Angular
Guia de atajos

{dev/talles}

Commandos Básicos

Instalar AngularCLI: Ejecutar esto como administrador

```
npm install -g @angular/cli
```

Iniciar un proyecto nuevo: dos formas

```
npm init @angular myApp
ng new my-app
```

Angular CLI: Command line interface

Estos son unos de los comandos comunes para usarlos en con Angular CLI

Comando para obtener ayuda de cualquier subrogando

```
ng --help
ng generate --help
ng build --help
```

Comandos para generar componentes en Angular

Comando	Descripción
ng g **component** [name] ng g **c** [name]	Genera un nuevo componente en el **[name]**, que puede ser también **[path/name]**
ng g **directive** [name] ng g **d** [name]	Crea una nueva directiva
ng g **guard** [name] ng g **g** [name]	Crea un nuevo guard para protección de rutas
ng g **interceptor** [name]	Crea un interceptor para observables.
ng g **module** [name] ng g **m** [name]	Crea un módulo que en sí, es un agrupador de diferentes componentes.
ng g **pipe** [name] ng g **p** [name]	Crea un pipe que en sí, es un transformador visual de la data.
ng g **service** [name] ng g **s** [name]	Crea un servicio, que en sí, permite compartir información entre la aplicación.

Banderas útiles a los comandos de generación

Bandera	Descripción
ng g c **--help**	Muestra la ayuda completa de todos los comandos disponibles.
ng g c [name] **--dry-run**	Ejecuta el comando sin modificar ningún archivo. Útil si no se está seguro de lo que el comando hará
ng g c [name] **--flat**	Crea los nuevos archivos en el nivel especificado sin crear un nuevo directorio.
ng g c [name] **--inline-style** ng g c [name] **-s**	Incluye los estilos en el mismo archivo controlador (.ts)

Template Syntax

Enlaza la propiedad **"value"**, al valor de lo que contenga **"firstName"** Puede ser cualquier expresión de JavaScript

```
<input [value]="firstName">
```

Lo mismo se puede enlazar **(bind)** con muchos otros atributos de los elementos HTML y componentes personalizados

```
Añade el "role" igual al valor de myAriaRole
<div [attr.role]="myAriaRole">

Añade la clase extra-sparkle si es verdadera
la expresión
<div [class.extra-sparkle]="isDelightful">

Incrementa el ancho en pixeles igual al
valor de mySize
<div [style.width.px]="mySize">

Añade el listener click y dispara
readRainbow
<button (click)="readRainbow($event)">
```

Enlaza el valor de la expresión al valor que vemos dentro del párrafo

```
<p> Hello {{ponyName}} </p>
```

Configura el enlace de datos **bidireccional**.

```
<my-cmp [(title)]="name">
```

Que sería el equivalente a:

```
<my-cmp
  [title]="name"
  (titleChange)="name=$event">
```

Uso de pipes:

```
<p>
  Nombre: {{ 'fernando' | uppercase }}
</p>
```

Angular
Guia de atajos

{dev/talles}

Operador de navegación segura

```
<p>
  Employer: {{employer?.companyName}}
</p>
```

Referencias locales en el html

```
<input #search type="text" />

<video #movieplayer …></video>
<button (click)="movieplayer.play()">
  Play
</button>
```

Directivas incluídas

Las siguientes directivas vienen incluidas dentro del módulo **"CommonModule"** de **@angular/common**

Ver la otra forma de @if, @for, @switch más adelante en este PDF.

ngIf:** Remueve o crea una parte del DOM basado en la expresión **"showSection"

```
<section *ngIf="showSection">
```

***ngFor:** Convierte el <li> en un template, y lo usa para duplicarlo basado en la cantidad de elementos dentro de la lista*

```
<li *ngFor="let item of list">
```

ngSwitch: Condicionalmente cambia el contenido del ** ** por el template que cumpla la condición.

```
<div [ngSwitch]="conditionExpression">
  <ng-template [ngSwitchCase]="case1Exp">
    …
  </ng-template>
  <ng-template
ngSwitchCase="case2LiteralString">
    …
  </ng-template>
  <ng-template ngSwitchDefault>
```

ngClass: Enlaza clases de css basado en un objeto o expresión.

```
<div [ngClass]=
   "{'active': isActive,
     'disabled': isDisabled}">
```

ngStyle: Permite asignar estilos a los elementos html utilizando CSS.

```
<div [ngStyle]="{'property': 'value'}">
<div [ngStyle]="dynamicStyles()">
```

