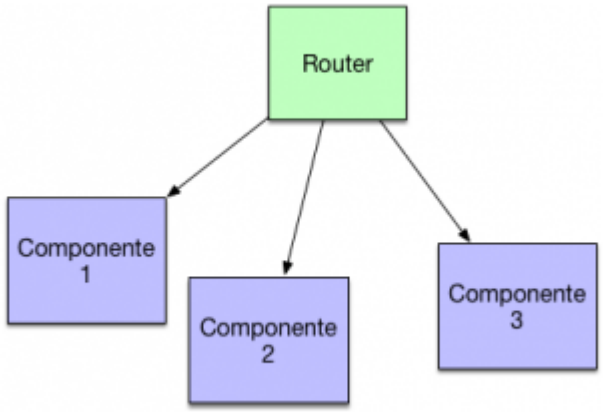


Cada componente tiene un ciclo de vida, una cantidad de etapas diferentes que atraviesa. Hay 8 etapas diferentes en el ciclo de vida de los componentes. Cada etapa se denomina lifecycle hook event o en 'evento de enlace de ciclo de vida'. Como un componente es una clase de TypeScript, cada componente debe tener un método constructor.
El constructor de la clase de componente se ejecuta primero, antes de la ejecución de cualquier otro lifecycle hook. Si necesitamos inyectar dependencias en el componente, el constructor es el mejor lugar para hacerlo. Después de ejecutar el constructor, Angular ejecuta sus métodos de enganche de ciclo de vida en un orden específico.

Estas etapas están divididas principalmente en dos fases, una vinculada al componente en si y la otra vinculada a los hijos del componente.
Dichos eventos son eventos:

- ngOnChanges: Este evento se ejecuta cada vez que se cambia un valor de un input control dentro de un componente.
- ngOnInit: Se ejecuta una vez que Angular ha desplegado los data-bound properties(variables vinculadas a datos) o cuando el componente ha sido inicializado, una vez que ngOnChanges se haya ejecutado. Este evento es utilizado principalmente para inicializar la data en el componente.
- ngDoCheck: Se activa cada vez que se verifican las propiedades de entrada de un componente. Este método nos permite implementar nuestra propia lógica o algoritmo de detección de cambios personalizado para cualquier componente.
- ngAfterContentInit: Se ejecuta cuando Angular realiza cualquier muestra de contenido dentro de las vistas de componentes y justo después de ngDoCheck. Actuando una vez que todas las vinculaciones del componente deban verificarse por primera vez. Está vinculado con las inicializaciones del componente hijo.
- ngAfterContentChecked: Se ejecuta cada vez que el contenido del componente ha sido verificado por el mecanismo de detección de cambios de Angular; se llama después del método ngAfterContentInit. Este también se invoca en cada ejecución posterior de ngDoCheck y está relacionado principalmente con las inicializaciones del componente hijo.
- ngAfterViewInit: Se ejecuta cuando la vista del componente se ha inicializado por completo. Este método se inicializa después de que Angular ha inicializado la vista del componente y las vistas secundarias. Se llama después de ngAfterContentChecked. Solo se aplica a los componentes.
- ngAfterViewChecked: Se ejecuta después del método ngAfterViewInit y cada vez que la vista del componente verifique cambios. También se ejecuta cuando se ha modificado cualquier enlace de las directivas secundarias. Por lo tanto, es muy útil cuando el componente espera algún valor que proviene de sus componentes secundarios.
- ngOnDestroy: Este método se ejecutará justo antes de que Angular destruya los componentes. Es muy útil para darse de baja de los observables y desconectar los event handlers para evitar memory leaks o fugas de memoria.

Angular Router



Angular router se encarga de decidir que componentes se muestran en cada momento de nuestra aplicación.

Elementos básicos que forman parte del routing en angular:

- El módulo llamado RouterModule.
- Rutas de la aplicación: es un array con un listado de rutas que nuestra aplicación soportará.
- Enlaces de navegación: son enlaces HTML en los que incluiremos una directiva para indicar que deben funcionar usando el sistema de routing.
- Contenedor: donde colocar las pantallas de cada ruta. Cada pantalla será representada por un componente.

Para que todo esto funcione, necesitaremos de una directiva que está disponible desde la librería de router, donde inserta los componentes necesarios cuando matchean con la URL. Esta directiva es router-outlet

Las rutas son definiciones de path y atributos de un componente, el path hace referencia a una URL que determina una única vista que va a ser mostrada y el componente hace referencia a un componente Angular que va a tener que ser asociado al path.

El path puede tener un wildcard (**) que el router va a seleccionar cuando no se matchee con ninguna ruta definida.Esto se puede utilizar para mostrar la clásica página de 404 Not Found.

Ejemplo de definición de ruta:

```
{ path:  'contacts', component:  ContactListComponent}
```

Si definimos esto en la configuración del router, el router va a renderizar el componente ContactListComponent cuando en la URL del browser de la aplicación sea “/contacts”.

Parámetros en las Rutas:

Tenemos dos formas de pasar parámetros dentro de las rutas:

- Por medio del servicio ActivatedRoute
- Usando el observable ParamMap

Ejemplo:

```
{ path:  'contacts/:id', component:
ContactDetailComponent}
```

Route Guard

```
ng generate guard nombreGuard
```

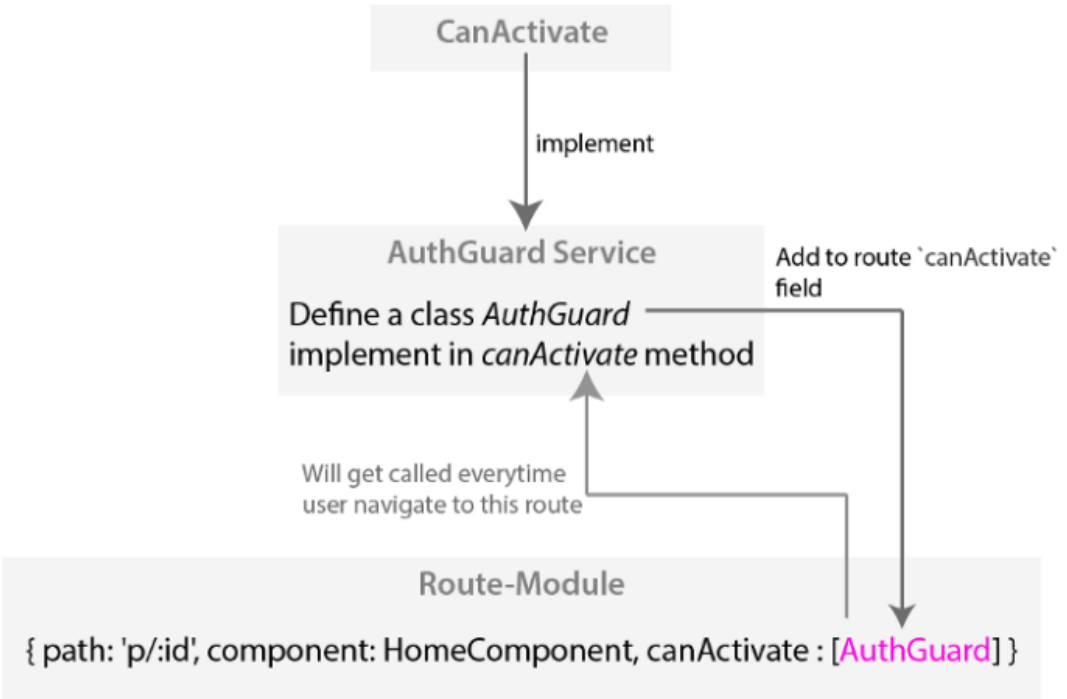
El route guard es una característica del Angular Router que nos permite ejecutar algún tipo de lógica cuando se solicita una ruta, y basado en esa lógica, permitimos o denegamos el acceso al usuario a esa ruta. Comúnmente es utilizado para verificar si un usuario está logueado o no en nuestro sistema para verificar si tiene autorización para acceder a esa URL.

El route guard se puede agregar implementando la interfaz CanActivate disponible en @angular/router y allí implementar el método canActivate() que contendrá la lógica para denegar o permitir el acceso a la ruta.

```
class MyGuard implements CanActivate {
  canActivate() {
    return true;
  }
}
```

Luego de esto, protegeremos la ruta con el guard utilizando el atributo canActivate en la definición de la ruta:

```
{ path:  'contacts/:id, canActivate:[MyGuard], component:
ContactDetailComponent}
```



Directiva de navegación

El router de angular nos da una directiva llamada routerLink que nos permite crear links de navegación, esta directiva agarra el path asociado al componente para navegar hacia él.

```
<a [routerLink]="'/contacts'">Contacts</a>
```

HTTP Interceptors

Como su nombre mismo dice, “intercepta” la petición HTTP, la modifica (si se requiere) y entonces continúa su camino. Para ello debemos implementar la interfaz HttpInterceptor y otras clases que se encuentran dentro del package ‘@angular/common/http’ y luego se implementa el método intercept definido en dicha interfaz.

```
ng generate service nombreInterceptor
```

Los interceptors se suelen utilizar mucho en conjunto con el angular guard, para generar soluciones de autenticación junto con otras herramientas como por ejemplo JWT.

En esos casos, una vez que desde el backend viene el token que generó luego de ingresar usuario y contraseña en un método POST, lo guardaremos en el localStorage, y luego de ello, utilizaremos el interceptor para que en cada petición HTTP que realizemos, incluya el token de JWT en la cabecera para que el backend realice las verificaciones correspondientes.

