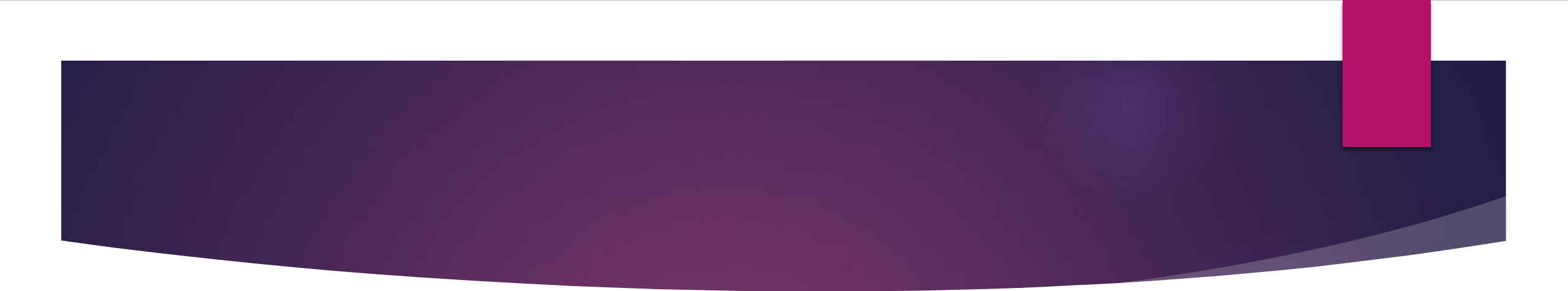
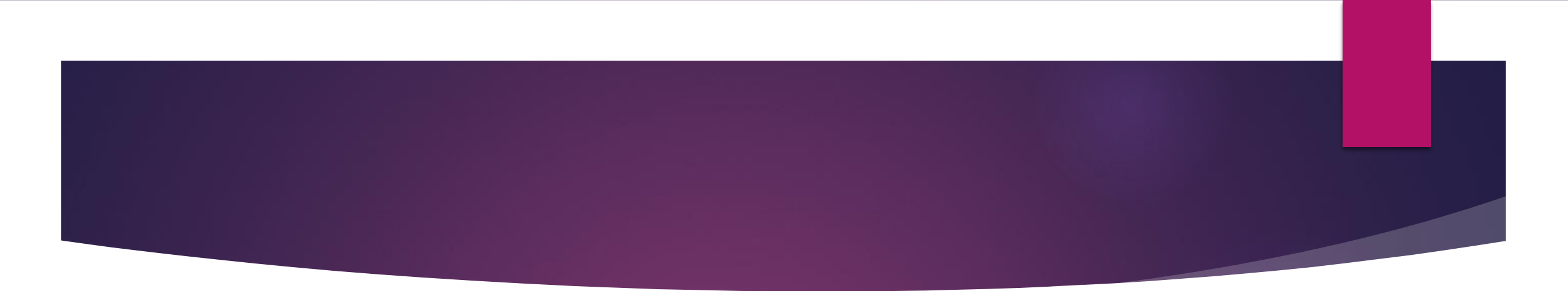


Construyendo un editor WYSIWYG para escritorio usando Electron

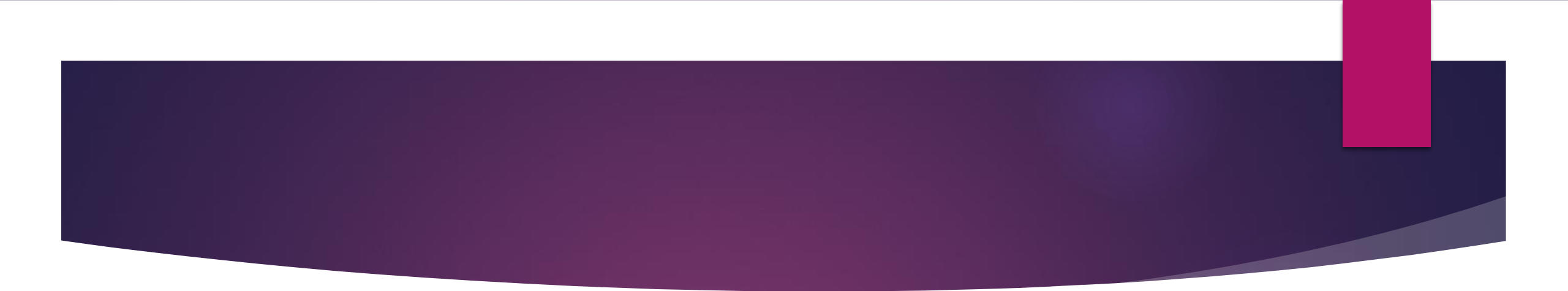
DESARROLLO DE APLICACIONES WEB II

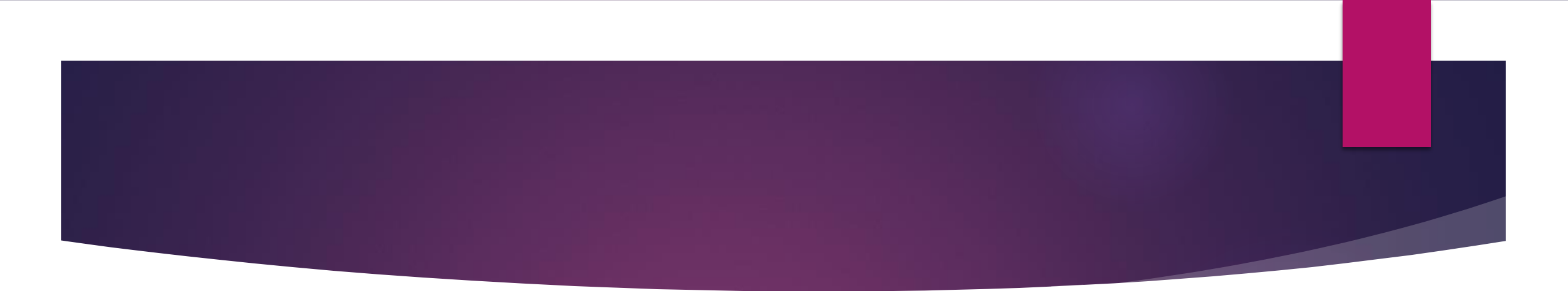
- 
- ▶ Las aplicaciones web se crean tradicionalmente con HTML, CSS y JavaScript. Su uso también se ha extendido ampliamente al desarrollo de servidores con Node.js. En los últimos años han surgido varias herramientas y frameworks que utilizan HTML, CSS y JavaScript para crear aplicaciones para escritorio y dispositivos móviles. En este capítulo, abordaremos cómo crear aplicaciones de escritorio usando Angular y Electron.
 - ▶ Electron es un framework de JavaScript que se utiliza para crear aplicaciones de escritorio nativas con tecnologías web. Si lo combinamos con Angular, podemos crear aplicaciones web rápidas y de alto rendimiento.

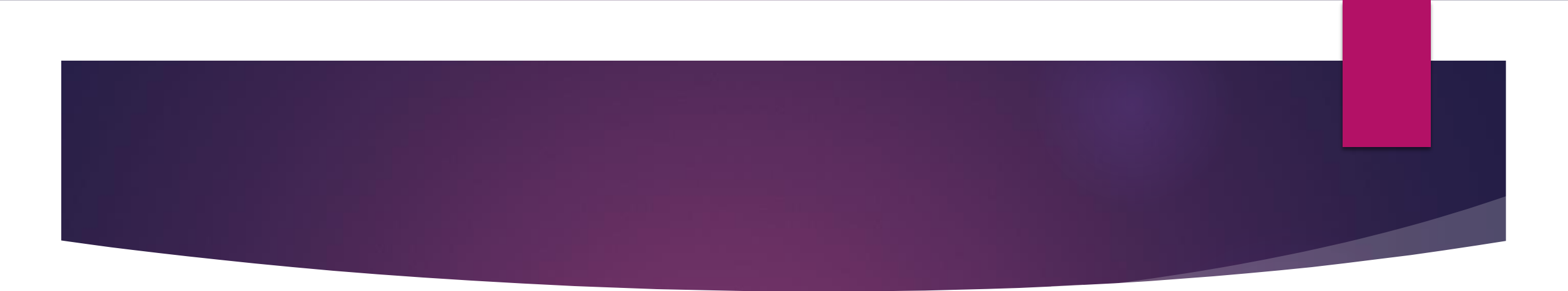
- 
- ▶ En este capítulo, vamos a construir un editor WYSIWYG como una aplicación de escritorio y cubriremos los siguientes temas:
 - ▶ Agregar una biblioteca de editor WYSIWYG para Angular
 - ▶ Integración de Electron en el espacio de trabajo
 - ▶ Comunicación entre Angular y Electron
 - ▶ Empaquetar una aplicación de escritorio

Teoría y contexto esenciales

- ▶ Electron es un framework multiplataforma que se utiliza para crear aplicaciones de escritorio para Windows, Linux y Mac. Muchas aplicaciones populares se crean con Electron, como Visual Studio Code, Skype y Slack.
- ▶ Electron se basa en Node.js y Chromium. Los desarrolladores web pueden aprovechar sus habilidades existentes en HTML, CSS y JavaScript para crear aplicaciones de escritorio sin tener que aprender un nuevo lenguaje como C++ o C#.

- 
- ▶ Las aplicaciones Electron tienen muchas similitudes con las aplicaciones PWA. Considere la posibilidad de crear una aplicación Electron para escenarios como la manipulación avanzada del sistema de archivos o cuando necesite una apariencia más nativa para su aplicación.
 - ▶ Otro caso de uso es cuando está creando una herramienta complementaria para su producto de escritorio principal y desea distribuirlos juntos.

- 
- ▶ Una aplicación en Electron consta de dos procesos:
 - ▶ Main: Interactúa con los recursos locales nativos usando la API de Node.js
 - ▶ Renderer: Es el responsable de manejar la Interfaz de usuario de la aplicación.
 - ▶ Una aplicación Electron puede tener un solo proceso main que puede comunicarse con uno o más procesos renderer. Cada proceso renderer opera de forma completamente aislada de los otros.

- 
- ▶ Electron proporciona las interfaces `ipcMain` e `ipcRenderer`, que podemos usar para interactuar con estos procesos.
 - ▶ La interacción se logra mediante la comunicación entre procesos (IPC), un mecanismo que intercambia mensajes de forma segura y asíncrona a través de un canal común a través de una API basada en promesas.

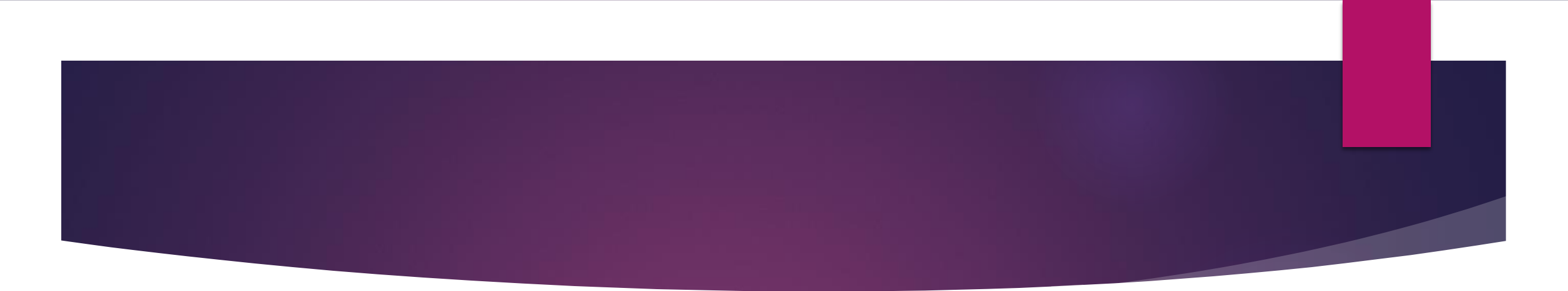
Proyecto

- ▶ Construiremos un editor WYSIWYG de escritorio que mantiene su contenido local en el sistema de archivos. Inicialmente, lo construiremos como una aplicación Angular usando ngx-wig, una popular biblioteca WYSIWYG.
- ▶ Luego lo convertiremos en una aplicación de escritorio usando Electron y aprenderemos cómo sincronizar contenido entre Angular y Electron.
- ▶ También veremos cómo conservar el contenido del editor en el sistema de archivos.
- ▶ Finalmente, empaquetaremos nuestra aplicación como un solo archivo ejecutable que se puede ejecutar en un entorno de escritorio.