FormsModule de **@angular/forms**

```
<input [(ngModel)]="userName">
```

Decoradores de clase para componentes

@Input: Define una propiedad que puede ser enviada desde el padre hacia el componente hijo. **@angular/core**

```
@Input() myProperty;
```

Ejemplo:

```
<my-cmp [myProperty]="someExpression">
```

@Output: Define una salida del componente que el componente padre puede suscribirse para escuchar.

```
@Output() myEvent = new EventEmitter();
```

Ejemplo:

```
<my-cmp (myEvent)="someExpression">
```

@HostBinding: Enlaza el elemento anfitrión (host) a la propiedad de la clase: **@angular/core**

```
@HostBinding('class.valid') isValid;
```

@HostListener: Se suscribe al evento click del anfitrión (host), opcionalmente se puede recibir el evento. **@angular/core**

```
@HostListener('click', ['$event'])
  onClick(e) {…}
```

@ViewChild y @ViewChildren: Enlaza el resultado final de la vista del componente basado en el predicado a la propiedad de la clase (no es válido para directivas) **@angular/core**

```
@ViewChild(myPredicate) myChildComponent;
@ViewChildren(myPredicate) myChildComponent;
```

Ciclo de vida - **Lifecycle Hooks**

Estos son los pasos de ejecución cuando un componente o directiva entra en pantalla.

Hook / Class Method	Descripción
constructor	Se llama antes de cualquier ciclo de vida.
ngOnChanges	Antes de cualquier cambio a una propiedad.
ngOnInit	Justo después del constructor.

Angular
Guia de atajos

{dev/talles}

Hook / Class Method	Descripción
ngDoCheck	Se llama cada vez que una propiedad del componente o directiva es revisada.
ngAfterContentInit	Después de ngOnInit, cuando el componente es inicializado.
ngAfterContentChecked	Se llama después de cada revisión del componente o directiva.
ngAfterViewInit	Después del ngAfterContentInit
ngAfterViewChecked	Llamado después de cada revisión de las vistas del componente o directiva.
ngOnDestroy	Se llama justo antes de que el componente o directiva va a ser destruida.

Configuración de rutas y Router

Este es un ejemplo de rutas comunes

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'path/:routeParam', component: … },
  { path: 'staticPath', component: … },
  { path: '**', component: … },
  { path: 'oldPath', redirectTo: '/staticPath' },
  { path: …, component: …,
    data: { message: 'Custom' }
  }
]);

const routing = RouterModule.forRoot(routes);
```

En el HTML:

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

LazyLoad: Permite de manera perezosa, cargar un módulo. Esto significa cargarlo bajo demanda (cuando un usuario lo solicita) y luego queda en memoria.

```
import { Routes } from '@angular/router';

const routes: Routes = [
  {
    path: 'items',
    loadChildren: () => import('./items/
items.module').then(m => m.ItemsModule)
  }
];
```

RouterLink: Diferentes anchor tags soportados para navegación

```
<a routerLink="/path">
<a [routerLink]="[ '/path', routeParam ]">

<a [routerLink]="[ '/path', { matrixParam: 'value' } ]">
<a [routerLink]="[ '/path' ]" [queryParams]="{ page: 1 }">
<a [routerLink]="[ '/path' ]" fragment="anchor">
```

RouterLinkActive: Mostrar anchor tag con una clase si nos encontramos en la respectiva ruta:

```
<a [routerLink]="[ '/path' ]"
   routerLinkActive="active">
```

Protección de rutas

CanActivateFn: Una interfaz que nos permite definir una función para validar si una ruta se puede activar.

```
import {
  CanActivateFn,
  ActivatedRouteSnapshot, RouterStateSnapshot
  } from "@angular/router"

  function canActivateGuard: CanActivateFn =
    (
      route: ActivatedRouteSnapshot,
      state: RouterStateSnapshot
    ) => { … }

  # Definición de la ruta
  { path: …, canActivate: [canActivateGuard] }
```

CanDeactivateFn: Interface para definir una función que permite indicarle a Angular si el usuario puede salir de una ruta, útil si hay cambios pendientes de guardar por parte del client.