Ejemplo:

```
import { Injectable } from '@angular/core';
import { HttpEvent, HttpHandler, HttpInterceptor,
HttpRequest } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
   const token = localStorage.getItem('auth_token');
```

```
if (!token) {
     return next.handle(req);
   }
const headers = req.clone({
     headers: req.headers.set('Authorization', `Bearer
${token}`)
   });
return next.handle(headers);
 }
}
```

Cuando implementamos el método intercept, lo que haremos es obtener la petición y modificarla para luego continuar su camino. Siguiendo el común uso de estas herramientas, en este paso deberíamos insertar en la cabecera HTTP el token que nos vino del backend y que tenemos guardado en el localStorage.

Una vez que creamos nuestro interceptor, deberemos integrarlo en el módulo principal de la aplicación (`app.module`) y lo meteremos dentro del array de providers. Esto lo realizamos dado que los interceptors se ejecutan en cada petición que se realiza al servidor, siempre y cuando sean registrados. Para registrar un interceptor, debe estar declarado en el array de providers en el módulo raíz. (Necesitaremos importar HTTP_INTERCEPTORS y HttpClientModule)

```
import { HttpClientModule,HTTP_INTERCEPTORS} from
'./interceptors/auth.interceptor';

import { AuthInterceptor } from
'./interceptors/auth.interceptor';

@NgModule({
 declarations: [
   ...
 ],
 imports: [
   ...,
   HttpClientModule
 ],
 providers: [
   { provide: HTTP_INTERCEPTORS, useClass:
AuthInterceptor, multi: true }
```

```
  ],
  exports: [
    ...
  ]
})
```

Podemos observar que en el array de providers lleva la propiedad “multi” en true, esto nos permite agregar más interceptors si lo necesitáramos, de lo contrario, se sobreescribirá.

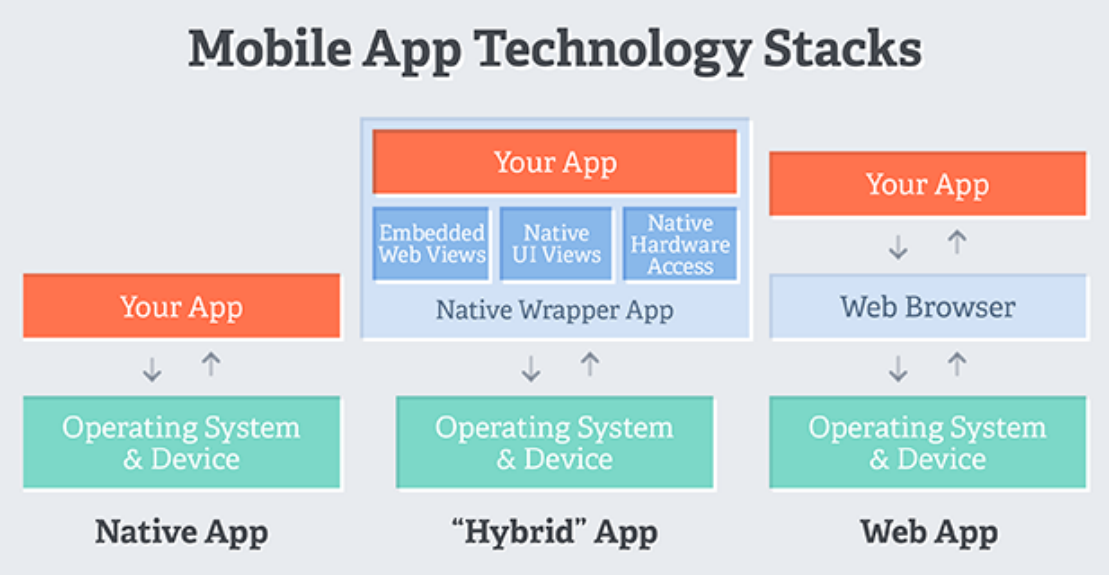
Aplicaciones Híbridas vs Nativas

Las aplicaciones nativas son las utilizadas con el lenguaje específico de cada plataforma, por ejemplo, Java para Android y Objective-C o Swift para iOS. Brindan una mejor experiencia de uso en comparación a las híbridas y en términos de performance son mucho más eficientes.

Las aplicaciones móviles híbridas son una combinación de tecnologías web como HTML, CSS y JavaScript, que no son ni aplicaciones móviles verdaderamente nativas, porque consisten en un WebView ejecutado dentro de un contenedor nativo, ni tampoco están basadas en Web, porque se empaquetan como aplicaciones para distribución y tienen acceso a las APIs nativas del dispositivo. Su característica principal es que es multiplataforma, es decir, se codea en una base y se compila para los diferentes sistemas operativos (Android, iOS, Windows Phone).

Los beneficios de una aplicación híbrida, pasan por la corta curva de aprendizaje ya que es sobre un lenguaje conocido, y lo que se tiene que aprender es sólo lo referente al framework para generar la aplicación híbrida y ya con el código base podemos compilar a cualquier plataforma. Junto con este beneficio, trae aparejado un coste mucho menor al desarrollar una aplicación híbrida que una nativa.

Entonces, optamos por aplicaciones nativas cuando queremos focalizar en brindar la mejor experiencia y performance en nuestra aplicación.
Y utilizaremos las aplicaciones híbridas cuando queremos llegar a una audiencia mucho mayor, con un costo mucho menor.



Ionic

Ionic es un framework open source basado en Angular, que se utiliza para el desarrollo de aplicaciones híbridas. La versión original fue lanzada en 2013 y construida sobre AngularJS y Apache Cordova.

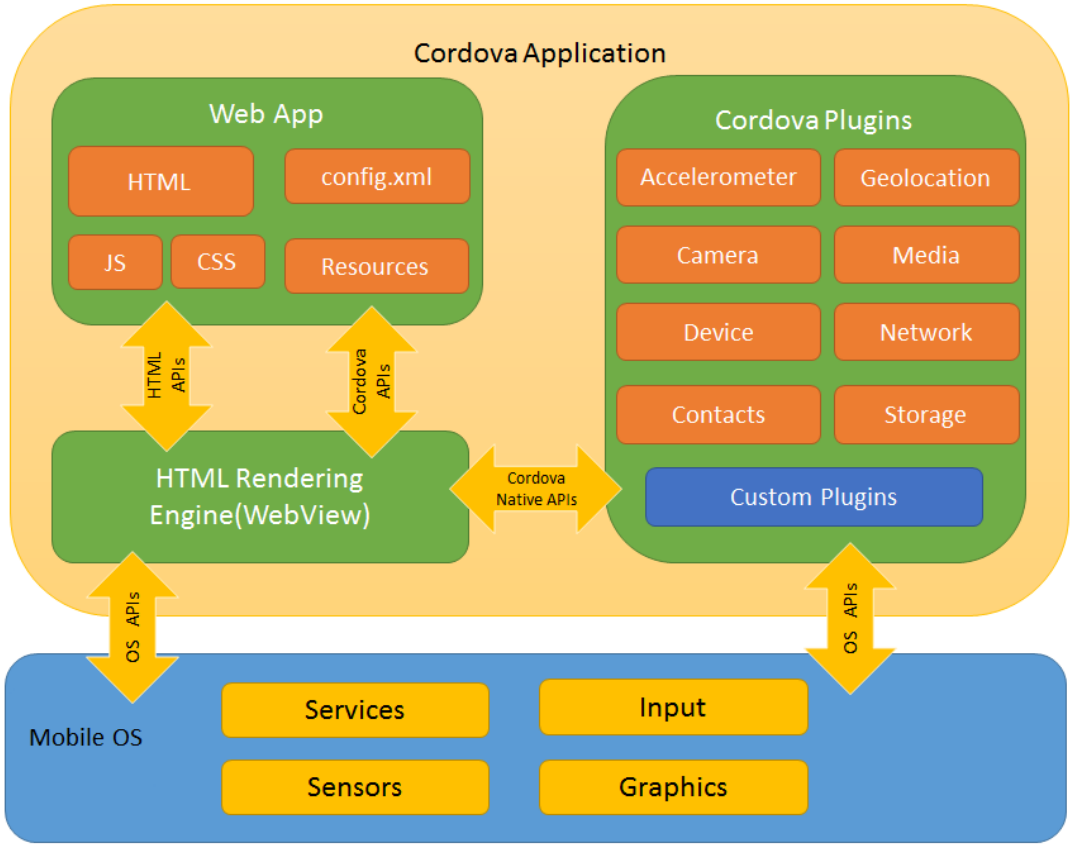
Cordova

Apache cordova es un framework de desarrollo donde se utiliza HTML, CSS y JS para el desarrollo multiplataforma, evitando el uso de los lenguajes nativos para cada plataforma. La aplicación se implementa como una página web, un archivo local llamado index.html, que hace referencia a cualquier CSS, JavaScript, imágenes u otros recursos necesarios para que se ejecute de forma predeterminada y corra sobre un WebView que está dentro de un wrapper nativo de cada plataforma para poder distribuirlo en los app-store.

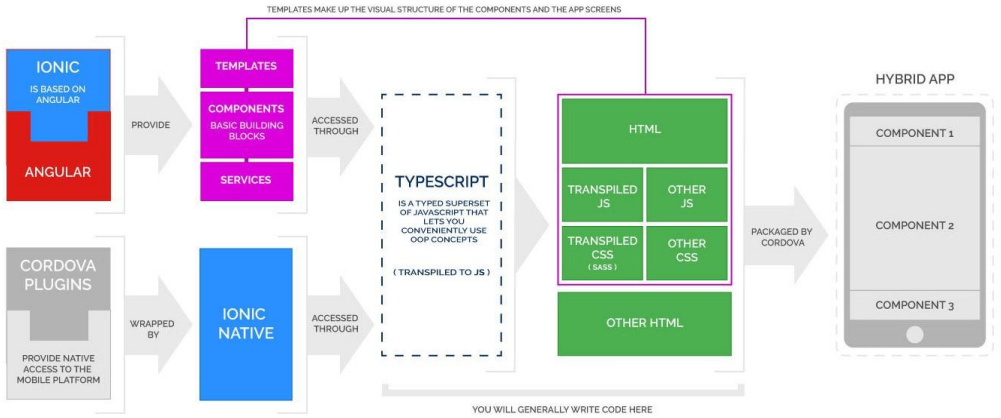
El archivo config.xml contendrá toda la información sobre la aplicación y los parámetros que afectarán a su funcionamiento, incluyendo comportamientos específicos para cada plataforma.