Agregar una biblioteca de editor WYSIWYG para Angular

- ▶ Primero, iniciaremos nuestro proyecto creando un editor WYSIWYG como una aplicación angular independiente.
- ▶ Use la CLI de Angular para crear una nueva aplicación Angular desde cero:

```
ng new my-editor --defaults
```

- 
- ▶ Pasamos las siguientes opciones al nuevo comando ng:
 - ▶ my-Editor: Define el nombre de la aplicación.
 - ▶ --defaults: define CSS como el formato de estilos preferido de la aplicación y desactiva el enrutamiento porque nuestra aplicación consistirá en un solo componente que albergará al editor

- ▶ Un editor WYSIWYG es un editor de texto rico, como Microsoft Word. Podríamos crear uno desde cero con angular, pero sería un proceso que consume mucho tiempo, y solo reinventaríamos la rueda.
- ▶ El ecosistema angular contiene una amplia variedad de bibliotecas para usar para este propósito. Uno de ellos es la Biblioteca NGX-WIG, que no tiene dependencias externas, ¡solo angular! Agregemos la biblioteca a nuestra aplicación y aprender a usarlo:
 - ▶ Use el cliente npm para instalar ngx-wig desde el registro de paquetes npm:

```
my-editor> npm install ngx-wig
```


- Abra el archivo app.module.ts y agregue ngxwigmodule a la matriz de importaciones del decorador @ngmodule:

```
src > app > TS app.module.ts > AppModule
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3
4  import { AppComponent } from './app.component';
5  import { NgxWigModule } from 'ngx-wig';
6  @NgModule({
7    declarations: [
8      AppComponent
9    ],
10   imports: [
11     BrowserModule,
12     NgxWigModule
13   ],
14   providers: []
```


- ▶ Crea un nuevo componente angular que albergará nuestro editor WYSIWYG:

```
tor> ng generate component editor
```

- ▶ Abra el archivo de plantilla del componente recién generado, editorponent.html, y reemplace su contenido con el siguiente fragmento HTML:

```
src > app > editor > <> editor.component.html >  ngx-wig  
1 <ngx-wig placeholder="Enter your content"></ngx-wig>
```

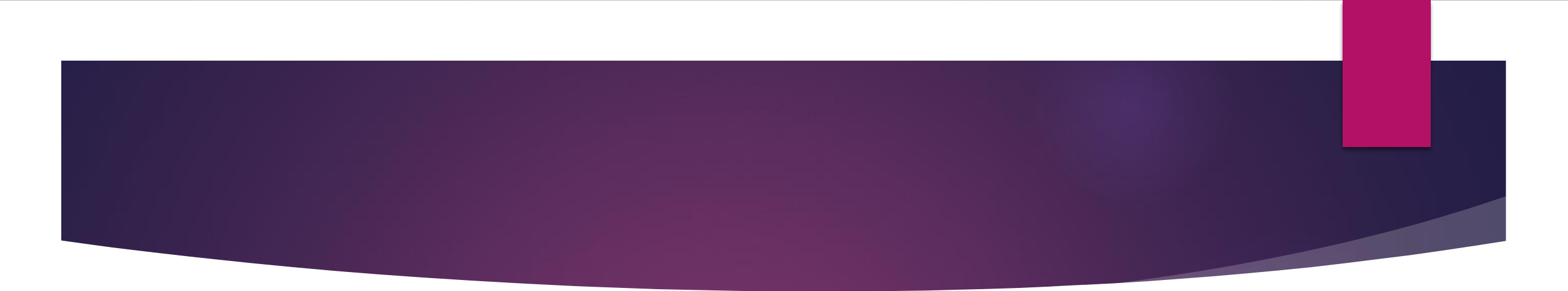
- Abra el archivo `app.component.html` y reemplace su contenido con el selector del editor de aplicaciones:

```
src > app > <> app.component.html >  app-editor  
1 | <app-editor></app-editor>
```

- Abra el archivo `styles.css`, que contiene estilos globales para la aplicación angular, y agregue los siguientes estilos para que el editor se acople y ocupe la página completa:

src > # styles.css > ...

```
1  html, body {
2      margin: 0;
3      width: 100%;
4      height: 100%;
5  }
6  .ng-wig, .nw-editor-container, .nw-editor {
7      display: flex !important;
8      flex-direction: column;
9      height: 100% !important;
10     overflow: hidden;
11 }
```

- 
- ▶ Abra el archivo HTML principal de la aplicación angular, index.html, y retire la etiqueta base del elemento principal.
 - ▶ La etiqueta base se usa desde el navegador hasta los scripts de referencia y los archivos CSS con una URL relativa.
 - ▶ Dejar la etiqueta base hará que nuestra aplicación de escritorio falle porque cargará todos los activos necesarios directamente desde el sistema de archivos local. Aprenderemos más en la sección de integración con electrón.

rc > <> index.html > ...

1 <!doctype html>


2 <html lang="en">

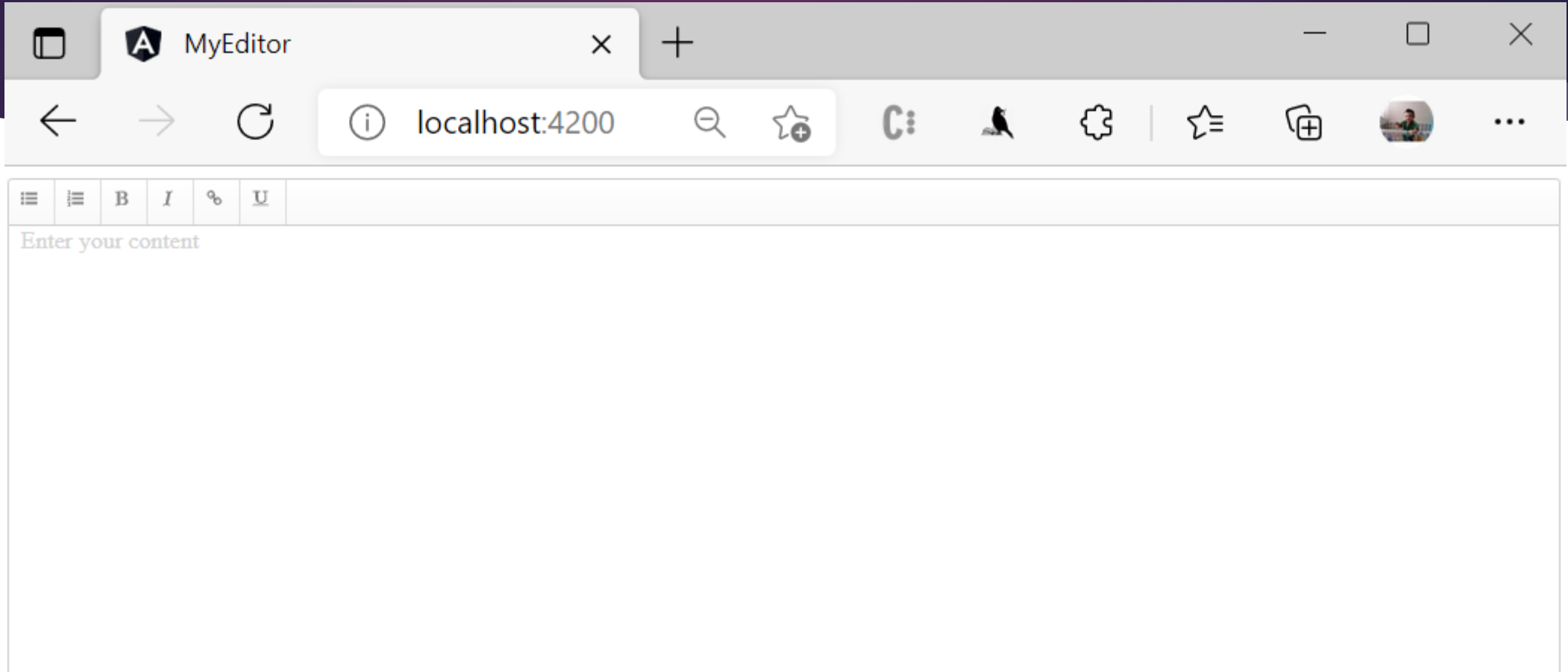
3 <head>

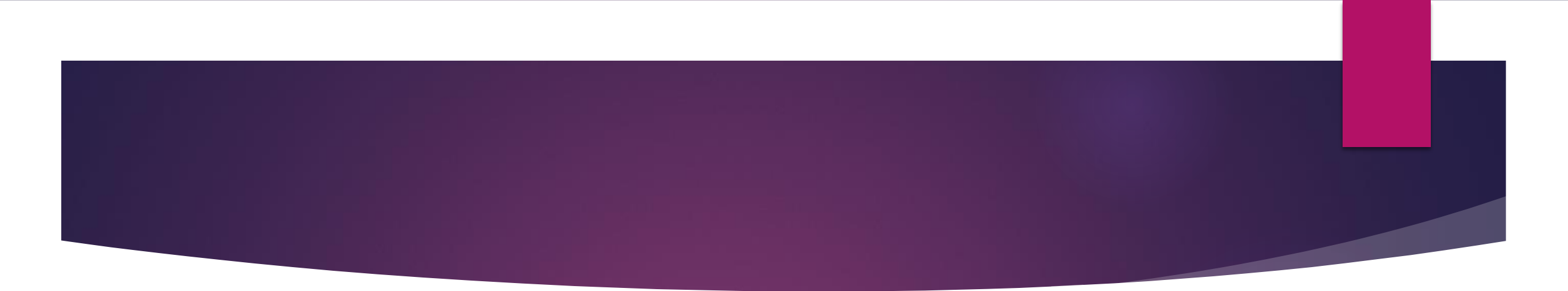
4 <meta charset="utf-8">

5 <title>MyEditor</title>

6 <base href="/">





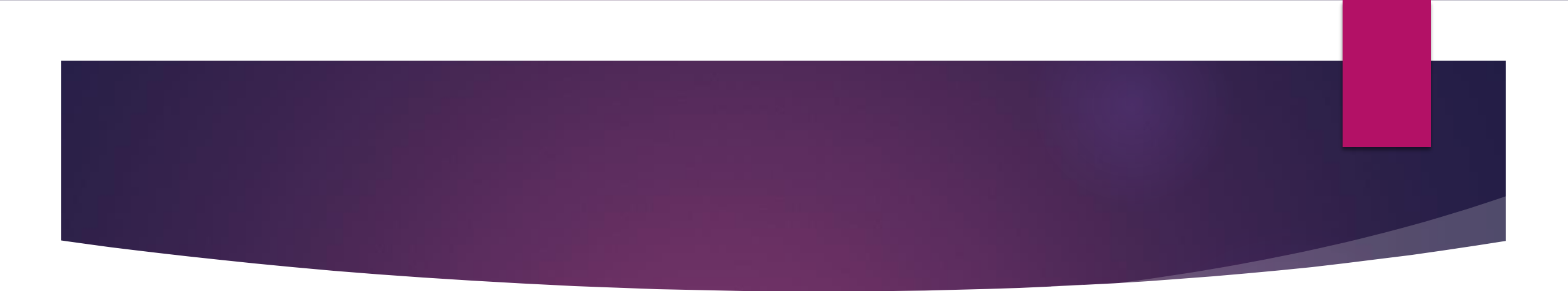
- 
- ▶ Nuestra aplicación consta de lo siguiente:
 - ▶ Una barra de herramientas con botones que nos permite aplicar diferentes estilos al contenido del editor
 - ▶ Un área de texto que se utiliza como contenedor principal para agregar contenido al editor
 - ▶ Ahora hemos creado una aplicación web utilizando angular que presenta un editor WYSIWYG en pleno funcionamiento. En la siguiente sección, aprenderemos cómo convertirlo en un app de escritorio usando Electron

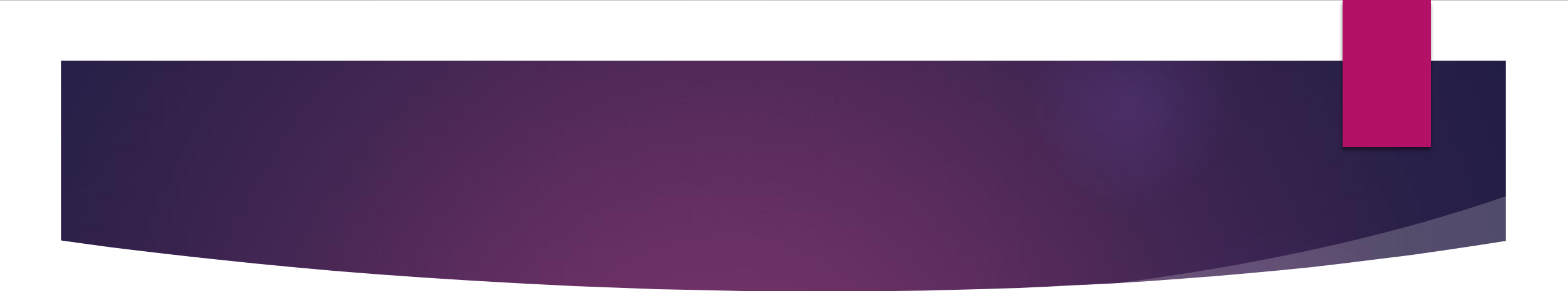
Integración de Electron en el espacio de trabajo.

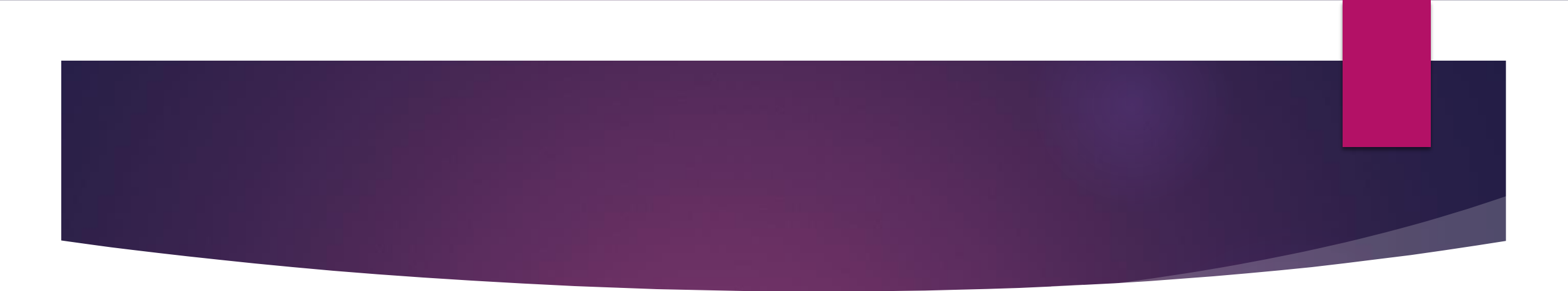
- ▶ Electron es un paquete npm que podemos instalar usando el siguiente comando:

```
or> npm install -D electron
```

- ▶ El comando anterior instalará la última versión del paquete de electrón en el espacio de trabajo de CLI angular. También agregará una entrada respectiva en la sección DevDependencies del archivo Package.json de nuestro proyecto.