```
import {
  CanDeactivateFn, ActivatedRouteSnapshot,
RouterStateSnapshot } from "@angular/router"

  function canDeactivateGuard: CanDeactivateFn<T> =
    (
      component: T,
      route: ActivatedRouteSnapshot,
      state: RouterStateSnapshot
    ) => { … }

  # Definición de la ruta
  { path: …, canDeactivate: [canDeactivateGuard] }
```

El mismo concepto se puede aplicara a las siguientes funciones:

Función	Descripción
CanActivateChildFn	El router determina si la ruta hija se puede activar
ResolveFn	El router determina si puede o no mostrar una ruta.
CanLoadFn	El router determina si puede cargar mediante lazy load un módulo

La forma común de protección de rutas en Angular es utilizar clases inyectables que implementan los métodos mencionados: CanActivate, CanLoad, CanDeactivate.

Angular
Guia de atajos

{dev/talles}

Guards con clases

Debe de implementar la interfaz de “CanActivate” para que Angular lo considere un pipe para proteger una ruta a la hora de activarla.

```
class UserToken {}
class Permissions {
  canActivate(): boolean {
    return true;
  }
}

@Injectable()
class CanActivateTeam implements CanActivate {
  constructor(private permissions: Permissions,
private currentUser: UserToken) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean|UrlTree>|Promise<boolean|
UrlTree>|boolean|UrlTree {
    return
this.permissions.canActivate(this.currentUser,
route.params.id);
  }
}
```

Y se utiliza de la siguiente manera en la definición de las rutas.

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: 'team/:id',
        component: TeamComponent,
        # AQUI!!!  <—
        canActivate: [CanActivateTeam]
      }
    ])
  ],
  providers: [CanActivateTeam, UserToken, Permissions]
})
class AppModule {}
```

Pero también se puede definir en línea con la siguiente función.

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: 'team/:id',
        component: TeamComponent,
        canActivate: [
        (route: ActivatedRouteSnapshot, state:
RouterStateSnapshot) => true
]
      }
    ])
  ],
})
class AppModule {}
```

La clave de los Guards, es que deben de retornar un valor **Boolean** si quieren dejar pasar la petición. También puede ser una promesa que resuelva un boolean o un observable que emita un boolean.

Pipes

Este es un listado de los pipes propios de Angular

Pipe	Descripción
DatePipe	Realizar formateos a una fecha
UpperCasePipe	Capitaliza todo el texto
LowerCasePipe	Coloca en minúscula todo el texto
CurrencyPipe	Formatea el número a un formato de moneda.
DecimalPipe	Transforma un número a un string con formato especificado
PercentPipe	Transforma el número a un porcentaje string, formateado basado en las reglas locales.
AsyncPipe	Espera el resultado de una tarea asíncrona (Promise u Observable) e imprime la resolución o emisión
I18nPluralPipe	Es una mapa de valores para ayudar con la localización de palabras
I18nSelectPipe	Similar al anterior, pero para singulares
JsonPipe	Convierte un valor en una representación con formato JSON
KeyValuePipe	Transforma un objeto o mapa, en un arreglo de pares de valores
SlicePipe	Crea un nuevo arreglo o string que contiene el subset (slice o corte) de los elementos
TitleCasePipe	Capitaliza cada palabra del string que este separado por espacios

RxJS - Official Docs

RxJS, es una librería incluida en Angular para poder trabajar con observables y en sí, la programación reactiva.

Hay muchos operadores y funciones para generar observables que se usan en Angular, considere estudiar RxJS para poder reducir la cantidad de código de los observables y hacer el trabajo más simple.

Pueden ver mis cursos de Angular y RxJs Aquí

Angular
Guia de atajos

{dev/talles}

Signals (Angular 17+)

Son un sistema que granularmente sigue cómo y dónde cambia un valor a lo largo de la aplicación. Esto optimiza la velocidad de renderización.

Las señales permiten con precisión quirúrgica, saber en dónde necesita Angular cambiar el valor de una variable, calcular un nuevo valor, disparar un efecto secundario, etc.

Writable Signals

Son señales que pueden cambiarse manualmente, se pueden usar como simples variables reactivas:

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


const count = signal(0);

console.log( 'Count es: ' + count()
```

Se usan los paréntesis para obtener su valor.