En el ecosistema de Cordova, los plugins son muy importantes dado que son la interfaz entre los componentes nativos y la aplicación, permitiéndonos ejecutar código nativo en Javascript.
Apache cordova tiene una serie de plugins que son denominados “Core Plugins” que están en todas las plataformas (contactos, batería, cámara,

acelerómetro, etc.). Adicionalmente existen librerías de terceros que nos dan acceso a funcionalidades que no necesariamente estén en todas las plataformas.



Ionic se basa en la interfaz del usuario y nos provee de una serie de herramientas para realizar aplicaciones al estilo nativo de una manera más fácil.



Capacitor

Capacitor es un proyecto Open Source que nos permite correr Web Apps de manera nativa en iOS,Android , a diferencia de Cordova, Capacitor incluye también a Electron y las PWA(Progressive Web App)

A diferencia de Cordova, Capacitor se centra más en trabajar sobre el proyecto de manera nativa, es decir, nos permite en el caso de que necesitemos de una manera fácil, poder agregar código nativo directamente al proyecto. Entonces, en el caso de que no encontremos algún plugin que resuelva algún problema que se nos presente, podremos buscar la solución al mismo directamente de la manera nativa. También nos permite mantener

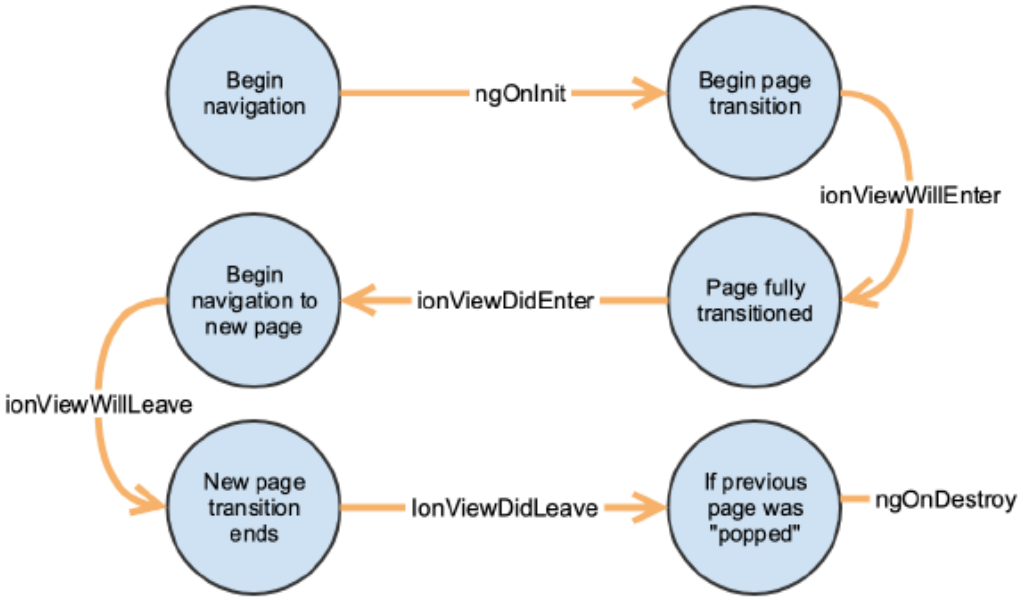
nuestra app actualizada sin necesidad de estar esperando que plugins de terceros actualicen la versión, lo podremos hacer nosotros mismos de la manera nativa. Aparejado a esto, no necesitaremos más el config.xml de cordova dado que utilizaremos los archivos de configuración específicos en cada plataforma nativa.

Como otra diferencia contra Cordova, encontramos que Capacitor utiliza al 100% NPM, es decir elimina completamente la instalación de plugins como hacíamos en cordova (`ionic cordova plugin add ...`) y pasamos a realizar instalaciones de package por medio de NPM.

Ciclo de vida de las page de Ionic

Ionic trabaja con el ciclo de vida que tiene Angular y le agrega algunos propios, los métodos más utilizados respecto del ciclo de vida son los siguientes:

- Angular
  - `ngOnInit:` Evento que se lanza luego de la inicialización del componente. Se utiliza generalmente para hacer las llamadas a los servicios que deben traer información una única vez.
  - `ngOnDestroy:`Se ejecuta luego de que Angular destruye a la view, generalmente usado para desuscribirse de los observables
- Ionic
  - `ionViewWillEnter:`Evento que se dispara cuando comienza a realizarse la animación de transición de la página
  - `ionViewDidEnter:`Evento que se dispara cuando terminó la animación y ya estamos en la página
  - `ionViewWillLeave:`Se dispara cuando comienza la transición a una nueva page, es decir, estoy saliendo de la página que estaba
  - `ionViewDidLeave:`Se dispara cuando termina la transición a la nueva página y salio por completo de la anterior.



Errores comunes al incorporar un proyecto nativo

Android

Al instalar el android studio(bumblebee) para poder levantar el proyecto android de manera nativa y compilarlo, podemos encontrarnos con el error que no encuentre el JAVA_HOME, para ello debemos asegurarnos que exista la variable de entorno de la misma. Por ejemplo, para macOS deberíamos setearla en caso de que no exista de la siguiente manera:

Versiones anteriores a BigSur: `export JAVA_HOME=/Applications/Android\ Studio.app/Contents/jre/jdk/Contents/Home/`

Versiones Catalina, BigSur y posteriores: `export JAVA_HOME=/Applications/Android\ Studio.app/Contents/jre/Contents/Home`

Para verificar su funcionamiento, vamos a realizar `java -version`

iOS

Al incorporar el proyecto nativo en iOS, nos podremos encontrar con problemas dentro del workspace con CocoaPods, lo notaremos porque al abrir el proyecto, nos marcará que la subparte Pods estará en rojo. Para solucionar eso primero deberemos verificar tener instalado CocoaPods(https://cocoapods.org/) , luego

correremos los siguientes comandos parados dentro de la carpeta del proyecto nativo iOS:

```
pod deintegrate
pod install
```

Comandos Ionic

Compilar y servir : `ionic serve`
Si queremos verlo en modo multiplataforma: `ionic serve --lab`
Crear una nueva page: `ionic generate page nombrePage`
Crear un nuevo módulo: `ionic generate module nombreMod`
Crear un nuevo service : `ionic generate service nombreServ`

Capacitor

Agregar una plataforma nativa a nuestro proyecto : `ionic capacitor add android/ios`
Eliminar una plataforma nativa de nuestro proyecto : `ionic cordova platform rm android/ios`
Compilar para android/ios: `ionic capacitor build android/ios`
Copiar web assets a las plataformas nativas: `ionic capacitor copy android/ios`
Abrir proyecto con el IDE nativo (Xcode para iOS, Android Studio para Android): `ionic capacitor open android/ios`
Correr en android/ios: `ionic capacitor run android/ios`
Realizar una sincronización del proyecto (copy + update): `ionic capacitor sync android/ios`

Cordova

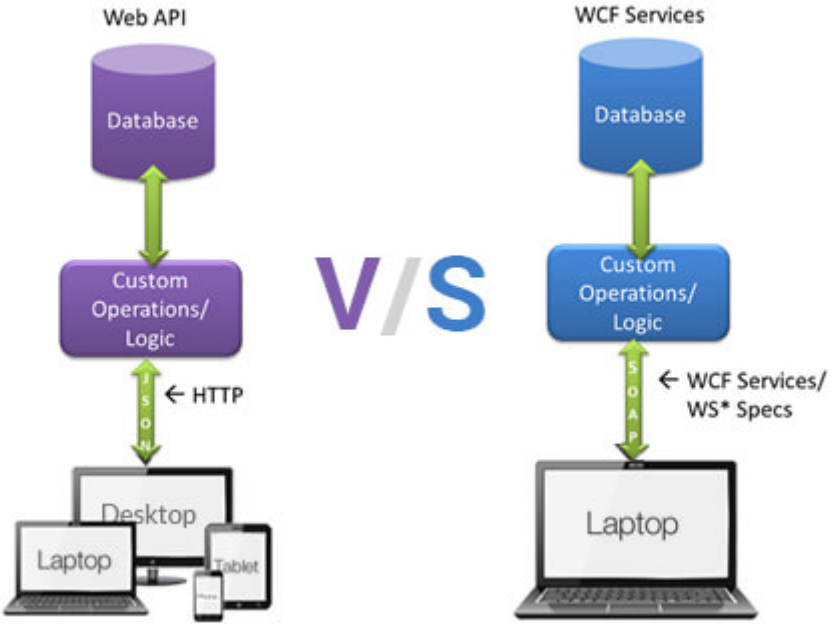
Agregar una plataforma nativa a nuestro proyecto : `ionic cordova platform add android/ios`
Eliminar una plataforma nativa de nuestro proyecto : `ionic cordova platform rm android/ios`
Agregar un plugin nativo: `ionic cordova plugin add nombrePlugin`
Eliminar un plugin nativo: `ionic cordova plugin rm nombrePlugin`
Compilar para android/ios: `ionic cordova build android/ios`
Correr en android/ios en un emulador: `ionic cordova emulate android/ios`
Correr en android/ios en un dispositivo físico: `ionic cordova run android/ios`

Web API

Una API es una interfaz de programación de aplicaciones (del inglés API: Application Programming Interface).
Permiten acceso a características de bajo nivel o propietarias, detallando solamente la forma en que cada rutina debe ser llevada a cabo y la funcionalidad que brinda, sin otorgar información acerca de cómo se lleva a cabo la tarea.