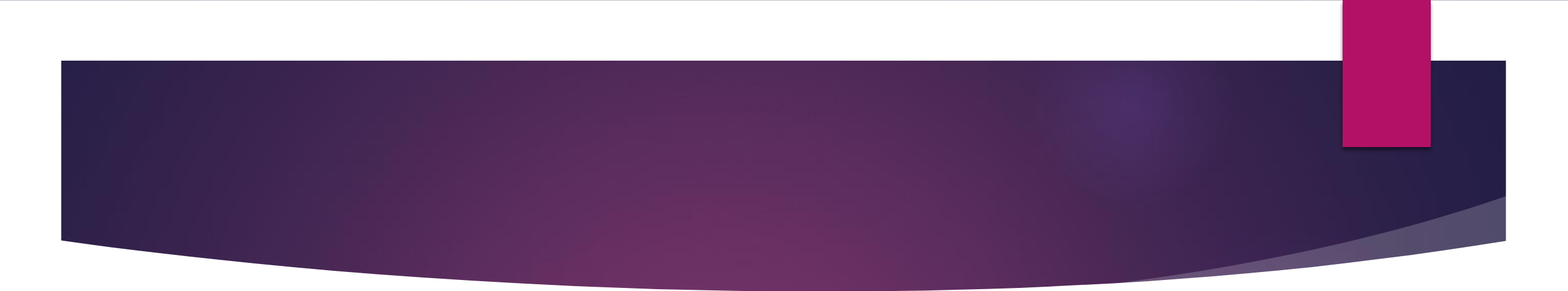
- 
- ▶ Electrón se agrega a la sección DevDependencies del archivo package.json porque es una dependencia de desarrollo de nuestra aplicación.
 - ▶ Se utiliza solo para preparar y construir nuestra aplicación como escritorio y no durante el tiempo de ejecución.

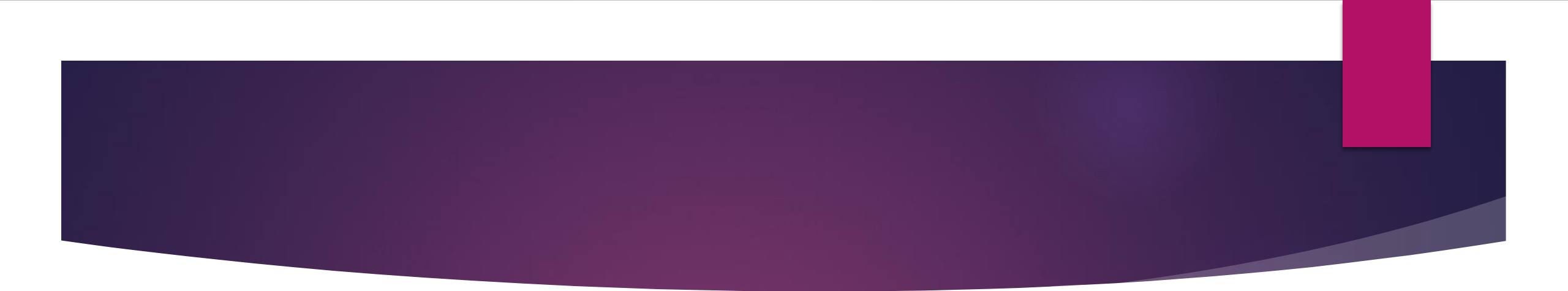
- 
- ▶ Las aplicaciones Electron se ejecutan en el tiempo de ejecución del Node.js y usan el navegador de chromium para propósitos de representación. Una aplicación Node.js tiene al menos un archivo JavaScript, generalmente llamado index.js o main.js, que es el punto de entrada principal de la aplicación. Dado que estamos usando Angular y TypeScript como nuestra pila de desarrollo, comenzaremos creando un archivo de TypeScript respectivo que finalmente se compilará a JavaScript:
 - ▶ 1. Cree una carpeta llamada electron dentro de la carpeta src del espacio de trabajo angular de CLI. La carpeta electron contendrá cualquier código fuente que esté relacionado con electron.

- 
- ▶ Podemos pensar en nuestra aplicación como dos plataformas diferentes.
 - ▶ La plataforma web es la aplicación angular, que reside en la carpeta `src/app`.
 - ▶ La plataforma de escritorio es la aplicación electron, reside en la carpeta `src/electron`.
 - ▶ Este enfoque tiene muchos beneficios, incluidos que impone la separación de las preocupaciones en nuestra aplicación y permite que cada uno se desarrolle independientemente de la otra. A partir de ahora, nos referiremos a ellos como la aplicación angular y la aplicación electron.

- Cree un archivo main.ts dentro de la carpeta electron con el siguiente contenido:

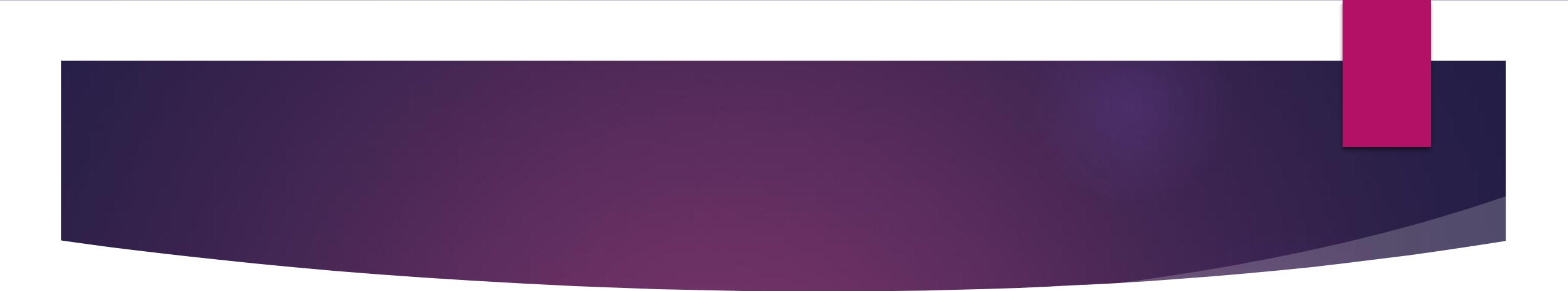
```
src > electron > TS main.ts > ...
1  import { app, BrowserWindow } from 'electron';
2  function createWindow () {
3      const mainWindow = new BrowserWindow({
4          width: 800,
5          height: 600
6      });
7      mainWindow.loadFile('index.html');
8  }
9  app.whenReady().then(() => {
10     createWindow();
11 });
```


- 
- ▶ Primero importamos el `BrowserWindow` y los artefactos de la aplicación desde el paquete `electron` npm.
 - ▶ La clase `BrowserWindow` se utiliza para crear una ventana de escritorio para nuestra aplicación. Definimos las dimensiones de la ventana, pasando un objeto de opciones en su constructor que establece los valores de ancho y alto de la ventana.
 - ▶ Luego llamamos al método `loadFile`, pasando como parámetro el archivo HTML que queremos cargar dentro de la ventana.

- 
- ▶ El archivo `index.html` que pasamos en el método `loadFile` es el archivo HTML principal de la aplicación Angular.
 - ▶ Se cargará utilizando `file:///protocol`, por lo que eliminamos la etiqueta `base`.
 - ▶ El objeto `app` es el objeto global de nuestra aplicación de escritorio, al igual que el objeto `window` en una página web. Expone una promesa `whenReady` que, cuando se resuelve, significa que podemos ejecutar cualquier lógica de inicialización para nuestra aplicación, incluida la creación de la ventana.

- Cree un archivo `tsconfig.json` dentro de la carpeta `electron` y agregue el siguiente contenido:

```
src > electron > TS tsconfig.json > ...  
1  {  
2      "extends": "../../tsconfig.json",  
3  }  
4      "compilerOptions": {  
5          "importHelpers": false  
6      },  
7      "include": [  
8          "**/*.ts"  
9      ]  
9  }
```

- 
- ▶ El archivo `main.ts` debe compilarse en JavaScript porque los navegadores actualmente no entienden TypeScript. El proceso de compilación se llama transpilación y requiere un archivo de configuración de TypeScript. El archivo de configuración contiene opciones que controlan el transpilador TypeScript, que es responsable del proceso de transpilación.
 - ▶ El archivo de configuración de TypeScript anterior define la ruta de los archivos de código fuente de Electron utilizando la propiedad `include` y establece la propiedad `importHelpers` en `false`.
 - ▶ Si habilitamos el indicador `importHelpers`, incluirá los helpers de la biblioteca `tslib` en nuestra aplicación, lo que dará como resultado un tamaño de paquete más grande.

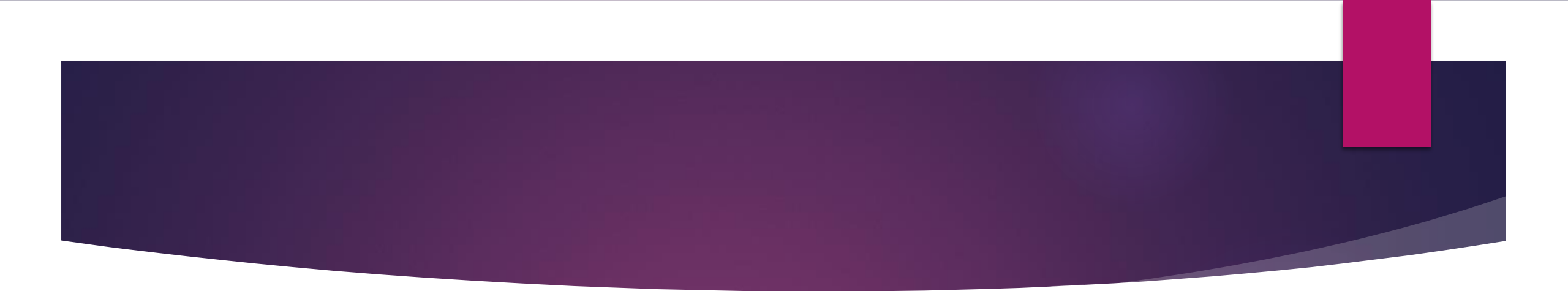
- ▶ Ejecute el siguiente comando para instalar la CLI del paquete web:

```
tor> npm install -D webpack-cli
```

- ▶ La CLI de webpack se utiliza para invocar webpack, un popular paquete de módulos, desde la línea de comandos. Usaremos webpack para construir y empaquetar nuestra aplicación Electron.
- ▶ Instale el paquete ts-loader npm con el siguiente comando npm:

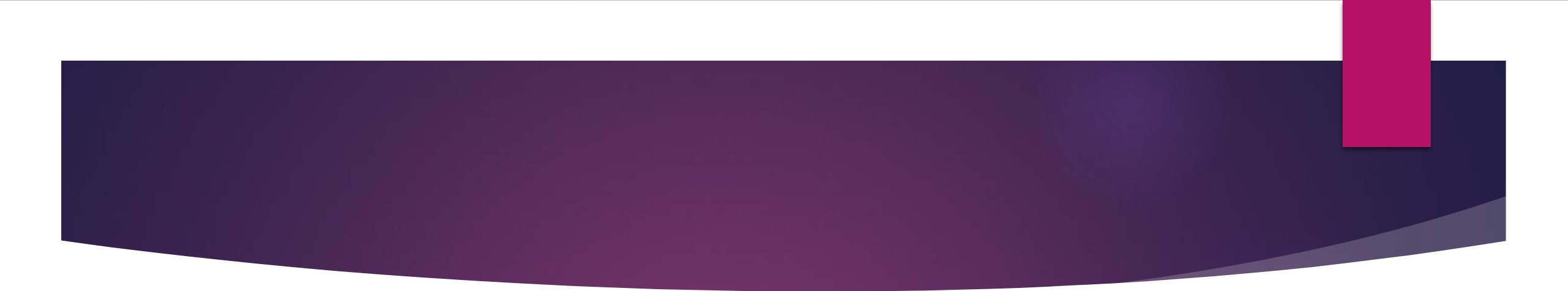
```
tor> npm install -D ts-loader
```

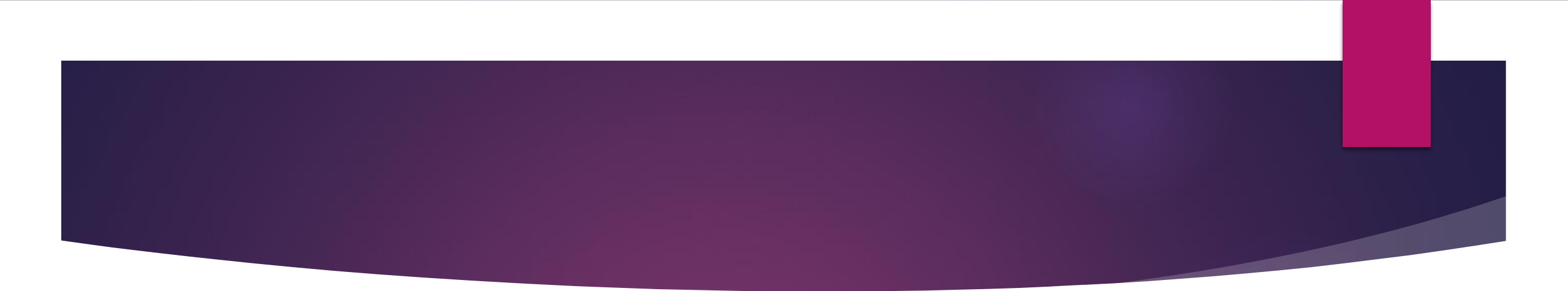
- ▶ La biblioteca ts-loader es un complemento de paquete web que puede cargar archivos TypeScript.