O como propiedades de clase **(preferido)** (se pueden usar en servicios, componentes, directivas, funciones, etc)

```
export class AppComponent {

  public count = signal(0);

}
```

Para actualizar el valor de una señal, hay 3 formas

Set

Para establecer un valor a la señal sin importar el valor que tuviera anteriormente. **Si el valor es el mismo** que antes, los efectos y señales computadas no se vuelven a disparar.

```
public count = signal(0);

changeSignal() {
 this.count.set(10);
}
```

Update

Si se desea cambiar el valor de la señal dependiendo del valor que tenga anteriormente, se puede utilizar el método **update**.

```
public count = signal(0);

changeSignal() {
  this.count.update( value => value +
1);
```

Mutar estado

Si se desea cambiar el valor de la señal dependiendo del valor que tenga anteriormente pero mutando el estado, se puede usar el método update de esta manera.

```
const todos = signal(
  [
    {title: 'Learn signals', done: false}
  ]
);

todos.update( value => {
  value[0].done = true;

  return value;
});
```

Computed Signals

Básicamente son señales de sólo lectura, pero es un valor que puede cambiar únicamente si una señal usada para crearlo cambia.

```
const count = signal<number>(0);
const doubleCount = computed(() => count() * 2);
```

En este caso, doubleCount sólo cambiará, si count cambia. Esto sucede automáticamente.

Tanto las señales computadas como los efectos (ver siguiente tema), pueden tener tantas señales internas como sea necesario

```
const showCount = signal(false);
const count = signal(0);

const conditionalCount = computed(() =>
{
  if (showCount()) {
    return `The count is ${count()}.`;
  } else {
    return 'Nothing to see here!';
  }
});
```

Referencias:
Signals - Advanced Topics

Effects

Un efecto es una operación que se ejecuta al menos una vez (similar al ngOnInit), pero si usa alguna señal, y esa señal cambia, el efecto se vuelve a ejecutar.

```
effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

Angular
Guia de atajos

{dev/talles}

El efecto retorna un **EffectRef**, que a su vez tiene un método **destroy,** en el caso que quieras destruir el efecto manualmente.
Los efectos a menos de especificar lo contrario, se limpian a si mismos cuando no son necesarios.

Usos comunes de efectos

- Mostrar logs, cuando la data cambia, ya sea para analytics o depuración.
- Mantener la data en sincronía (window.localStorage)
- Añadir comportamiento al DOM cuando la sintaxis de template no pueda hacerlo.
- Realizar renderizado personalizado a un canvas, gráficos o librerías de terceros
- Disparar efectos secundarios necesarios basados en señales.

Cuando **no** usar efectos

Evitar efectos para propagar cambios de estado, esto puede resultar en el error **ExpressionChangedAfterItHasBeenChecked**, actualizaciones circulares infinitas o cambios no deseados.

Debido a esto, las señales por defecto marcan warnings o errores si quieres cambiar el valor de una señal en un efecto. (Se puede pero no se debe), para este tipo de casos, usar computed.

Effect Injection Context

Registar un efecto requiere un “injection context” básicamente acceso a la función inject (usada para inyectar dependencias desde Angular 14+)

La forma más fácil es registrarlos en el constructor, pero puedes colocarlos como propiedades de clase también.

En constructor:

```
@Component({...})
export class EffectiveCounterCmp {
  readonly count = signal(0);
  constructor() {
    // Registrar el efecto
    effect(() => {
      console.log(`The count is: ${this.count()}`);
    });
  }
}
```

Como propiedades de clase:

```
@Component({...})
export class EffectiveCounterCmp {
  readonly count = signal(0);

  private loggingEffect = effect(() => {
    console.log(`The count is: ${this.count()}`);
  });
}
```

Si se necesita crear un efecto fuera de esos lugares, se puede mediante opciones y un servicio de inyección **(Injector),** más información aquí

Effect cleanup functions

Los efectos pueden correr tareas largas o bien ser ejecutados por largos periodos de tiempo, a veces el efecto debe o será destruido y/o regenerado, si se necesita un procedimiento especial que deben de ejecutar cuando se destruyen, puedes mandar una función de limpieza.