Web Service

El término Web Services describe una forma estandarizada de integrar aplicaciones WEB mediante el uso de XML, SOAP, WSDL y UDDI sobre los protocolos de la Internet. XML es usado para describir los datos, SOAP se ocupa para la transferencia de los datos, WSDL se emplea para describir los servicios disponibles y UDDI se ocupa para conocer cuáles son los servicios disponibles. Uno de los usos principales es permitir la comunicación entre las empresas y entre las empresas y sus clientes. Los Web Services permiten a las organizaciones intercambiar datos sin necesidad de conocer los detalles de sus respectivos Sistemas de Información.



Rest vs SOAP

REST (Representational State Transfer), es un estilo de arquitectura de software dirigido a sistemas hipermedias distribuidos como lo es la web. Es liviano dado que utiliza JSON, pero no es tan seguro como SOAP.
Algunas características de REST:

- Protocolo cliente/servidor sin estado: cada petición HTTP contiene toda la información necesaria para ejecutarla, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla. Aunque esto es así, algunas aplicaciones HTTP incorporan memoria caché. Se configura lo que se conoce como protocolo cliente-caché-servidor sin estado: existe la posibilidad de definir algunas respuestas a peticiones HTTP concretas como cacheables, con el objetivo de que el cliente pueda ejecutar en un futuro la misma respuesta para peticiones idénticas. De todas formas, que exista la posibilidad no significa que sea lo más recomendable.
- Las operaciones más importantes relacionadas con los datos en cualquier sistema REST y la especificación HTTP son cuatro: POST (crear), GET (leer y consultar), PUT (editar) y DELETE (eliminar).
- Los objetos en REST siempre se manipulan a partir de la URI. Es la URI y ningún otro elemento el identificador único de cada recurso de

ese sistema REST. La URI nos facilita acceder a la información para su modificación o borrado.
- Interfaz uniforme: para la transferencia de datos en un sistema REST, este aplica acciones concretas (POST, GET, PUT y DELETE) sobre los recursos, siempre y cuando estén identificados con una URI.

SOAP (Abreviación de Simple Object Access Protocol), es un protocolo de mensajería construido en XML que se usa para codificar información de los requerimientos de los Web Services y para responder los mensajes “antes” de enviarlos por la red. Los mensajes SOAP son independientes de los sistemas operativos y pueden ser transportados por los protocolos que funcionan en la Internet, como ser: SMTP, MIME y HTTP.

Express JS

Express es un framework web, escrito en JavaScript y alojado dentro del entorno de ejecución NodeJS. Es robusto, rápido, flexible y muy simple. Soporta los métodos GET, POST, PUT, DELETE entre otros y posee un método de direccionamiento especial que no se deriva de ningún método HTTP (.all).

Ejemplo de ruteo en REST

Nombre	**Ruta**	**Método HTTP**	**Funcionalidad**
Index	/dispositivo	GET	Muestra todos los dispositivos
Mostrar	/dispositivo/:id	GET	Muestra un dispositivo en específico
Crear	/dispositivo	POST	Inserta un nuevo dispositivo
Modificar	/dispositivo/:id	PUT	Modifica un dispositivo en particular

Borrar	/dispositivo/:id	DELETE	Borra un dispositivo en particular

La definición de ruta tiene la siguiente estructura:

app.MÉTODO(ruta,handler)

Donde:

app es una instancia de *express.*

METHOD es un método de solicitud *HTTP.*

PATH es una vía de acceso a un servidor

HANDLER es la función que se ejecuta cuando se correlaciona la ruta.

Dentro de los handler, uno posible es el denominado Middleware.

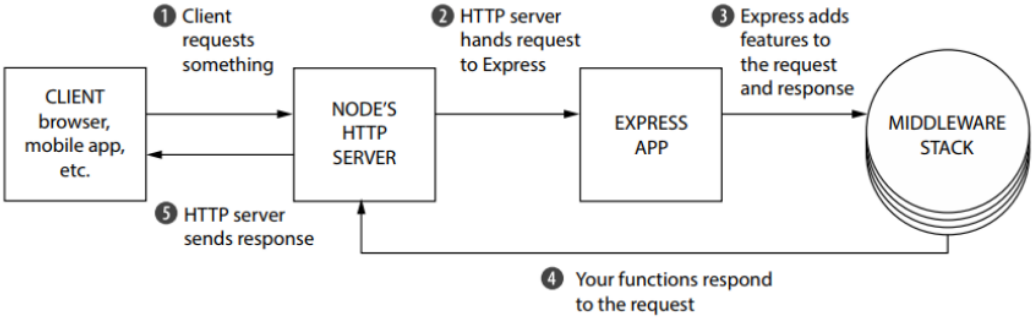
Middleware

Las funciones de middleware son funciones que tienen acceso al objeto de solicitud (req), al objeto de respuesta (res) y a la siguiente función de middleware en el ciclo de solicitud/respuestas de la aplicación. La siguiente función de middleware se denota normalmente con una variable denominada next.

Las funciones de middleware pueden realizar las siguientes tareas:

- Ejecutar cualquier código.
- Realizar cambios en la solicitud y los objetos de respuesta.
- Finalizar el ciclo de solicitud/respuestas.
- Invocar la siguiente función de middleware en la pila.

Si la función de middleware actual no finaliza el ciclo de solicitud/respuestas, debe invocar next() para pasar el control a la siguiente función de middleware. De lo contrario, la solicitud quedará colgada.



Objeto Request

Representa a la request HTTP y algunas de sus properties más útiles son:

- req.body
  - Contiene un par de clave-valor de los datos que se enviaron en el cuerpo de la request. Por defecto, su valor es “undefined” y se va a llenar con valores cuando usemos el “body-parsing” middleware

Ejemplo:

```
var express = require('express')

var app = express()

app.use(express.json()) // para parsear application/json

app.post('/perfil, function (req, res, next) {
  console.log(req.body)
  res.json(req.body)
})
```

- req.ip
  - Contiene la dirección IP de la request
- req.params
  - Esta propiedad es un objeto de propiedades vinculadas a los parámetros de ruta (GET), por ejemplo, si tenemos la ruta

"/usuario/:id", la propiedad "id" la podremos ver usando "req.params.id". Por defecto el valor del objeto es {}

- req.secure
  - Booleano para ver si se estableció una conexión TLS.

Objeto Response

Representa la respuesta HTTP que Express envía cuando recibe una request HTTP.

Algunos métodos que podemos utilizar con Express:

METODO	DESCRIPCIÓN
res.download(path [, filename] [, options] [, fn])	Solicita un archivo para descargarlo.
res.end()	Finaliza el proceso de respuesta sin data.
res.json([body])	Envía una respuesta JSON.
res.jsonp()	Envía una respuesta JSON con soporte JSONP.
res.redirect()	Redirecciona una solicitud.
res.render()	Representa una plantilla de vista.
res.send([body])	Envía una respuesta de varios tipos. En el body puede ir un objeto Buffer, un String, un objeto o un Array por ejemplo.
res.status(code)	Setea el estado de la respuesta HTTP, permite encadenarse con los otros métodos.
res.sendStatus()	Establece el código de estado de la respuesta y envía su representación de serie como el cuerpo de respuesta.

Algunos ejemplos:

```
res.download('/reporteVentas.pdf', 'ventas.pdf');

res.json({ nombre: 'brian' , apellido:'ducca' });

res.redirect('http://www.google.com');

res.send({ usuario: 'bducca' });

res.send([1, 2, 3]);

res.send('test');

res.status(403).end();

res.status(400).send('Bad Request');

res.sendStatus(404) // igual a hacer res.status(404).send('Not Found')
```

Tipos Middleware

- Middleware de nivel de aplicación
- Middleware de nivel de direccionador
- Middleware de manejo de errores
- Middleware incorporado
- Middleware de terceros

Para cargar la función middleware, se llama a app.use(), especificando la función que va a ejecutar.

```
var express = require('express');
var app = express ();
var myLogger = function (req, res, next) {
  console.log('LOGGED');
  next();
};
app.use(myLogger);
```

En el anterior ejemplo, vemos como se aplica el middleware a nivel de aplicación, pero también lo podemos aplicar a nivel de direccionador, la

diferencia pasa en que este último esta enlazado a una instancia de express.Router().

Express.Router

Se utiliza esta clase para crear manejadores de rutas modulares, es un sistema de middleware y direccionamiento completo.

Para utilizarlo, se crea un nuevo archivo de direccionador en el directorio de la aplicación y luego se carga este módulo de direccionador en la aplicación. En el siguiente ejemplo, creo el módulo direccionador “usuarios” que contiene el siguiente código:

```
var express = require('express');
var routerUsuarios = express.Router();

// middleware específico de mi router
router.use(function timeLog(req, res, next) {
  console.log('Tiempo: ', Date.now());
  next();
});
// Defino la ruta para la direccion base
router.get('/', function(req, res) {
  res.send(Usuarios');
});

//Exporto el router

module.exports = routerUsuarios ;
```