- 
- ▶ Ahora hemos creado todas las piezas individuales necesarias para convertir nuestra aplicación Angular en una de escritorio usando Electron.
 - ▶ Solo necesitamos juntarlos para que podamos construir y ejecutar nuestra aplicación de escritorio.
 - ▶ La pieza principal que organiza la aplicación Electron es el archivo de configuración del paquete web que necesitamos crear en la carpeta raíz de nuestro espacio de trabajo de Angular CLI:

src > electron > TS webpack.config.ts > ...

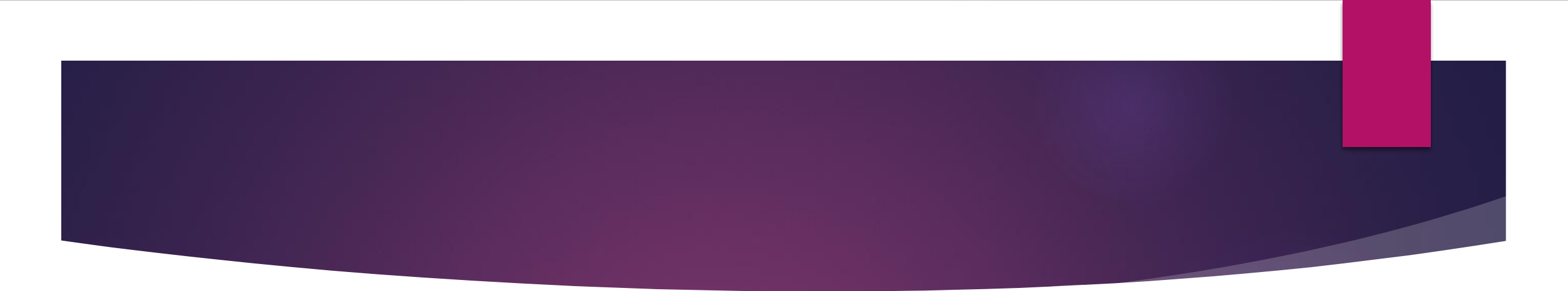
```
1  const path = require('path');
2  const src = path.join(process.cwd(), 'src', 'electron');
3  module.exports = {
4    mode: 'development',
5    devtool: 'source-map',
6    entry: path.join(src, 'main.ts'),
7    output: {
8      path: path.join(process.cwd(), 'dist', 'my-editor'),
9      filename: 'shell.js'
10   },
11   module: {
12     rules: [
13       {
14         test: /\.ts$/,
15         loader: 'ts-loader',
16         options: {
17           configFile: path.join(src, 'tsconfig.json')
18         }
19       }
20     ]
21   },
22   target: 'electron-main'
23 };
```

- 
- ▶ El archivo anterior configura el paquete web en nuestra aplicación usando las siguientes opciones:
 - ▶ mode: indica que actualmente nos estamos ejecutando en un entorno de desarrollo.
 - ▶ devtool: permite la generación de archivos de mapas de origen con fines de depuración.
 - ▶ module: indica al paquete web que cargue el complemento ts-loader para manejar archivos TypeScript.
 - ▶ target: indica que actualmente estamos ejecutando el proceso principal de Electron.

- 
- ▶ entry: indica el punto de entrada principal de la aplicación Electron, que es el archivo main.ts.
 - ▶ output: define la ruta y el nombre de archivo del paquete Electron que se generará a partir del paquete web. La propiedad de la ruta apunta a la misma carpeta que se usa desde la CLI de Angular para generar el paquete de la aplicación Angular. La propiedad del nombre de archivo se establece en shell.js porque el predeterminado generado a partir de webpack es main.js, y causará un conflicto con el archivo main.js generado desde la aplicación Angular.


- ▶ webpack ahora contiene toda la información necesaria para construir y empaquetar la aplicación Electron. Por otro lado, Angular CLI se encarga de construir la aplicación Angular. Veamos cómo podemos combinarlos y ejecutar nuestra aplicación de escritorio:
- ▶ Ejecute el siguiente comando npm para instalar simultáneamente el paquete npm:

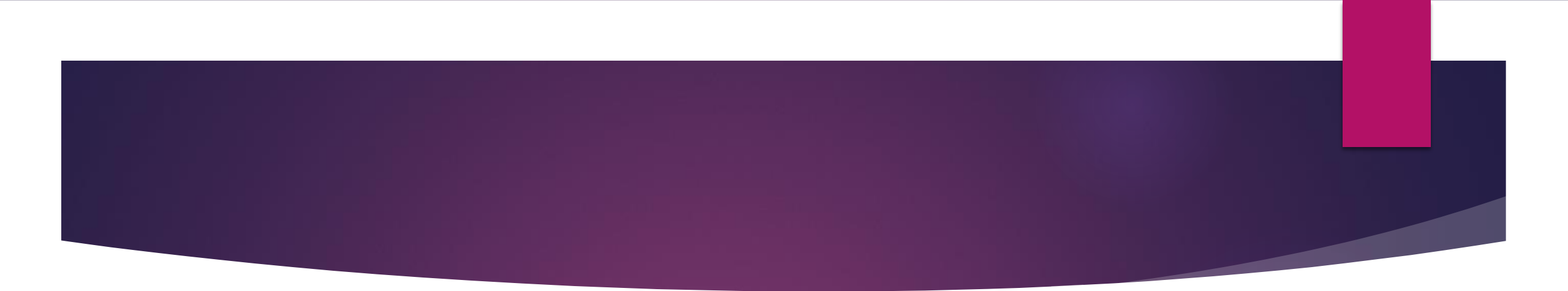
```
-editor> npm install -D concurrently
```

- 
- ▶ La biblioteca concurrently nos permite ejecutar múltiples procesos al mismo tiempo.
 - ▶ En nuestro caso, nos permitirá ejecutar las aplicaciones Angular y Electron en paralelo.

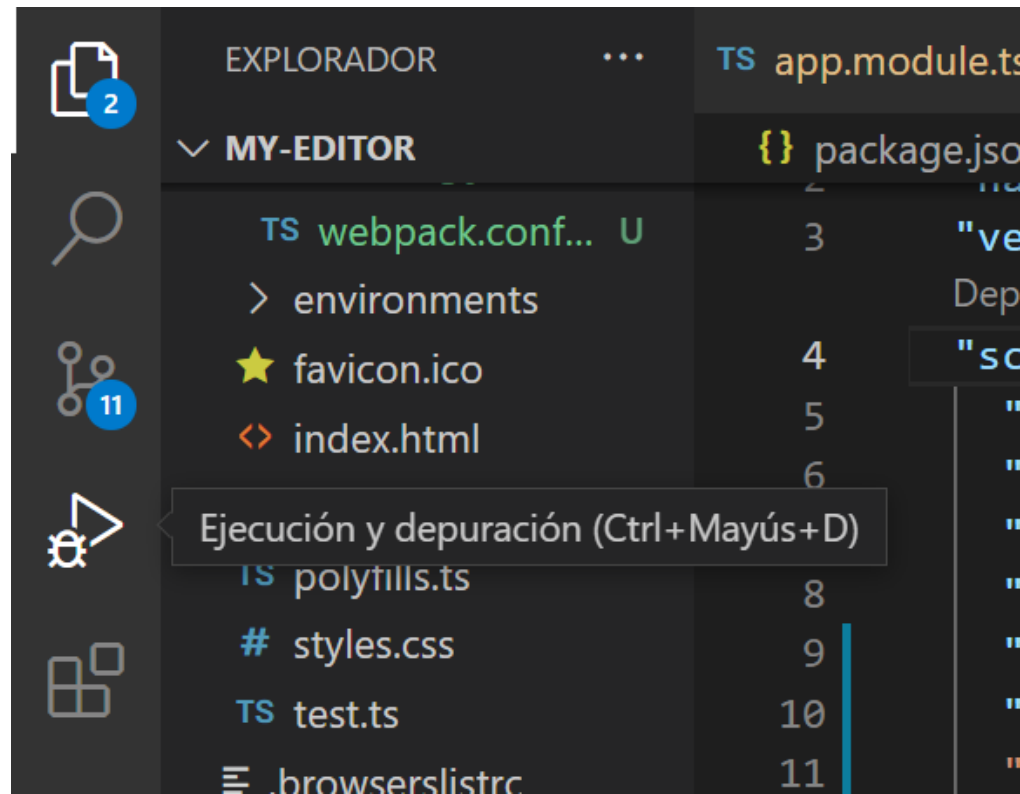
- Abra el archivo package.json y agregue una nueva entrada en la propiedad de scripts:

```
4  "scripts": {  
5    "ng": "ng",  
6    "start": "ng serve",  
7    "build": "ng build",  
8    "watch": "ng build --watch --configuration development",  
9    "test": "ng test",  
10   "start:desktop":  
11     "concurrently \"ng build --delete      output-path=false --watch\" \"webpack--watch\"",  
12  }
```

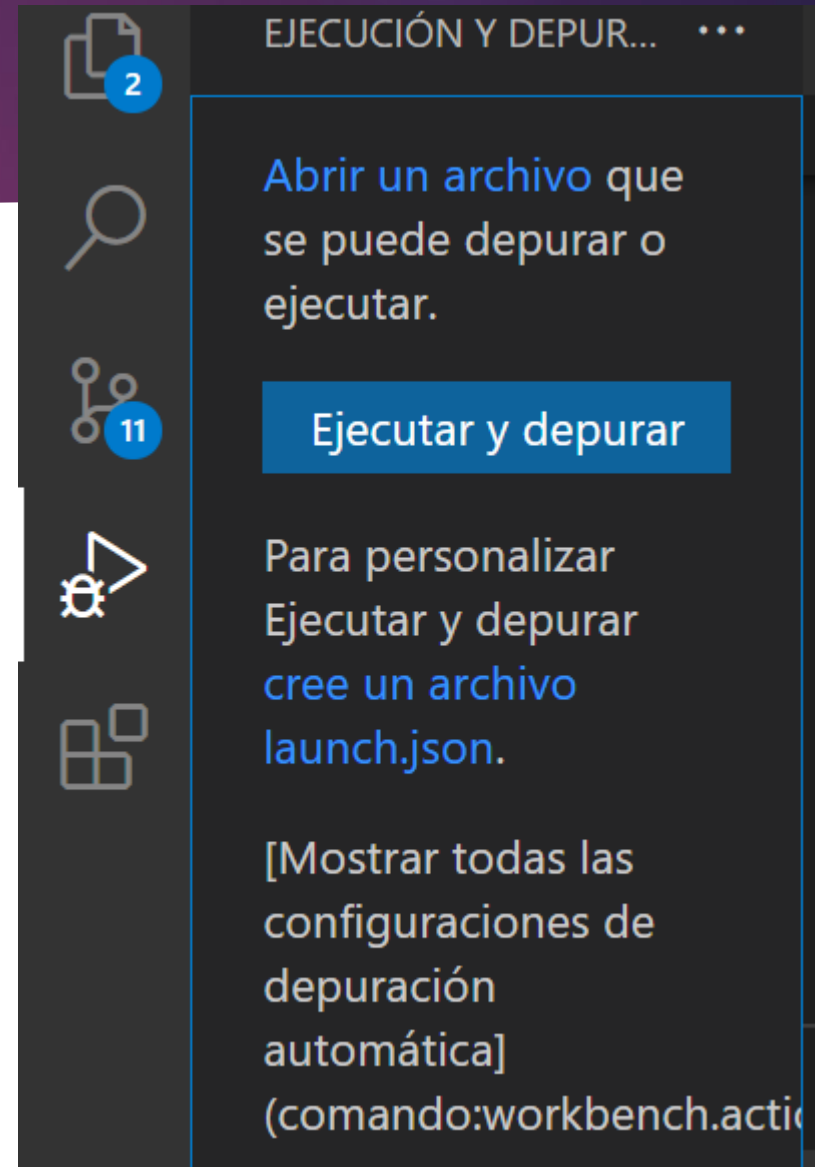


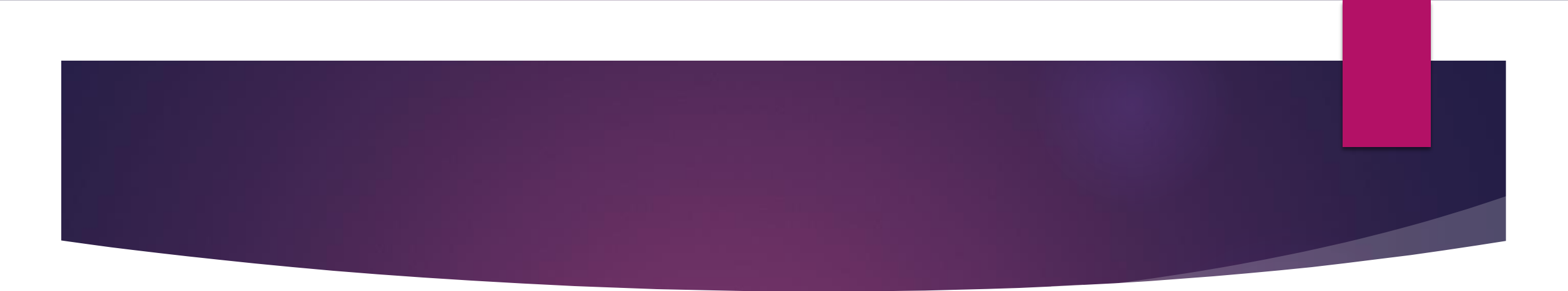
- 
- ▶ `start:desktop` construye la aplicación Angular usando el comando `ng build` de Angular CLI y la aplicación Electron usando el comando `webpack`.
 - ▶ Ambas aplicaciones se ejecutan en modo reloj usando la opción `--watch`, de modo que cada vez que hagamos un cambio en el código, la aplicación se reconstruirá para reflejar el cambio. Siempre que modifiquemos la aplicación Angular, la CLI de Angular eliminará la carpeta `dist` por defecto.
 - ▶ Podemos evitar este comportamiento usando la opción `--deleteoutput-path = false` porque la aplicación Electron también está construida en la misma carpeta.