```
effect((onCleanup) => {
  const user = currentUser();

  const timer = setTimeout(() => {
    console.log(`1 second ago, the user became ${user}`);
  }, 1000);

  onCleanup(() => {
    clearTimeout(timer);
  });
});
```

Referencia:
https://angular.io/guide/signals

Nuevo flujo de control

@if

Similar al *ngIf ahora tenemos un @if mucho más poderoso

```
// user-controls.component.ts
@Component({
  standalone: true,
  selector: 'user-controls',
  template: `
    @if (isAdmin) {
      <button>Erase database</button>
    }
  `,
})
export class UserControls {
  isAdmin = true;
}
```

Angular
Guia de atajos

{dev/talles}

@else

```
// user-controls.component.ts
@Component({
  standalone: true,
  selector: 'user-controls',
  template: `
    @if (isAdmin) {
      <button>Erase database</button>
    } @else {
      <p>You are not authorized.</p>
    }
  `,
})
export class UserControls {
  isAdmin = true;
}
```

@for

Similar al *ngFor, ahora tenemos una nueva forma de realizar ciclos en nuestra data, que también es combinable con el @if (que no se podía hacer antes)

```
<!-- ingredient-list.component.html -->
<ul>
  @for (ingredient of ingredientList;
track ingredient.name) {
    <li>{{ ingredient.quantity }} -
{{ ingredient.name }}</li>
  }
</ul>
```

La propiedad **"track"** le permite a angular poder seguir el elemento en caso de que el listado cambie.

Variables disponibles dentro del @for

Pipe	Descripción
$count	Numero total de elementos del listado
$index	Posición index del elemento actual
$first	Valor boolean que determina si es el primer elemento del listado
$last	Valor boolean que determina si es el último elemento del listado
$even	Determina si el índice actual es par
$odd	Determina si el índice es impar

```
@for (item of items; track item.id;
       let idx = $index, e = $even) {
  Item #{{ idx }}: {{ item.name }}
}
```

@empty

Si el arreglo no tiene elementos, puedes usarlo para mostrar un elemento.

```
@for (item of items; track
item.name) {
  <li> {{ item.name }} </li>
} @empty {
  <li> There are no items. </li>
}
```

@switch

Similar al @if, tenemos una forma inspirada en el Switch de JavaScript.

```
@switch (condition) {
  @case (caseA) {
    Case A.
  }
  @case (caseB) {
    Case B.
  }
  @default {
    Default case.
  }
}
```

Vistas aplazables - Deferrable Views

Es un mecanismo que tenemos para poder controlar de forma granular, la manera cómo queremos que nuestros componentes se carguen en pantalla y atrasar su ejecución.

@defer y @placeholder

Permite definir que el bloque de código será cargado de forma perezosa y no renderizado de forma inicial, permitiéndonos especificar el momento en el cual cargarlo.

```
@defer {
  <large-component />
} @placeholder (minimum 500ms) {
  <p>Placeholder content</p>
}
```

Angular
Guia de atajos

{dev/talles}

@loading

Es un bloque opcional de código que permite declarar el contenido a mostrar durante la carga de cualquier dependencia aplazable.

```
@defer {
  <large-component />
} @loading (after 100ms; minimum 1s) {
  <img alt="loading..." src="loading.gif" />
}
```

@error

Permite mostrar un contenido en caso de que la carga diferida falle.

```
@defer {
  <calendar-cmp />
} @error {
  <p>Failed to load the calendar</p>
}
```

Defer Triggers

Hay dos opciones que tenemos para controlar cuando un componente será cargado de forma aplazable, y es el **"on"** y **"when".** También se pueden mezclar.

```
@defer (on interaction; on timer(5s)) {
  <calendar-cmp />
} @placeholder {
  <img src="placeholder.png" />
}
```

Este código significa que el bloque será disparado cuando el usuario interactúe con el placeholder o después de 5 segundos.

When <condition>

Similar a una condición aplicable a un @if, puedes añadir la condición de carga

```
@defer (when cond) {
  <calendar-cmp />
}
```

Recuerda que se pueden mezclar.