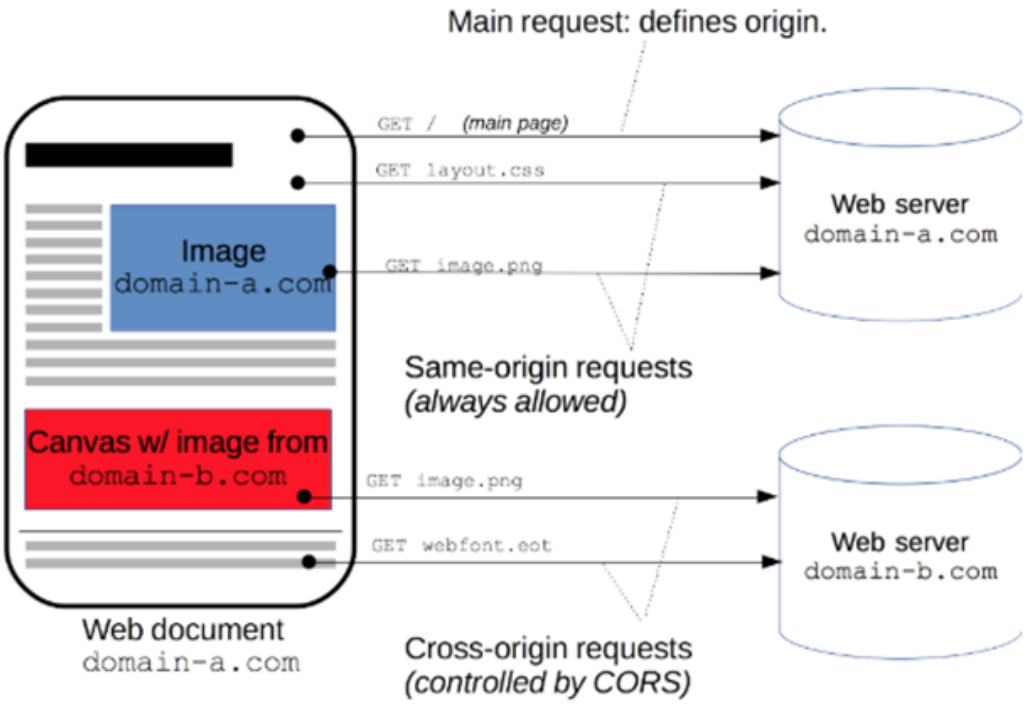
Ahora que ya tengo mi módulo de direccionador “usuarios”, paso a cargarlo en mi archivo index.js de la aplicación.

```
var usuarios = require('./usuarios'); //Dentro del require irá la ruta donde esté ubicado nuestro módulo de direccionador

...

app.use('/usuarios, usuarios);
```

Con esto, la aplicación podrá manejar solicitudes /usuarios y todas las rutas /usuario/* que se hayan definido dentro del módulo de direccionador. A su vez, también invocará a la función middleware timeLog que es específica de la ruta

Cors

Mecanismo que utiliza cabeceras HTTP adicionales para permitir que un user-agent obtenga permiso para acceder a recursos desde un servidor en un origen distinto al que pertenece.



En express vamos a instalar cors ingresando en la carpeta de nuestra API y poniendo: `npm install --save cors`
y luego lo incluimos en nuestro archivo index.js de la siguiente manera:

```
var cors = require('cors');
```

Luego se configura las opciones que queramos aceptar request en el cors de la siguiente manera (El * representa cualquier dirección):

```
var corsOptions = {origin:'*',optionsSucessStatus:200};
```

Por último, lo incluimos como middleware utilizando:

```
app.use(cors(corsOptions));
```

Donde app puede ser una instancia de express o una instancia del router de express.

Observables

Nos permite pasar mensajes entre los denominados “publishers” y “subscribers” en nuestra aplicación. Funciona de manera declarativa, es decir, definimos una función para que publique valores (“publisher”) pero no se va a ejecutar hasta que el consumidor no se subscriba (“subscriber”) y este último recibe notificaciones hasta que la función se complete o hasta que se cancele la subscripción.
Permite manejar 0,1 o más eventos y tiene la posibilidad de ser cancelado cuando queramos.
Microsoft creo una librería conocida como RxJS (Reactive Extensions for JavaScript) utilizada para programar de manera asíncrona, es decir utilizan los observables para hacer operaciones asíncronas. También nos permite utilizar funciones como map(), filter() y concat().
En angular como convención, a los valores que son de tipo Observable se los nombra con el signo $.
Ejemplo:

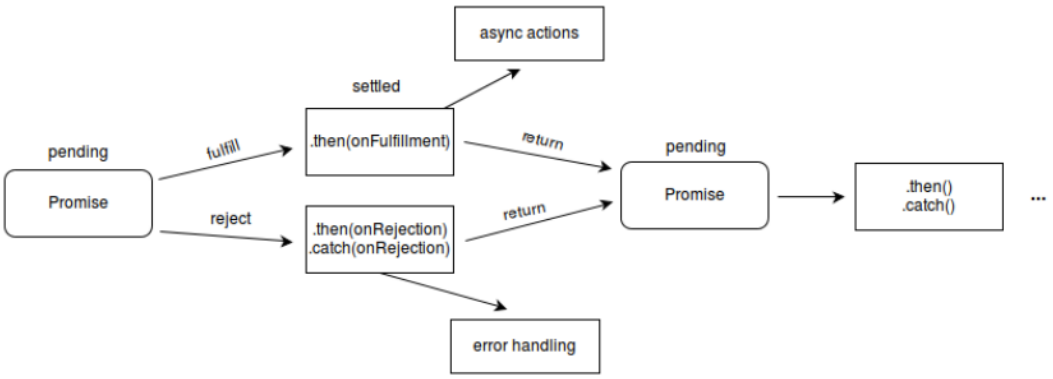
```
export class StopwatchComponent {

 stopwatchValue: number;
 stopwatchValue$: Observable<number>;

 start() {
   this.stopwatchValue$.subscribe(num =>
     this.stopwatchValue = num
   );
 }
}
```

Promesas

Las promesas manejan un solo evento, es decir una operación asíncrona o se completa o falla. Una promesa pendiente puede ser *cumplida* con un valor, o *rechazada* con una razón (error).
Cuando cualquiera de estas dos opciones sucede, los métodos asociados, encolados por el método *then* de la promesa, son llamados.
Una Promesa se encuentra en uno de los siguientes estados:

- pendiente (pending): estado inicial, no cumplida o rechazada.

- cumplida (fulfilled): significa que la operación se completó satisfactoriamente.
- rechazada (rejected): significa que la operación falló.



Como característica principal es que las promesas no se pueden cancelar. Como los métodos .then() y .catch() retornan promesas, éstas pueden ser encadenadas.

- **then**: Se ejecuta cuando la promesa es resuelta satisfactoriamente, también nos permite encolar código, esto lo podremos apreciar en el ejemplo que realizaremos.
- **catch**: Se ejecuta cuando la promesa no puede resolverse de una manera satisfactoria podríamos decir cuando surge un error.

Ejemplo:
Creando una promesa

```
var promise = new Promise((resolve, reject) => {
});
```

Utilizando reject y resolve

```
let error = true;
function doAsyncTask() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (error) {
        reject('error'); // pass values
      } else {
        resolve('done'); // pass values
```

```
      }
    }, 1000);
  });
}

doAsyncTask().then(
  (val) => console.log(val),
  (err) => console.error(err)
);
```

Async-Await

Es una forma alternativa para presentar datos recibidos desde una API, una función async puede contener una expresión await. (ES 7)
El await pausa la ejecución de la función async, espera a la resolución de la promesa, luego continua con la ejecución de la función async y retorna el valor.
En otras palabras, podremos hacer que nuestro código asíncrono se comporte como si fuera síncrono, pero está limitado al scope de las funciones declaradas como “async”.

Ejemplo: Se utiliza la palabra reservada async para especificar que esa función va a ser asincrónica y luego utilizamos el await para esperar el valor de otra función y cuando esta termina y devuelve el valor, continuamos con la ejecución.

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  var result = await resolveAfter2Seconds();
  console.log(result);
}