- Haga clic en el menú Ejecutar que existe en la barra lateral de Visual Studio Code:




- En el panel Ejecutar y Depurar que aparece, haga clic en el enlace crear un archivo launch.json:



- 
- ▶ Visual Studio Code abrirá un menú desplegable que nos permite seleccionar el entorno para ejecutar nuestra aplicación. Una aplicación de Electron usa Node.js, así que seleccione cualquiera de las opciones disponibles de Node.js.
 - ▶ Visual Studio Code creará una carpeta `.vscode` en nuestro espacio de trabajo de Angular CLI con un archivo `launch.json` dentro. En el archivo `launch.json` que se ha abierto, establezca el valor de la propiedad `program` en `${workspaceRoot}/dist/my-editor/shell.js`.
 - ▶ La propiedad del programa indica la ruta absoluta del archivo del paquete Electron.




```
11 "skipFiles": [  
12 |   "<node_internals>/**"  
13 | ],  
14 "program": "${workspaceRoot}/dist/my-editor/shell.js",  
15 "outFiles": [  
16 |   "${workspaceFolder}/**/*.js"  
17 | ]
```



- 
- Agregue la siguiente entrada debajo de la propiedad program en el archivo launch.json:

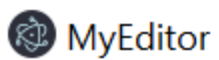
```
],  
"program": "${workspaceRoot}/dist/my-editor/shell.js",  
"runtimeExecutable": "${workspaceRoot}/node_modules/.bin/electron",  
"outFiles": [
```

- La propiedad runtimeExecutable define la ruta absoluta del ejecutable de Electron.

- 
- ▶ Ahora estamos listos para ejecutar nuestra aplicación de escritorio y obtener una vista previa. Ejecute el siguiente comando npm para compilar la aplicación:

```
editor> npm run start:desktop
```

- ▶ El comando anterior construirá primero la aplicación Electron y luego la Angular.
- ▶ Espere a que finalice la compilación angular y luego presione F5 para obtener una vista previa de la aplicación:



MyEditor



File Edit View Window Help



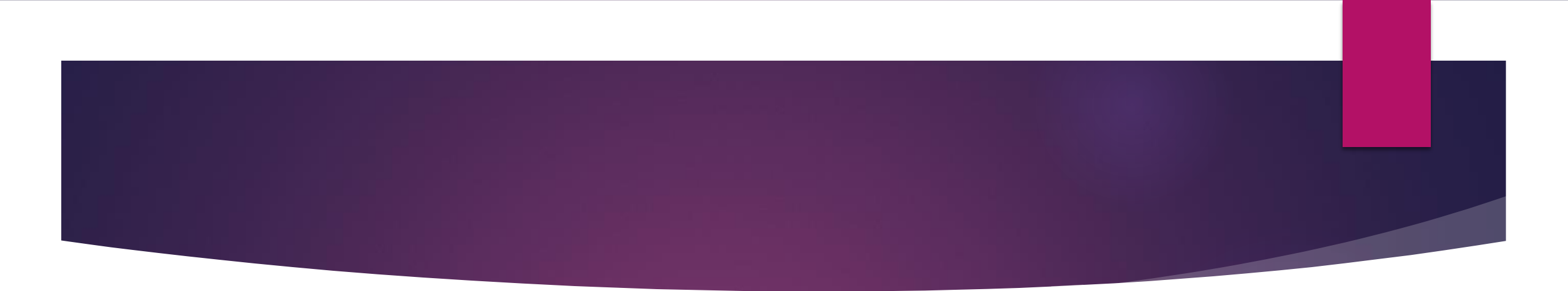
B

I



U

Enter your content

- 
- ▶ En la captura de pantalla anterior, podemos ver que nuestra aplicación Angular con el editor WYSIWYG está alojada dentro de una ventana de escritorio nativa. Contiene las siguientes características que solemos encontrar en aplicaciones de escritorio:
 - ▶ El encabezado con un icono
 - ▶ El menú principal
 - ▶ Botones de Minimizar, maximizar y cerrar
 - ▶ La aplicación Angular se procesa dentro del navegador Chromium. Para verificar eso, haga clic en el elemento del menú Ver y seleccione la opción Alternar herramientas de desarrollo.
 - ▶ ¡Bien hecho! Ha logrado crear su propio editor WYSIWYG de escritorio.

Comunicación entre Angular y Electron

- ▶ De acuerdo con las especificaciones del proyecto, el contenido del editor WYSIWYG debe conservarse en el sistema de archivos local. Además, el contenido se cargará desde el sistema de archivos al iniciar la aplicación.
- ▶ La aplicación Angular maneja cualquier interacción entre el editor WYSIWYG y sus datos usando el proceso del renderizador, mientras que la aplicación Electron administra el sistema de archivos con el proceso principal. Por lo tanto, necesitamos establecer un mecanismo de IPC para comunicarse entre los dos procesos de Electron de la siguiente manera:
 - ▶ Configuración del espacio de trabajo de Angular CLI
 - ▶ Interactuar con el editor
 - ▶ Interactuar con el sistema de archivos

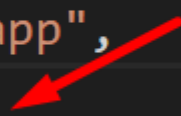
Configuración del espacio de trabajo de Angular CLI

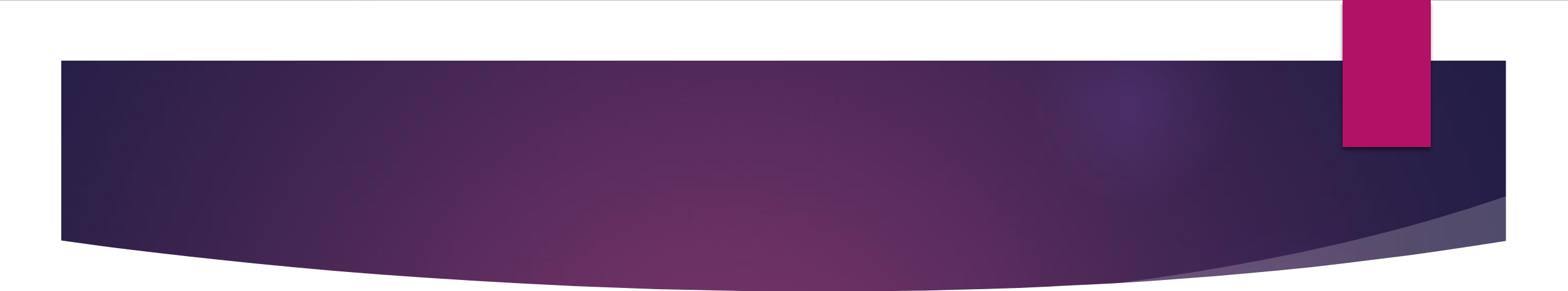
- Necesitamos modificar varios archivos para configurar el espacio de trabajo de nuestra aplicación:
- Abra el archivo `main.ts` que existe en la carpeta `src\electron` y establezca la propiedad `nodeIntegration` en `true` en el constructor `BrowserWindow`:

```
src > electron > TS main.ts > ...  
6  function createWindow () {  
7      const mainWindow = new BrowserWindow({  
8          width: 800,  
9          height: 600,  
10         webPreferences: {  
11             nodeIntegration: true,  
12             contextIsolation: false  
13         }  
14     });  
15
```

- ▶ La bandera anterior habilitará Node.js en el proceso del renderizador y expondrá la interfaz ipcRenderer, que necesitaremos para comunicarnos con el proceso principal.
- ▶ Abra el archivo tsconfig.app.json que existe en la carpeta raíz del espacio de trabajo de AngularCLI y agregue la entrada del electrón dentro de la propiedad de tipos:

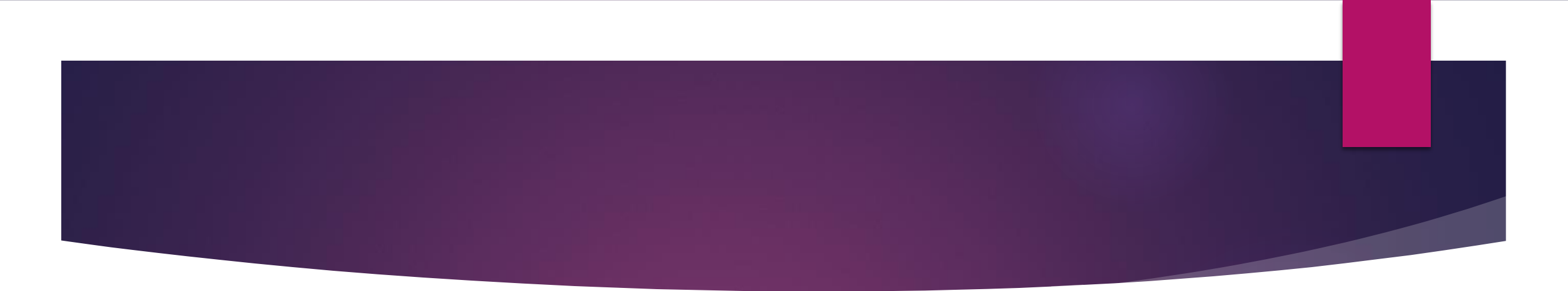
```
2  {
3    "extends": "./tsconfig.json",
4    "compilerOptions": {
5      "outDir": "./out-tsc/app",
6      "types": ["electron"]
7    },
8    "files": [
```

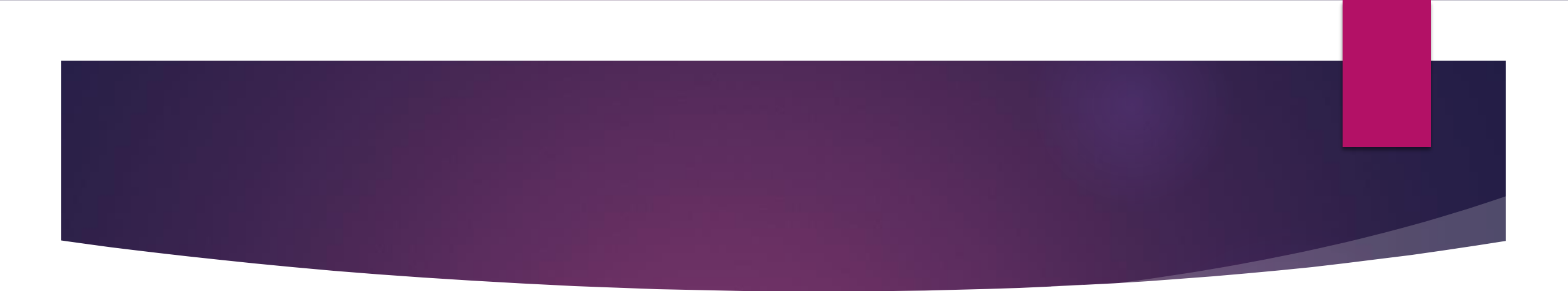


- 
- ▶ El framework de Electron incluye tipos que podemos usar en nuestra aplicación Angular.
 - ▶ Cree un nuevo archivo llamado `window.ts` dentro de la carpeta `src\app` e ingrese el siguiente código:

src > app > TS window.ts > ...

```
1  import { InjectionToken } from '@angular/core';
2  export const WINDOW = new
3    |   InjectionToken<Window>('Global window object', {
4    |     factory: () => window
5    |   });
6  export interface ElectronWindow extends Window {
7    |   require(module: string): any;
8  }
```

- 
- ▶ El framework de Electron es un módulo de JavaScript que se puede cargar desde el objeto de ventana global del navegador. Usamos la interfaz InjectionToken para hacer que el objeto de la ventana sea inyectable para que podamos usarlo en nuestros componentes y servicios de Angular. Además, utilizamos un método de fábrica para devolverlo, de modo que sea fácil reemplazarlo en plataformas sin acceso al objeto de ventana, como el servidor.
 - ▶ Electron se carga mediante el método require del objeto de ventana, que solo está disponible en el entorno Node.js. Para usarlo en una aplicación Angular, creamos la interfaz ElectronWindow que extiende la interfaz de Windows definiendo ese método.

- 
- ▶ Las aplicaciones Angular y Electron ahora están listas para interactuar entre sí utilizando el mecanismo IPC.
 - ▶ Comencemos a implementar la lógica necesaria en la aplicación Angular primero.