```
@defer (on viewport; when cond) {
  <calendar-cmp />
} @placeholder {
  <img src="placeholder.png" />
}
```

Aquí se cargará cuando cualquiera de las dos condiciones se cumpla.

Posibles opciones "on"

Estas son las posibles opciones en el defer con "on"

Pipe	Descripción
on idle	Se dispara cuando el navegador llega a un estado inactivo "idle"
on viewport	Se dispara cuando el bloque entra al punto de vista del usuario. Por defecto se puede conectar el placeholder y otro elemento
on interaction	Se dispara cuando el usuario interactúa con un elemento específico mediante un click o keydown
on hover	Se dispara cuando el mouse pasa sobre el elemento o la referencia.
on immediate	Se dispara tan pronto el cliente termina de renderizar la pantalla.
on timer	Se dispara después de cierta duración de tiempo en **MS** milliseconds.

On Interaction - Formas

```
@defer (on interaction) {
  <calendar-cmp />
} @placeholder {
  <div>Calendar placeholder</div>
}
```

Otra interesante

```
<button type="button"
   #greeting>Hello!</button>

@defer (on interaction(greeting)) {
  <calendar-cmp />
} @placeholder {
  <div>Calendar placeholder</div>
}
```

Prefetching

@defer permite precargar el código componente (no lo ejecuta, carga el código) y así tenerlo listo tan pronto sea el momento de llamarlo.

```
@defer (on interaction; prefetch on idle) {
  <calendar-cmp />
} @placeholder {
  <img src="placeholder.png" />
}
```

Angular
Guia de atajos

{dev/talles}

Glosario de Términos

Angular y AngularJS
Angular es el framework de Google, es el sucesor de la librería AngularJS, la cual ya está obsoleta.

Angular y AngularJS son dos cosas distintas.

Inyección de dependencias - DI
También conocido como DI, es un patrón de diseño en la cual una clase requiere una o más dependencias para poder inicializarse.

Usualmente esas dependencias ya están inicializadas en otro lugar y el componente utiliza esa misma instancia.

Observable
Es un objeto en el cual a lo largo del tiempo, puede estar emitiendo diferentes valores.

Usualmente cuando hablamos de **"suscribimos a los observables"**, significa estar escuchando las emisiones que ese objeto estará emitiendo a lo largo de su vida.

Components
Usualmente es un archivo que está compuesto por una clase controladora escrita en TypeScript, un template HTML que puede estar separado y un archivo de CSS opcional. También se puede juntar con los archivos de pruebas.

Directives - Directivas
Hay varios tipos de directivas, pero en sí, nos permiten cambiar el comportamiento de elementos HTML o componentes personalizados.

Adicionalmente Angular cuenta con varias directivas propias como ngSwitch, *ngFor y ngIf.

Guard
Es un tipo de middleware, middleman, o guardia de acceso. Es un objeto que está pendiente si alguien puede entrar, cargar o salir de una ruta.

Interceptor
Es un objeto que brinda un mecanismo para interceptar y/o mutar las solicitudes y respuestas HTTP.

RXJS
Es una librería pre-cargada en Angular para trabajar con programación reactiva. Básicamente todo lo relacionado a Observables, es de rxjs.

Pipe
Objeto que permite transformar de manera visual la información que mostraremos en pantalla sin mutar la data real.

Module
Son contenedores para realizar la cohesion de código que tiene funcionamiento en común. Este funcionamiento común puede ser un flujo de trabajo, conjunto de capacidades o código dedicado a un dominio de la aplicación.

camelCase
Usarlo para variables, métodos, propiedades.

UpperCamelCase
Usarlo para nombres de clases e interfaces.

dash-case / kebab-case
Usarlo para describir nombres de archivos y selectores de componentes.

UPPER_UNDERSCORE_CASE
UPPER_SNAKE_CASE: Usarlo tradicionalmente para constantes de compilación.

Change Detection
Es el mecanismo en el cual Angular sincroniza el estado del UI con el estado de la aplicación.

Decorator
Es una simple función que modifica una clase o propiedad. Básicamente añade una funcionalidad o comportamiento específico.

Polyfill
Usualmente es una forma en la cual podemos expandir el soporte a plataformas que no cuentan con lo que necesitamos para correr un proceso.