asyncCall();
```

Si intentamos utilizar “await” en un lugar donde no se declaró como una función “async”, nos dará error, pero en cambio, podemos hacer una función con la palabra reservada async, sin necesidad de utilizar el await. Esto último, aunque no sea muy útil, devolverá una promesa.

NoSQL

Sistemas de gestión de bases de datos que difieren del modelo clásico de SGBDR (Sistema de Gestión de Bases de Datos Relacionales) en aspectos importantes, siendo el más destacado que no usan SQL como lenguaje principal de consultas.
En NoSQL las filas de llamaran documentos. En comparación a las bases de datos relacionales no hay columnas, sino colecciones de documentos. Otra gran diferencia es que la información puede ser 100% redundante no se trata de cumplir con las formas de normalización.

ACID & BASE

En bases de datos se denomina **ACID** a las características de los parámetros que permiten clasificar las transacciones de los sistemas de gestión de bases de datos

- Atomicidad: una transacción se procesa entera o no se procesa ninguna parte. No existen transacciones a medias. Por ejemplo, al realizar una transferencia bancaria se debita de una cuenta y se acredita en la otra, en base de datos estos es un update que se tiene que completar de ambas partes. (Todo o nada)
- Consistencia: Es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Sólo información válida se escribe en la base.
- Isolation (Aislación): Esta propiedad asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información sean independientes entre sí y no generen ningún tipo de error.
- Durable: Esta propiedad asegura que, una vez realizada la operación, esta persistirá y no se podrá deshacer, aunque falle el sistema y que de esta forma los datos sobrevivan de alguna manera.

BASE – Similar a ACID, pero para bases no relacionales.

Muy usado para redes sociales, donde la información va a estar disponible "eventualmente" pero la falta momentánea de consistencia, no es crítica.

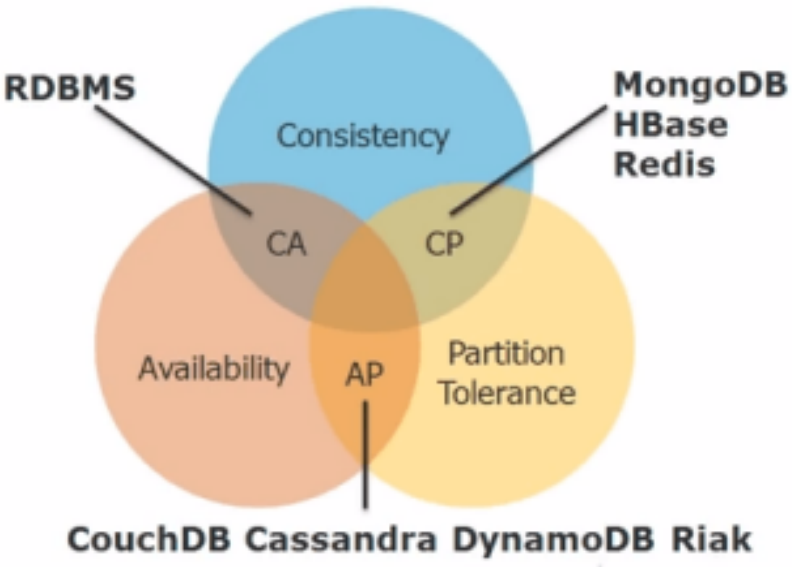
- **B**asically **A**vailable: El sistema estará disponible y garantizará los datos
- **S**oft State: los datos en las diferentes réplicas no tienen que ser mutuamente consistentes en todo momento.
- **E**ventually Consistent – Eventualmente es consistente, se asegura la consistencia solo después de que pase cierto tiempo.

Teorema CAP

Teorema de Brewer: "es imposible para un sistema computacional distribuido ofrecer simultáneamente las siguientes tres garantías". Un sistema no puede asegurar más de dos de estas tres características simultáneamente:

- Consistencia (Consistency)– todos los nodos vean los mismos datos al mismo tiempo
- Disponibilidad (Availability) – garantizar que los clientes encuentran una copia de los datos solicitados, aunque haya nodos caídos en el clúster.
- Tolerancia a la partición (Partition) – el sistema continúa funcionando a pesar de la pérdida de mensajes o fallos parciales del sistema.

Equivalente a: "You can have it good, you can have it fast, you can have it cheap: pick two."



Tipos BD NoSQL

- Base de datos orientadas a documentos
  - Datos almacenados como documentos (HTML, XML y comúnmente JSON)
  - Similar a clave-valor, pero lo almacenado en valor es visible y se puede consultar
  - Ejemplo: CouchDB, MongoDB
- Base de datos Clave-Valor
  - Los datos se almacenan basados en una una clave (key) en un hash-map
  - Datos asociados como Blob (Binary large objects)
  - Alto rendimiento para operaciones CRUD básicas, pero bajo para atender consultas complejas
  - Ejemplo: Cassandra, BigTable
- Base de datos orientada a grafos
  - Datos almacenados como nodos y enlaces (links-relationships)
  - Optimizados para consultas rápidas y búsquedas
  - Ejemplo: Neo4J, OrientDB
- Base de datos orientados a columnas (Tabulares)
  - Datos almacenados en familias de columnas, cada fila tiene una key y una o varias columnas. Las columnas no tienen que ser las mismas en cada fila
  - Ejemplo: Apache HBase

MySql Pool con Express Js

Hemos visto anteriormente que para la conexión a la base de datos utilizabamos el `createConnection` para realizar las consultas y operar; ahora veremos que diferencia existe entre esto y el `createPool`.

Cuando pedimos una conexión al pool, obtendremos una conexión que no está siendo utilizada o una nueva. Si llegamos al límite de conexiones, esperará a que una conexión esté libre antes de continuar.

Entonces, el pool nos da el manejo de manera eficiente de múltiples conexiones a la base de datos y nos permite reutilizar las que no están en uso, mejorando la performance a la hora de ejecutar consultas.

Agregando el middleware de conexión

```
var mysql = require('mysql');
var configMysql = {
    connectionLimit: 10,
    host: 'mysql-server',
    port: 3307,
    user: 'root',
    password: 'userpass',
    database: 'DAM'
}
var pool = mysql.createPool(configMysql);
pool.getConnection((err, connection) => {
    if (err) {
        switch (err.code) {
            case 'PROTOCOL_CONNECTION_LOST':
                console.error('La conexion a la DB se cerró.');
                break;
            case 'ER_CON_COUNT_ERROR':
                console.error('La base de datos tiene muchas
conexiones');
                break;
            case 'ECONNREFUSED':
                console.error('La conexion fue rechazada');
        }
        if (connection) {
            connection.release();
        }
        return;
    }
```

```
});
module.exports = pool;
```

Luego de establecer las credenciales de la base de datos, definimos el número máximo de conexiones que el pool tiene permitido mantener, por ejemplo si establecemos 90 conexiones, pero solamente usamos 10 conexiones simultáneas, sólo se crearán 10 conexiones.
Por eso deberemos establecer un número razonable que vamos a poder manejar en determinado momento. Si nos pasamos de ese número, obtendremos un *ER_CON_COUNT_ERROR*.

Con el middleware de conexión, lo utilizaremos en donde necesitemos de la siguiente manera:

```
var pool = require('../../db');

//Devuelve un array de dispositivos
app.get('/', function(req, res) {
    pool.query('Select * from Dispositivos', function(err,
result, fields) {
        if (err) {
            res.send(err).status(400);
            return;
        }
        res.send(result);
    });
});
```

Pero nos preguntaremos, ¿Qúe pasa con nuestras conexiones, se liberan solas? y la respuesta es sí, una vez utilizadas las conexiones , vuelven a estar en el pool disponibles, pool.query se encarga de establecer la conexión, ejecutar la query y luego hacer el “release” de la misma para que vuelva a estar disponible en el pool. (`pool.getConnection() + connection.query() + connection.release()`)

Mongo DB

Base de datos orientada a documentos, estos documentos
son almacenados en BSON, que es una representación binaria de JSON.

Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que no es necesario seguir un esquema. Los documentos de una misma colección -concepto similar a una tabla de una base de datos relacional -, pueden tener esquemas diferentes.
Imaginemos que tenemos una colección a la que llamamos Personas. Un documento podría almacenarse de la siguiente manera:

```
{
  Nombre: "Pedro",
  Apellidos: "Martínez Campo",
  Edad: 22,
  Aficiones: ["fútbol","tenis","ciclismo"],
  Amigos: [
   {
         Nombre:"María",
         Edad:22
   },
   {
         Nombre:"Luis",
         Edad:28
   }
  ]
}
```

El documento anterior es un clásico documento JSON. Tiene strings, arrays, subdocumentos y números. En la misma colección podríamos guardar un documento como este:

```
{
  Nombre: "Luis",
  Estudios: "Administración y Dirección de Empresas",
  Amigos:12
}
```

Este documento no sigue el mismo esquema que el primero. Tiene menos campos, algún campo nuevo que no existe en el documento anterior e incluso un campo de distinto tipo.
Esto que es algo impensable en una base de datos relacional, es algo totalmente válido en MongoDB.

MongoDB está escrito en C++, aunque las consultas se hacen pasando objetos JSON como parámetro. Es algo bastante lógico, dado que los propios documentos se almacenan en BSON. Por ejemplo:
db.Clientes.find({Nombre:'Pedro'});
La consulta anterior buscará todos los clientes cuyo nombre sea Pedro.
MongoDB viene de serie con una consola desde la que podemos ejecutar los distintos comandos.
Cualquier aplicación que necesite almacenar datos semi-estructurados puede usar MongoDB. Es el caso de las típicas aplicaciones CRUD o de muchos de los desarrollos web actuales. Eso sí, aunque las colecciones de MongoDB no necesitan definir une esquema, es importante que diseñemos nuestra aplicación para seguir uno. Tendremos que pensar si necesitamos normalizar los datos, desnormalizarlos o utilizar una aproximación híbrida. Estas decisiones pueden afectar al rendimiento de nuestra aplicación.
En definitiva, el esquema lo definen las consultas que vayamos a realizar con más frecuencia. MongoDB es especialmente útil en entornos que requieran escalabilidad. Con sus opciones de replicación y sharding, que son muy sencillas de configurar, podemos conseguir un sistema que escale horizontalmente sin demasiados problemas.

MongoDB tiene un framework para realizar consultas de agregación llamado Aggregation Framework. También puede usar Map Reduce. Aun así, estos métodos no llegan a la potencia de un sistema relacional. Si vamos a necesitar explotar informes complejos, deberemos pensar en utilizar otro sistema.

Instalación Local

Simplemente tenemos que bajar los binarios para nuestro sistema operativo. Hay versiones para Windows, Linux y MacOs (https://www.mongodb.com/download-center/community). Una vez bajados podremos arrancar el servicio de MongoDB con un solo comando.

```
mongod --dbpath data
```

Con este comando arrancamos el servicio mongod, que empezará a escuchar peticiones por el puerto 27017. Es importante indicar el parámetro --dbpath, con la ruta dónde se almacenarán los ficheros de nuestra base de datos.

Si ya tenemos el servidor lanzado en nuestra máquina, bastará con lanzar desde la consola el siguiente comando
`mongo localhost`
Desde ese momento entraremos en la consola y podremos realizar consultas. Si escribimos help tendremos un listado con los comandos más comunes y su descripción.

Tenemos también un GUI para MongoDB llamado Robo3T (https://robomongo.org/download), que nos permitirá de manera más visual, realizar nuestras consultas.

Docker Compose (Mongo - Administrador + Node)

El archivo docker-compose.yml está preparado para correr poder levantar todos los servicios necesarios (Antes de ejecutarlo verificar los path de las bases de datos [atributo volume dentro de la configuración de mongo]) para ello deberemos ejecutar este comando parado sobre la carpeta que contiene nuestro docker compose:

```
docker-compose up
```

En el caso de que lo querramos parar, haremos:

```
docker-compose down
```

Requerimientos

- `docker pull abassi/nodejs-server:10.0-dev`
- `docker pull mongo`
- `docker pull mongoexpress-admin`

Contenedores

- Node Js(localhost:8080)
- MongoDB (localhost:27017)
- Mongo-Express (localhost:8081)

```
version: '3'
services:
 mongo:
   image:                                mongo
```