Interactuando con el editor

- ▶ La aplicación Angular es responsable de administrar el editor WYSIWYG. El contenido del editor se mantiene sincronizado con el sistema de archivos mediante el proceso de renderizado de Electron. Averigüemos cómo usar el proceso de renderizado:
- ▶ Cree un nuevo servicio Angular usando el comando generate de Angular CLI:

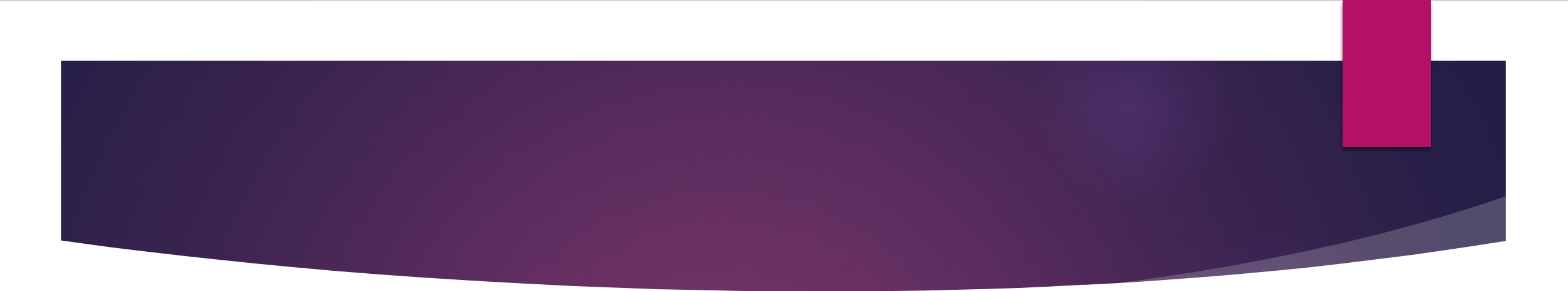
```
tor> ng generate service editor
```

- Abra el archivo editor.service.ts e inyecte el token WINDOW en el constructor de la clase EditorService:

```
src > app > TS editor.service.ts > ...  
1  import { Inject } from '@angular/core';  
2  import { Injectable } from '@angular/core';  
3  import { ElectronWindow, WINDOW } from './window';  
4  @Injectable({  
5    providedIn: 'root'  
6  })  
7  export class EditorService {  
8    constructor(@Inject(WINDOW) private window:  
9      ElectronWindow) { }  
10 }
```

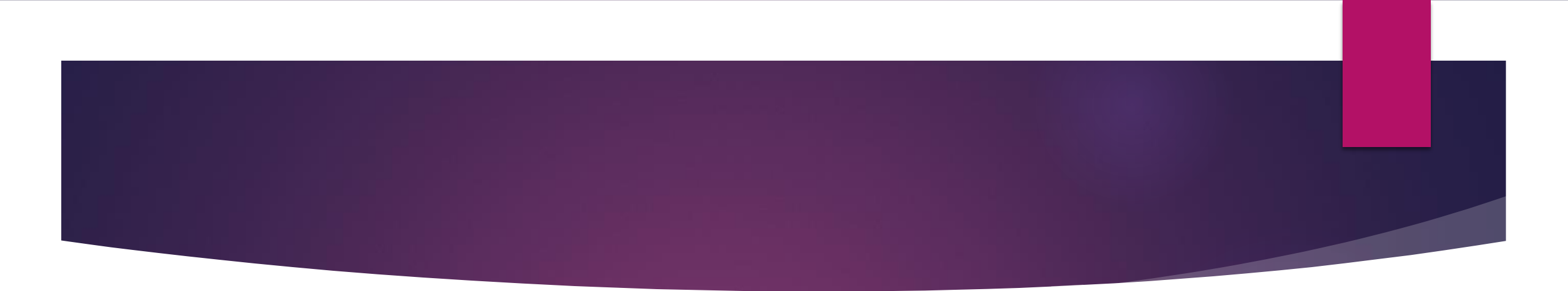
- Cree una propiedad getter que devuelva el objeto ipcRenderer del módulo de electron:

```
ElectronWindow) [ ]  
private get ipcRenderer(): Electron.IpcRenderer {  
    return this.window.require('electron').ipcRenderer;  
}
```

- 
- ▶ El módulo de electron es el módulo principal del framework electron que da acceso a varias propiedades, incluido el proceso principal y el de renderizado.
 - ▶ También establecemos el tipo de la propiedad ipcRenderer en `electron.IpcRenderer`, que es parte de los tipos integrados de Electron.

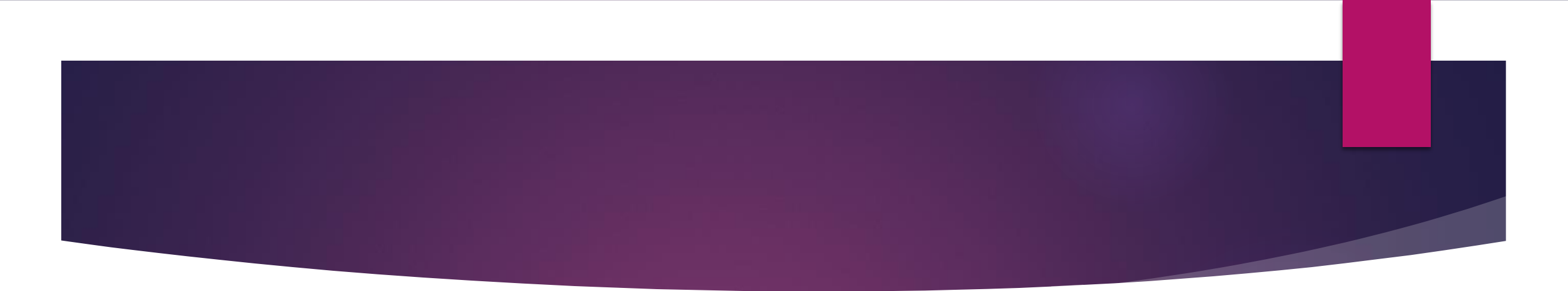
- 
- Cree un método al que se llamará para obtener el contenido del editor del sistema de archivos:

```
getContent(): Promise<string> {  
  return this.ipcRenderer.invoke('getContent');  
}
```

- 
- ▶ Usamos el método `invoke` de la propiedad `ipcRenderer`, pasando el nombre del canal de comunicación como parámetro. El resultado del método `getContent` es un objeto `Promise` del tipo cadena, ya que el contenido del editor son datos de texto sin procesar.
 - ▶ El método de invocación inicia una conexión con el proceso principal a través del canal `getContent`.
 - ▶ En la sección Interactuar con el sistema de archivos, veremos cómo configurar el proceso principal para responder a la llamada al método de invocación en ese canal.

- 
- Cree un método al que se llamará para guardar el contenido del editor en el sistema de archivos:

```
setContent(content: string) {  
  this.ipcRenderer.invoke('setContent', content);  
}
```

- 
- ▶ El método setContent vuelve a llamar al método invoke del objeto ipcRenderer pero con un nombre de canal diferente.
 - ▶ También utiliza el segundo parámetro del método de invocación para pasar datos al proceso principal.
 - ▶ En este caso, el parámetro de contenido contendrá el contenido del editor. Veremos cómo configurar el proceso principal para el manejo de datos en la sección Interactuar con el sistema de archivos.

- Abra el archivo editor.component.ts y cree una propiedad myContent para contener los datos del editor. Además, inyecte EditorService en el constructor de la clase EditorComponent:

```
1 import { Component, OnInit } from '@angular/core';
2 import { EditorService } from '../editor.service';
3 @Component({
4   selector: 'app-editor',
5   templateUrl: './editor.component.html',
6   styleUrls: ['./editor.component.css']
7 })
8 export class EditorComponent implements OnInit {
9   myContent = '';
10   constructor(private editorService: EditorService) { }
11
12   ngOnInit(): void {
13   }
```

- Cree un método que llame al método getContent de la variable editorService y ejecútelo dentro del método ngOnInit:

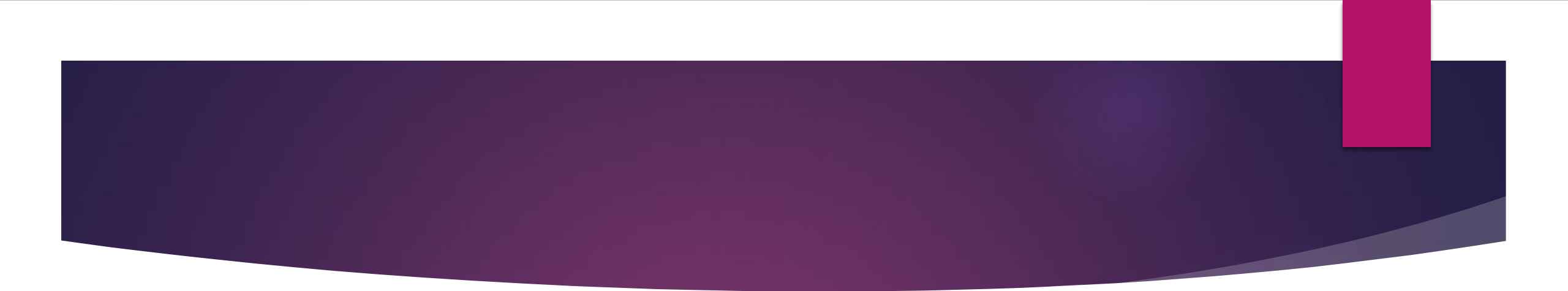
```
ngOnInit(): void {  
  this.getContent();  
}  
private async getContent() {  
  this.myContent = await  
    this.editorService.getContent();  
}
```

- ▶ Usamos la sintaxis `async/await`, que permite la ejecución sincrónica de nuestro código en llamadas a métodos basados en promesas.
- ▶ Cree un método que llame al método `setContent` de la variable `editorService`:


```
saveContent(content: string) {  
    this.editorService.setContent(content);  
}
```

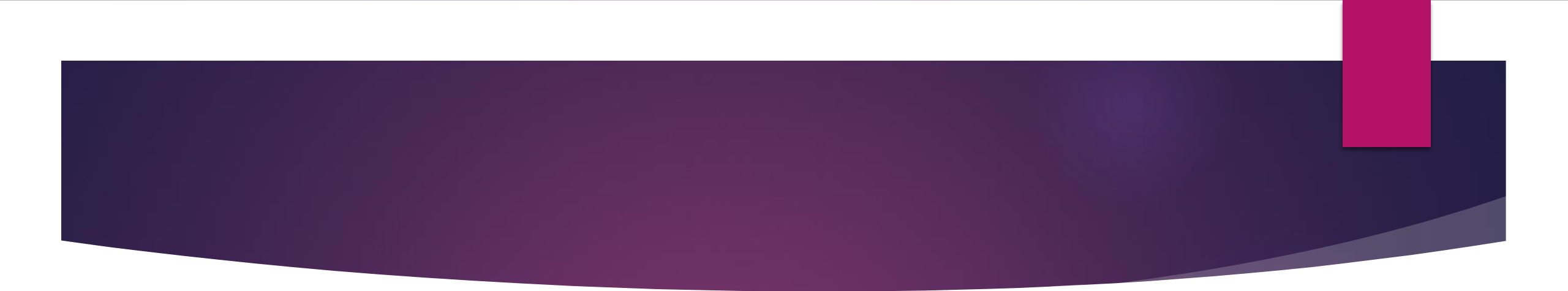
- Conectemos esos métodos que hemos creado con la plantilla del componente. Abra el archivo `editor.component.html` y agregue los siguientes bindings:

```
src > app > editor > <> editor.component.html > ...  
1   <ngx-wig placeholder="Enter your content"  
2   [ngModel]="myContent"  
3   (contentChange)="saveContent($event)">  
4  
5   </ngx-wig>  
6
```


- 
- ▶ Usamos la directiva `ngModel` para vincular el modelo del editor a la propiedad del componente `myContent`, que se usará para mostrar el contenido inicialmente.
 - ▶ También usamos el binding de eventos `contentChange` para guardar el contenido del editor cada vez que cambia, es decir, mientras el usuario escribe.

- ▶ La directiva `ngModel` es parte del paquete `@angular/forms`.
- ▶ Importe `FormsModule` en el archivo `app.module.ts` para usarlo:

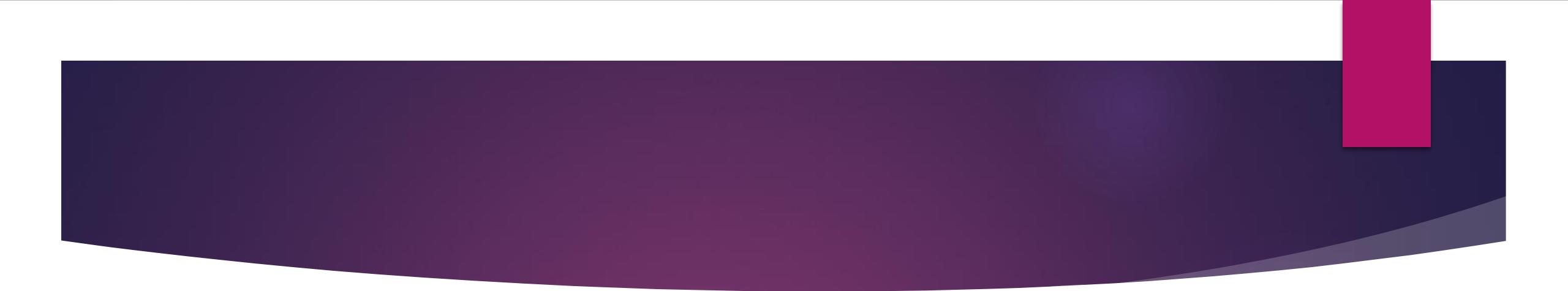
```
12     ],  
13     imports: [  
14         BrowserModule,  
15         NgxWigModule,  
16         FormsModule,   
17     ],  
18     providers: [],
```

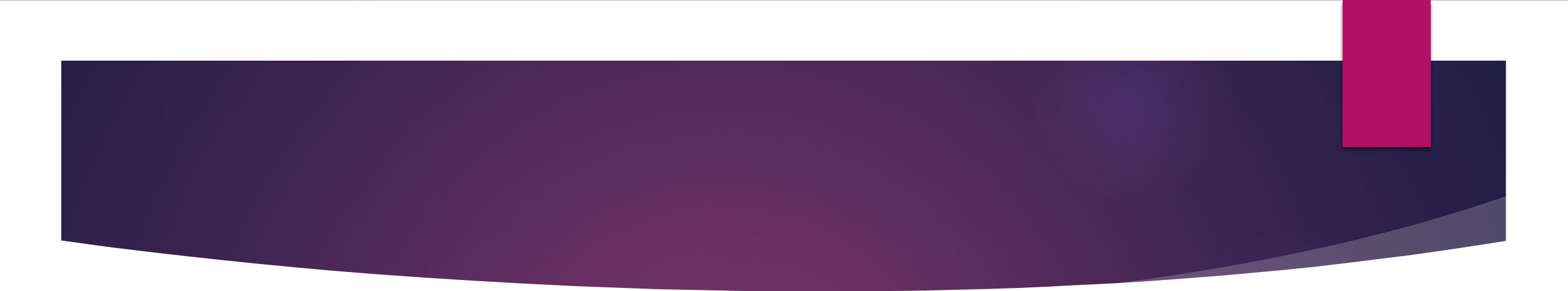
- 
- ▶ Ahora hemos implementado toda la lógica para que nuestra aplicación Angular se comuniquen con el proceso principal.
 - ▶ Ha llegado el momento de implementar el otro extremo del mecanismo de comunicación, la aplicación Electron y su proceso main.

Interactuar con el sistema de archivos

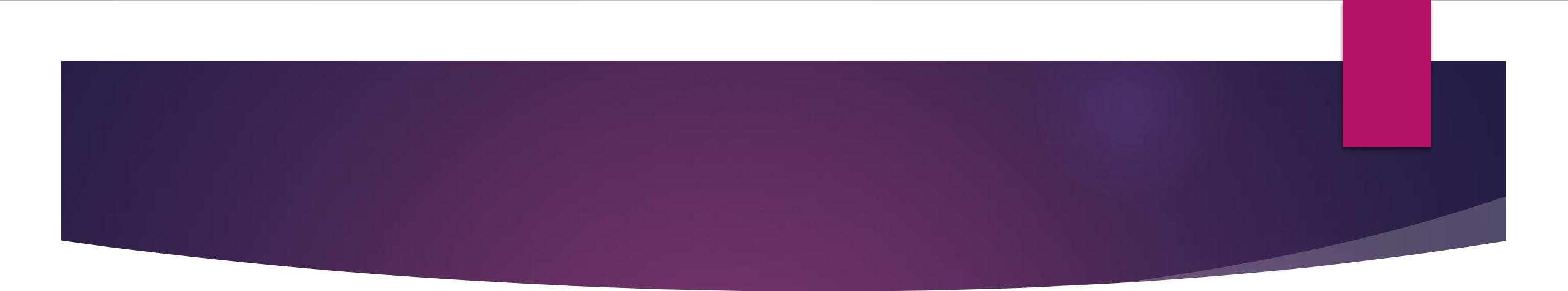
- ▶ El proceso main interactúa con el sistema de archivos utilizando la biblioteca fs Node.js, que está integrada en el marco de Electron. Veamos cómo podemos usarlo:
- ▶ Abra el archivo main.ts que existe en la carpeta src\electron e importe los siguientes artefactos:

```
import { app, BrowserWindow, ipcMain } from 'electron';  
import * as fs from 'fs';  
import * as path from 'path';
```

- 
- ▶ La biblioteca fs es responsable de interactuar con el sistema de archivos.
 - ▶ La biblioteca de rutas proporciona utilidades para trabajar con rutas de archivos y carpetas.
 - ▶ El objeto ipcMain nos permite trabajar con el proceso principal de Electron.

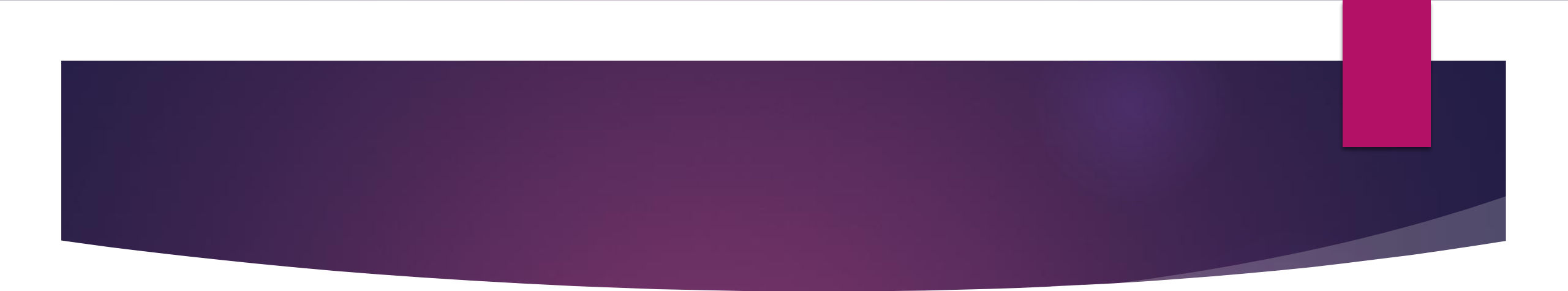
- 
- Cree una variable que contenga la ruta del archivo que contiene el contenido del editor:

```
const contentFile = path.join(app.getPath('userData'), 'content.html');
```

- 
- ▶ El archivo que guarda el contenido del editor es el archivo `content.html` que existe dentro de la carpeta reservada `userData`. La carpeta `userData` es un alias para una carpeta de sistema de propósito especial, diferente para cada sistema operativo, y se usa para almacenar archivos específicos de la aplicación, como la configuración. Puede encontrar más detalles sobre la carpeta `userData` y otras carpetas del sistema en <https://www.electronjs.org/docs/api/app#appgetpathname>.
 - ▶ El método `getPath` del objeto de la aplicación funciona en varias plataformas y se utiliza para obtener la ruta de carpetas especiales, como el directorio de inicio de un usuario o los datos de la aplicación.

- Llame al método `handle` del objeto `ipcMain` para comenzar a escuchar solicitudes en el canal `getContent`:

```
ipcMain.handle('getContent', () => {  
  if (fs.existsSync(contentFile)) {  
    const result = fs.readFileSync(contentFile);  
    return result.toString();  
  }  
  return '';  
});
```


- 
- ▶ Cuando el proceso principal recibe una solicitud en este canal, usa el método `existSync` de la biblioteca `fs` para verificar si el archivo con el contenido del editor ya existe. Si existe, lo lee usando el método `readFileSync` y devuelve su contenido al proceso de renderizado.
 - ▶ Vuelva a llamar al método `handle`, pero esta vez para el canal `setContent`:

- ▶ En el fragmento anterior, usamos el método `writeFileSync` de la biblioteca `fs` para escribir el valor de la propiedad de contenido en el archivo.
- ▶ Abra el archivo `package.json` y cambie la versión del paquete `@types/node`:

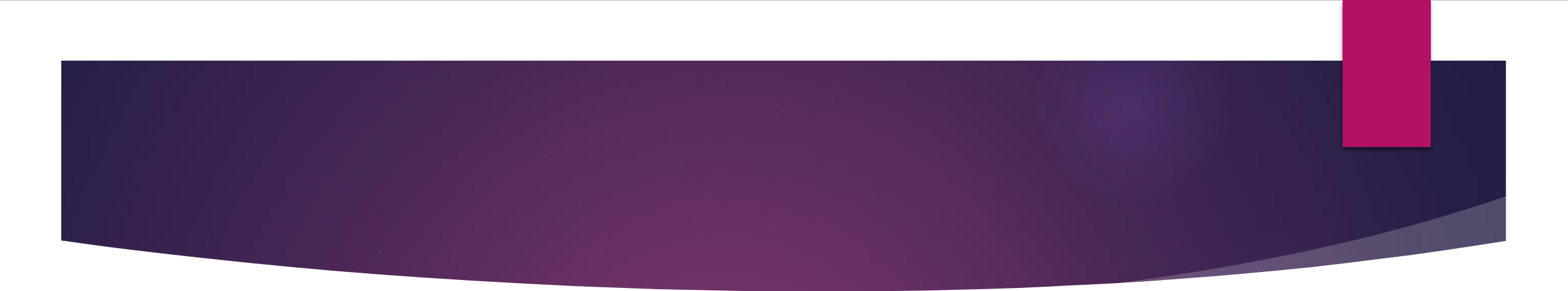
```
1  "@types/jasmine": "~3.8.0",  
2  "@types/node": "^15.6",  
3  "concurrently": "^6.3.0",
```

- ▶ Ahora que hemos conectado la aplicación Angular y Electron, es hora de obtener una vista previa de nuestra aplicación de escritorio WYSIWYG:
- ▶ Ejecute el script `start: desktop npm` y presione F5 para ejecutar la aplicación.
- ▶ Utilice el editor y su barra de herramientas para ingresar contenido como el siguiente:



Bienvenidos a DAW II

Aplicación que se hizo con Angular y Electron!!!

- 
- ▶ Cierre la ventana de la aplicación y vuelva a ejecutar la aplicación. Si todo funcionó correctamente, debería ver el contenido que ingresó dentro del editor.
 - ▶ ¡Felicidades! Ha enriquecido su editor WYSIWYG al agregarle capacidades de persistencia.
 - ▶ En la siguiente sección, daremos el último paso hacia la creación de nuestra aplicación de escritorio y aprenderemos cómo empaquetarla y distribuirla.

Empaquetar una aplicación de escritorio

- ▶ Las aplicaciones web generalmente se empaquetan y se implementan en un servidor web que las aloja. Por otro lado, las aplicaciones de escritorio se empaquetan como un solo archivo ejecutable que se puede distribuir fácilmente. Empaquetar nuestra aplicación WYSIWYG requiere los siguientes pasos:
 - ▶ Configuración del paquete web para el modo de producción
 - ▶ Usando un empaquetador de electron

Configuración de webpack para producción

- ▶ Ya hemos creado un archivo de configuración de paquete web para el entorno de desarrollo. Ahora necesitamos crear uno nuevo para producción. Ambos archivos de configuración compartirán alguna funcionalidad, así que comencemos creando uno común:
- ▶ Cree un archivo `webpack.dev.config.js` en la carpeta raíz del espacio de trabajo de Angular CLI con el siguiente contenido:

JS webpack.dev.config.js > ...

```
1  const path = require('path');
2  const baseConfig = require('./webpack.config');
3  module.exports = {
4      ...baseConfig,
5      mode: 'development',
6      devtool: 'source-map',
7      output: {
8          path: path.join(process.cwd(), 'dist', 'my-editor'),
9          filename:
10             'shell.js'
11         }
12     };
13
```

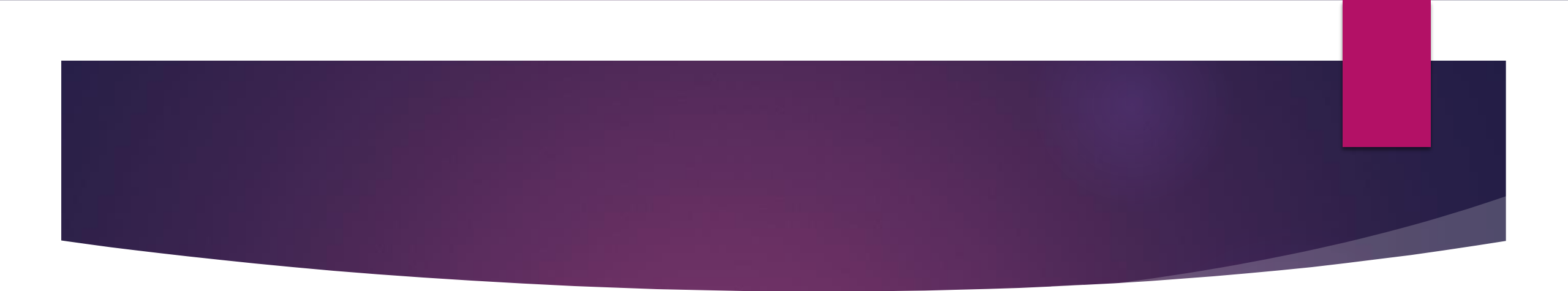
- ▶ Elimine las propiedades de mode, devtool y output del archivo webpack.config.js.
- ▶ Abra el archivo package.json y pase el nuevo archivo de configuración de desarrollo del paquete web al script start:desktop :

```
ent",
```

```
e --watch\" \"webpack --config webpack.dev.config.js --watch\""
```


- Cree un archivo `webpack.prod.config.js` en la carpeta raíz del espacio de trabajo de Angular CLI con el siguiente contenido:

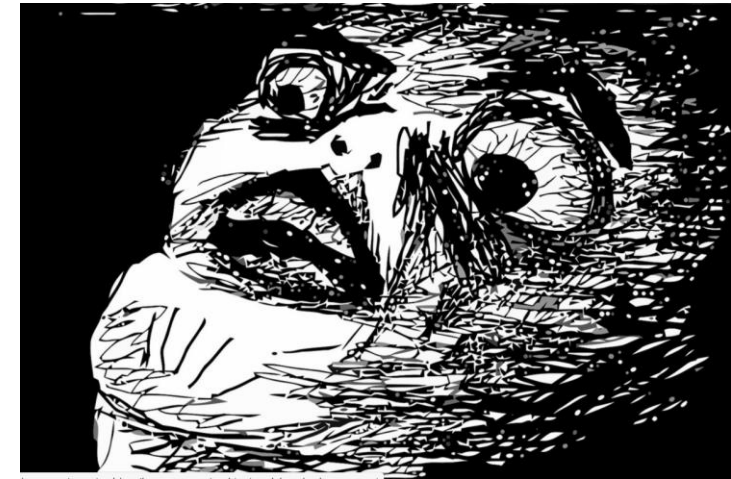
```
JS webpack.prod.config.js > ...
1  const path = require('path');
2  const baseConfig = require('./webpack.config');
3  module.exports = {
4      ...baseConfig,
5      output: {
6          path: path.join(process.cwd(), 'dist', 'my-editor'),
7          filename:
8              'main.js'
9      }
10 }
11 ;
```

- 
- ▶ La principal diferencia con el archivo de configuración de webpack para el entorno de desarrollo es que cambiamos el nombre del archivo del paquete de salida a main.js.
 - ▶ Angular CLI agrega un número hash en el archivo main.js de la aplicación Angular en producción, por lo que no habrá conflictos.
 - ▶ Otras cosas a tener en cuenta es que el modo está configurado en producción de forma predeterminada cuando lo omitimos, y falta la propiedad devtool porque no queremos habilitar los mapas de origen en el modo de producción.