```
  hostname:                          mongo
  container_name:                    mongo
  restart:                           always
  environment:
    MONGO_INITDB_ROOT_USERNAME:      root
    MONGO_INITDB_ROOT_PASSWORD:      rootPass
    MONGO_INITDB_DATABASE:           DAM
  volumes:
  -                                  mongodbData:/data/db
  networks:
  -                                  mongo-net
  ports:
  -                                  "27017:27017"
nodeapp:
  image:
abassi/nodejs-server:10.0-dev
  hostname:                          nodeapp
  container_name:                    nodeapp
  restart:                           always
  environment:
    MONGO_HOSTNAME:                  mongo
    MONGO_PORT:                      27017
    MONGO_USERNAME:                  root
    MONGO_PASSWORD:                  rootPass
  volumes:
    -                                ./src:/home/node/app/src
  networks:
    -                                mongo-net
  depends_on:
    -                                mongo
  ports:
    -                                "8080:8000"
  command:                           nodemon src/index.js

mongoexpress-admin:
```

```
    image:                                  mongo-express
    hostname:                               mongoexpress-admin
    container_name:                         mongoexpress-admin
    restart:                                always
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME:      root
      ME_CONFIG_MONGODB_ADMINPASSWORD:      rootPass
    networks:
      -                                     mongo-net
    depends_on:
      -                                     mongo
    ports:
      -                                     "8081:8081"

  mongo_seed:
    container_name: mongo_import
    image: mongo
    links:
          - mongo
    volumes:
          - ./mongo-seed:/mongo-seed
    command:
          /mongo-seed/import.sh
    networks:
          -                                     mongo-net

networks:
  mongo-net:
    driver:                                 bridge

volumes:
  mongodbData:
    external: false
```

Para este docker compose , necesitaremos tener dentro de nuestro proyecto una carpeta de nombre “src” donde allí ubicaremos nuestra API para que Node la levante.

Por otro lado si necesitamos que cuando se levante el contenedor se importe data, deberemos crear una carpeta de nombre “mongo-seed” y allí crear un archivo de nombre “import.sh” que contendrá el siguiente código:

```
#! /bin/bash

mongoimport --host mongo --port 27017 --authenticationDatabase
admin --username root --password rootPass --db DAM --collection
Usuarios --type json --file /mongo-seed/import.json --jsonArray
```

También deberemos tener en cuenta que necesitaremos tener el archivo import.json que contenga la data que vamos a insertar en la colección de la DB.

MongoDB + Express JS

Para conectar nuestra BD con express debemos instalar el package en nuestra api en express de la siguiente manera:

```
npm install mongodb
```

Luego nos crearemos una carpeta de nombre db y allí pondremos nuestro index.js con el siguiente código:

```
var MongoClient = require('mongodb').MongoClient;
var ObjectID = require('mongodb').ObjectID;
var nombreBD = "DAM";
var mongoConection = { db: "", objectId: "" };


const {
    MONGO_USERNAME,
    MONGO_PASSWORD,
```

```
    MONGO_HOSTNAME,
    MONGO_PORT
} = process.env;

var URL_CONNECTION =
    "mongodb://" + MONGO_USERNAME + ":" + MONGO_PASSWORD +
    "@" +
    MONGO_HOSTNAME + ":" + MONGO_PORT;

var dbSettings =
    " - MONGO_HOSTNAME: " + MONGO_HOSTNAME + "\n" +
    " - MONGO_PORT: " + MONGO_PORT + "\n" +
    " - MONGO_USERNAME: " + MONGO_USERNAME + "\n" +
    " - MONGO_PASSWORD: " + MONGO_PASSWORD + "\n";

console.log(dbSettings);

MongoClient.connect(URL_CONNECTION, { useNewUrlParser: true,
useUnifiedTopology: true }, function(err, client) {
    if (err) {
        console.error(err);
        console.log("Error" + err)
        throw err;
    }
    console.log("Conectado a mongo");
    mongoConection.db = client.db(nombreBD);
    mongoConection.objectId = ObjectID;
```

```
});


module.exports = mongoConection;
```

Terminado de crear nuestro middleware de conexión, lo importaremos cuando sea necesario realizar alguna acción sobre la BD como por ejemplo:

```
var mongo = require('../db');
```

```
app.get('/all', function(req, res) {
    console.log(mongo.db);
    mongo.db.collection("Usuarios").find().toArray(function(err, docs) {
        if (err != null) {
            res.send(err).status(400);
            return;
        }
        console.log(docs);
        res.send(docs);
    });
});
```

Operaciones

SQL	MongoDB (NOSQL)
Select * from Usuarios	db.collection("Usuarios").fin

	d()
Select count(*) from Usuarios	db.collection("Usuarios").countDocuments({},{},callback)
Select * from Usuarios where id>1	db.collection("Usuarios").find({id:{$gt:1}})
Select email,dni from Usuarios where id=1	db.collection("Usuarios").find({id:1}).project({email:1,dni:1})
Select * from Usuarios where nombre like "%B%"	db.collection("Usuarios").find({nombre:/B/ })
Select * from Usuarios where nombre like "B%"	db.collection("Usuarios").find({nombre:/^B/ })
Select * from Usuarios limit 5 order by id DESC	db.collection("Usuarios").find().limit(5).sort({id:-1})
Delete from Usuarios where id=5	db.collection("Usuarios").remove({id:5},callback)
Insert into Usuarios (nombre,apellido) Values (Brian,Ducca)	db.collection("Usuarios").insert({nombre:"Brian",apellido:"Lopez"},callback)
Update Usuarios set nombre="Jose" where id=1	db.collection("Usuarios").update({id:1},{$set:{nombre:"Jose"}},callback)

Deploy

A la hora de realizar la subida de nuestra aplicación a la tienda, vamos a necesitar tener en cuenta los siguientes requerimientos:

iOS

- XCode (ide) para poder compilar nuestra versión en modo productivo.
- Una cuenta de developer de Apple
- Un certificado de provisioning profile en modo productivo vigente.
- Un certificado de app development y distribution válido

Primero tendremos que compilar nuestra app en modo productivo de la siguiente manera:

```
ionic cordova build ios --prod
```

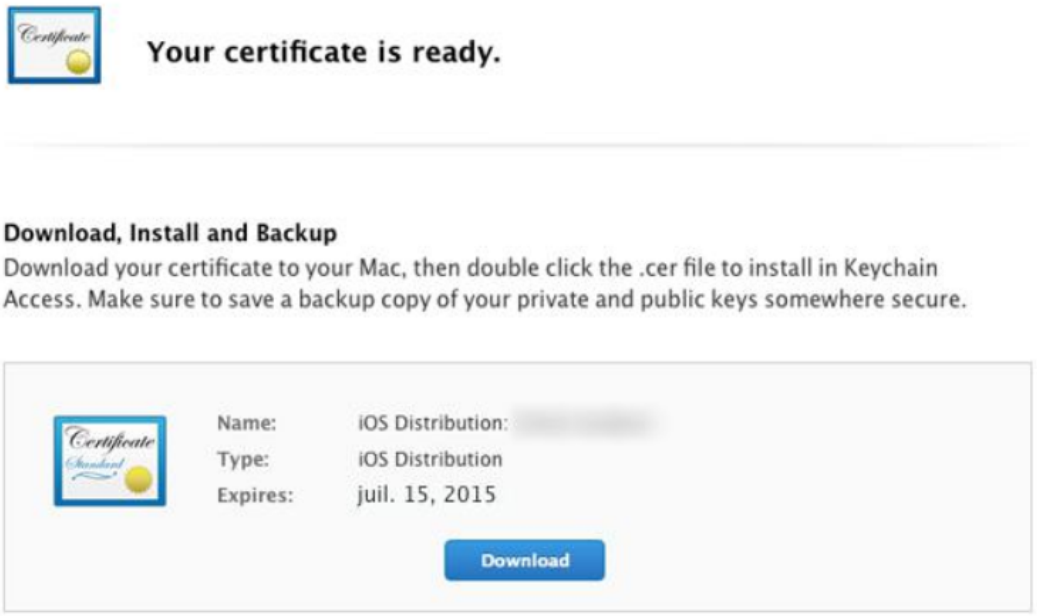
Esto compilará nuestro código que nos permitirá abrirlo como un proyecto de iOS nativo, para ello utilizaremos el Xcode y abriremos el proyecto de extensión `.xcworkspace` ubicado en `./platforms/ios/` de nuestro proyecto.

Una vez que tenemos los certificados correctos, XCode los manejará de manera automática según el tipo de compilación que elijamos. Para subir nuestra app al store, iremos a la opción Archive, del menú Product-> Archive, esto nos abrirá el Xcode Organizer que contiene todos los builds que realizamos, seleccionamos este último y le damos a “Upload to App Store” y con esto damos por finalizada la subida, pero todavía nuestra app no está disponible en la tienda.

Con ello lo que hicimos hasta ahora fue subir nuestra app compilada al store, pero antes de que esté disponible para todos, debe pasar por una revisión por parte de Apple, y recién finalizado y aprobado por ellos, nuestra app estará disponible en la tienda.

Certificados iOS

Como Desarrollador de Apple (Apple Developer), cuando se crea un proyecto necesita ser firmado con un Certificado de Distribución o Distribution Certificate. Este certificado te autentifica como el creador de la app. Es por ese motivo, que el nombre con el que te hayas registrado o el de la empresa (si te has registrado como una organización) aparecerá como "Vendedor" de la app dentro de la App Store.

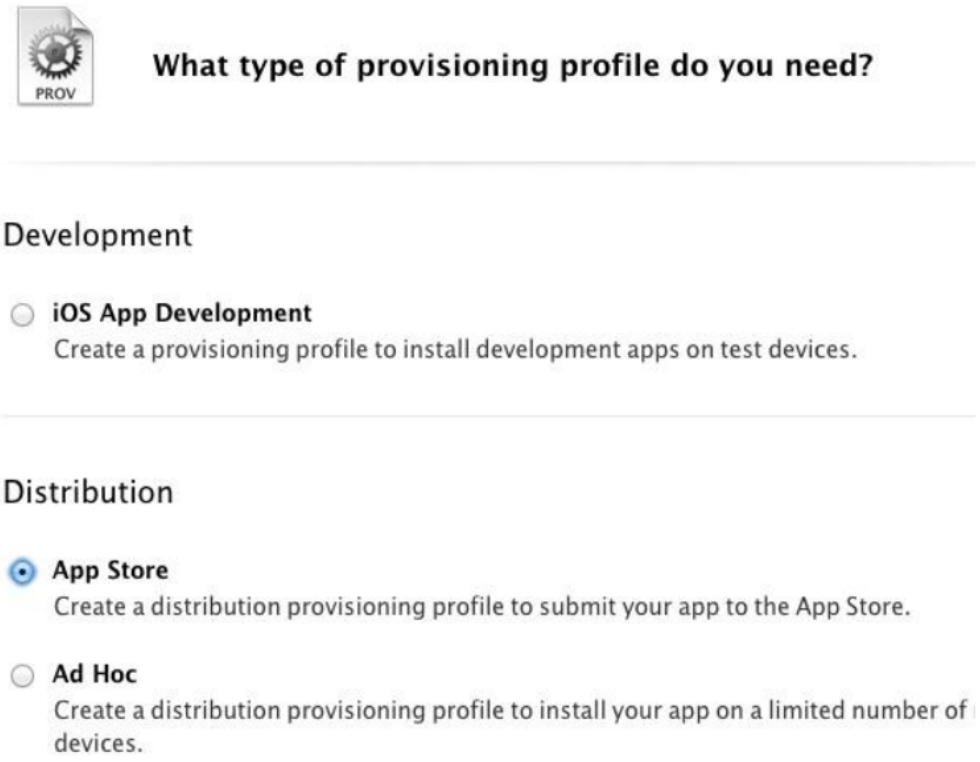


Para generarlo, se necesita subir el archivo Certificate Signing Request (CSR). Para crearlo se necesita recurrir al Keychain Access en una Mac y completado el proceso de subida del CSR, nos descargaremos el archivo ios_distribution.cer que se instalará en nuestro KeyChain.

Con la cuenta de desarrollador se pueden publicar varias apps y utilizar el mismo Certificado de Distribución para ellas. Sin embargo, hay que tener en cuenta que el Certificado de Distribución caduca después de 1 año.

Lo que identifica a cada app como única es lo que se denomina App ID, tiene que existir un App Id por cada App. Si queremos habilitar las notificaciones push dentro de tu app, cuando configuremos el App ID tendremos que habilitar la casilla "Push notifications" antes de terminar y validar el proceso de registro.

La creación del Provisioning Profile es otro paso obligatorio. Es el link entre el desarrollador de la app y el proyecto (un App ID). Se necesita el Provisioning Profile tanto para el Ad Hoc (Distribution - AdHoc) como para el App Store distribution. En este caso, el Provisioning Profiles caduca junto con el Certificado de Distribución.



Por último, si queremos incluir las Notificaciones Push, necesitaremos un último certificado, llamado Push SSL Certificate, este, nos permite la conexión entre el servidor de notificaciones y el servicio de notificaciones push de Apple. Cada app que tengamos que implementarle Notificaciones Push, necesitará de este certificado en concreto. El proceso para generar el certificado de push es el mismo que para generar el certificado de distribución, es decir necesitamos generar un archivo CSR del KeyChain de una Mac, subirlo en la interfaz de Apple developer y luego descargar en este caso el aps_production.cer

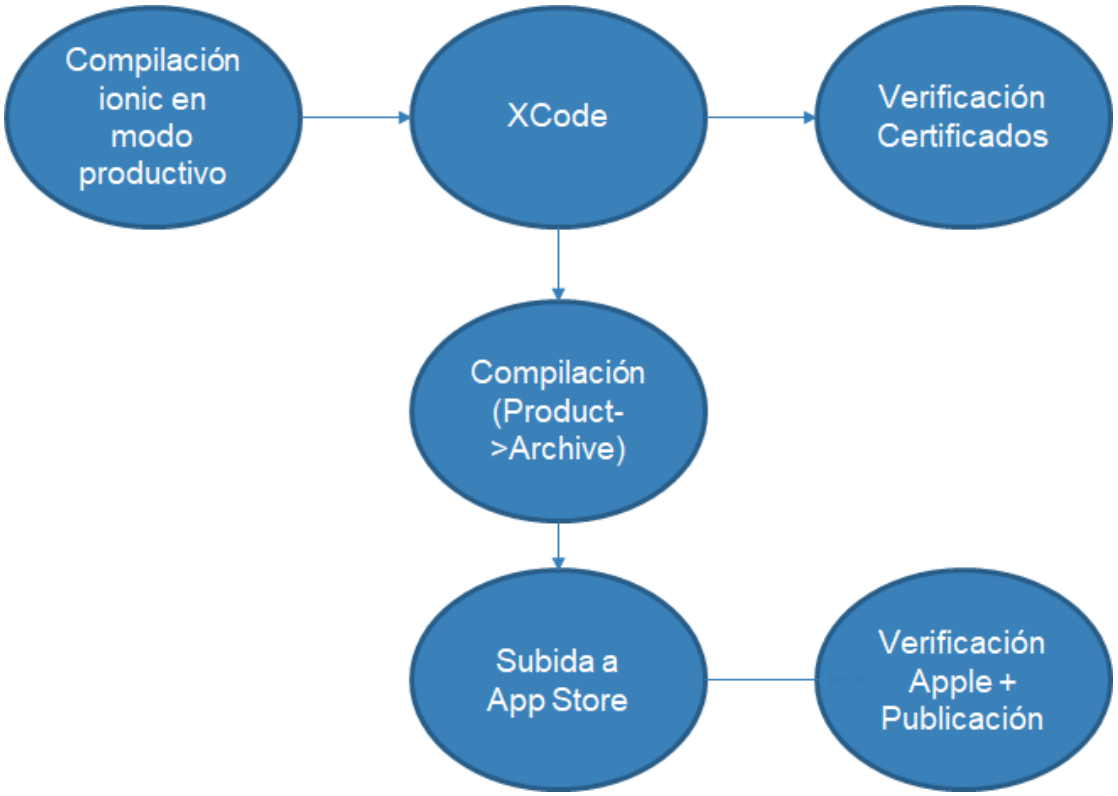
Vencimiento certificados

En el caso de que la cuenta de Apple developer expire, las apps se eliminarán del App Store, pero continuarán funcionando en los dispositivos donde han sido instaladas. Si se renueva la licencia, las apps aparecerán de nuevo en la tienda.

Para los Distribution Certificate se tiene que crear un nuevo certificado para poder compilar la aplicación, realizar una actualización o publicarla de nuevo. Las apps que ya han sido publicadas en la tienda no se verán afectadas.

Si el Provisioning Profiles caducó, hay que generarlo de nuevo para poder actualizar la app conectada a este.

Por último, en el caso de que el Push Certificate caduque, no se podrá enviar notificaciones push desde la app que está conectada a este certificado.



Android

Para android los requerimientos cambian, solo necesitaremos una cuenta de developer de google play que cuesta unos $25USD por única vez.

Generamos nuestro archivo compilado de la siguiente manera:

```
ionic cordova build android --prod --release
```

Esto nos generará el compilado según las configuraciones establecidas en el archivo config.xml y el archivo se encontrará en la ruta:

```
platforms/android/build/outputs/apk
```

Dicha apk está sin firmar, por eso todavía no se puede subir al store, para firmarla debemos utilizar la llave (signing key), en el caso de que no la tengamos, tendremos que generarla de la siguiente manera.
Vamos a utilizar una herramienta que viene con el android SDK, dentro de esta carpeta por línea de comando haremos lo siguiente:

```
keytool -genkey -v -keystore nombreLlave.keystore -alias nombreAlias
-keyalg RSA -keysize 2048 -validity 10000
```

Con este ya tendremos nuestro archivo .keystore que es la llave para firmar nuestra apk. (Con esta llave tendremos que firmar todos los updates de la apk una vez subida al store, si esta llave la perdemos, no podremos actualizar nuestra apk).

Para firmarla, utilizaremos otra herramienta que viene con el android SDK que se llama jarsigner.

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
nombreLlave.keystore nuestroAPK-unsigned.apk nombreAlias
```

Con la apk ya firmada, debemos utilizar una última herramienta llamada zip aling para optimizar nuestro APK, para encontrar la herramienta debemos ir a la carpeta: `/path/to/Android/sdk/build-tools/VERSION/zipalign` y para macOS por ejemplo estaría en: `~/Library/Android/sdk/build-tools/VERSION/zipalign` y corremos este comando:

```
zipalign -v 4 nuestroAPK-unsigned.apk nuestroAPK.apk
```

Con esto terminamos todos los pasos y estamos listos para subir nuestro apk al Google Play Store, para ello debemos ingresar aquí https://play.google.com/apps/publish/signup/ , y una vez que ingresamos

vamos al botón de Crear Aplicación y completamos todos los datos requeridos.

Como nota, cabe aclarar que cada vez que subamos una nueva versión de la apk, deberemos modificar el config.xml cambiando el número de versión en el atributo “versión” del .XML.

Links útiles

Deploy android paso a paso con capacitor :
https://www.joshmorony.com/deploying-capacitor-applications-to-android-development-distribution/

https://developer.android.com/studio/publish/

Consideraciones generales :
https://developer.android.com/distribute/best-practices/launch/launch-checklist

---

Deploy iOS paso a paso con capacitor:
https://www.joshmorony.com/deploying-capacitor-applications-to-ios-development-distribution/

https://developer.apple.com/app-store/submitting/

Creación de splash screen y iconos de nuestra app:
https://capacitorjs.com/docs/guides/splash-screens-and-icons