- 
- Agregue una nueva entrada en la propiedad scripts del archivo package.json para construir nuestra aplicación en modo de producción:

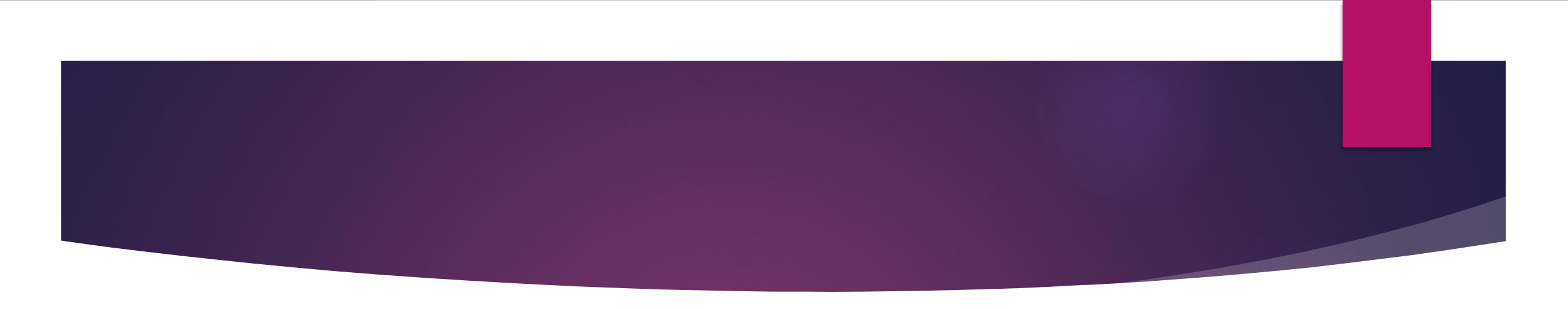
```
start:desktop :  
"concurrently \"ng build --delete-output-path=false --watch\" \"webpack -  
\"build:electron\": \"ng build && webpack --config webpack.prod.config.js"
```

- ▶ El script `build:electron` construye la aplicación Angular y Electron en modo de producción simultáneamente.
- ▶ Hemos completado todas las configuraciones necesarias para empaquetar nuestra aplicación de escritorio.
- ▶ En la siguiente sección, aprenderemos cómo convertirlo en un solo paquete específico para cada sistema operativo.



Usando un paquete de electron

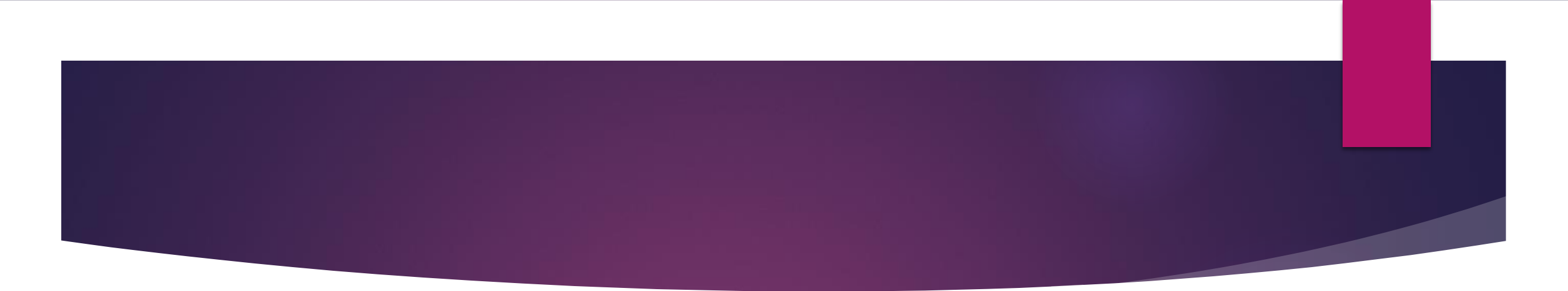
- ▶ El framework Electron tiene una amplia variedad de herramientas creadas y mantenidas por la comunidad de código abierto.
- ▶ Puede ver una lista de proyectos disponibles en la sección Herramientas en el siguiente enlace: <https://www.electronjs.org/community>
- ▶ Una de estas herramientas es la biblioteca electron-packager, que podemos usar para empaquetar nuestra aplicación de escritorio como un solo archivo ejecutable para cada sistema operativo (Windows, Linux y macOS).

- 
- Ejecute el siguiente comando npm para instalar electron-packager como una dependencia de desarrollo para nuestro proyecto:

```
my-editor> npm install -D electron-packager
```

- Agregue una nueva entrada en la propiedad scripts del archivo package.json para empaquetar nuestra aplicación:

```
"start:desktop": "concurrently \"ng build --delete-output-path=false --watch\"  
"build:electron": "ng build && webpack --config webpack.prod.config.js",  
"package": "electron-packager dist/my-editor --out=dist --asar"  
},
```

- 
- ▶ En el script anterior, electron-packager leerá todos los archivos en la carpeta dist/my-editor, los empaquetará y generará el paquete final en la carpeta dist.
 - ▶ La opción --asar indica al empaquetador que archive todos los archivos en formato ASAR, similar a un archivo ZIP o TAR.


- Cree un archivo package.json en la carpeta src\electron y agregue el siguiente contenido:

```
src > electron > {} package.json > ...  
1  {  
2      "name": "my-editor",  
3      "main": "main.js"  
4  }
```

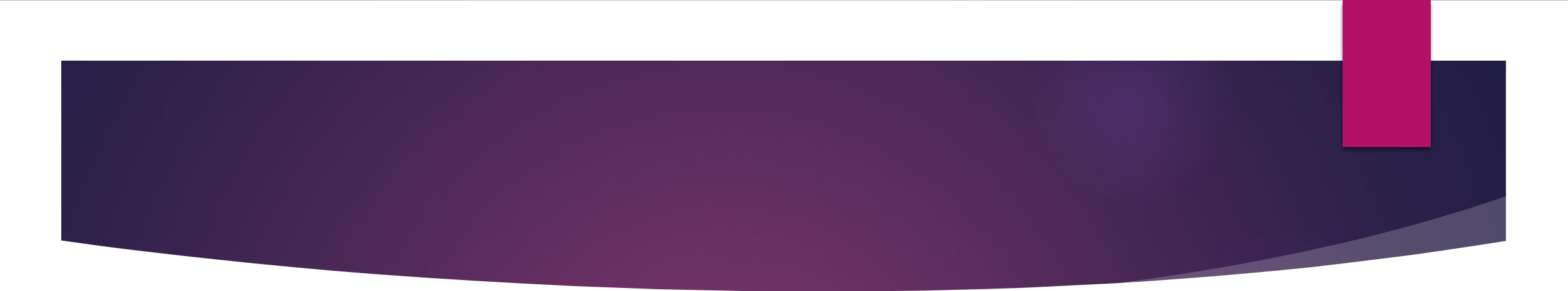
- La biblioteca electron-packager requiere que un archivo package.json esté presente en la carpeta de salida y apunte al archivo de entrada principal de la aplicación Electron.

- Abra el archivo webpack.prod.config.js y agregue el CopyWebpackPlugin en la propiedad de los complementos:

```
JS webpack.prod.config.js > [🔗] CopyWebpackPlugin
1  const path = require('path');
2  const baseConfig = require('./webpack.config');
3  const CopyWebpackPlugin = require('copy-webpack-plugin');
4  module.exports = {
```



```
11     },
12     plugins: [
13         new CopyWebpackPlugin({
14             patterns: [
15                 {
16                     context: path.join(process.cwd(), 'src',
17                     'electron'),
18                     from: 'package.json'
19                 }
20             ]
21         })
22     ]
```

- 
- ▶ Usamos CopyWebpackPlugin para copiar el archivo package.json de la carpeta src\electron a la carpeta dist\my-editor mientras construimos la aplicación en modo de producción.
 - ▶ Ejecute el siguiente comando para construir la aplicación en modo de producción:

```
\my-editor> npm run build:electron
```

JS webpack.prod.config.js > ...

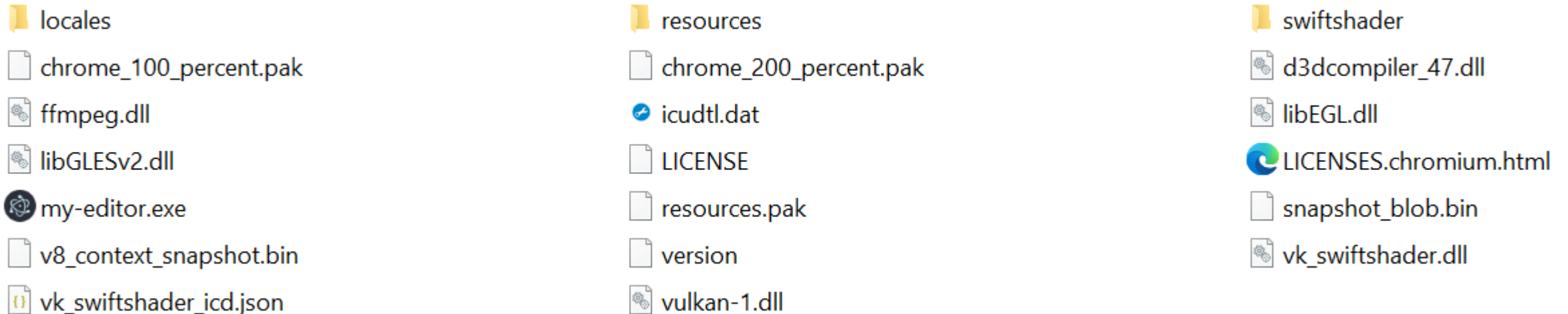
```
1  const path = require('path');
2  const baseConfig = require('./webpack.config');
3  const CopyWebpackPlugin = require('copy-webpack-plugin');
4
5  module.exports = {
6    ...baseConfig,
7    output: {
8      path: path.join(process.cwd(), 'dist', 'my-editor'),
9      filename: 'main.js'
10   },
11   plugins: [
12     new CopyWebpackPlugin({
13       patterns: [
14         {
15           context: path.join(process.cwd(), 'src', 'electron'),
16           from: 'package.json'
17         }
18       ]
19     })
20   ]
21 };|
```

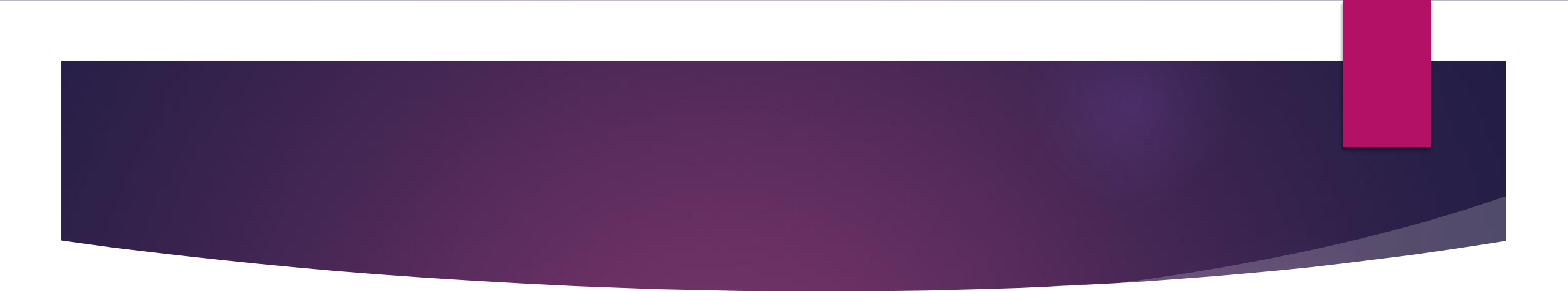
- ▶ Ahora ejecute el siguiente comando npm para empaquetarlo:

```
y-editor> npm run package
```

- ▶ El comando anterior empaquetará la aplicación para el sistema operativo en el que se está ejecutando actualmente, que es el comportamiento predeterminado de la biblioteca de empaquetadores de electron.
- ▶ Puede modificar este comportamiento pasando opciones adicionales, que encontrará en el repositorio de GitHub de la biblioteca que se enumera en la sección Lectura adicional.

- ▶ Navegue a la carpeta dist del espacio de trabajo de Angular CLI. Encontrará una carpeta llamada my-editor- {OS}, donde {OS} es su sistema operativo actual y su arquitectura.
- ▶ Por ejemplo, en Windows, será my-editor-win32-x64. Abra la carpeta y obtendrá los siguientes archivos:



- 
- ▶ En la captura de pantalla anterior, el archivo my-editor.exe es el archivo ejecutable de nuestra aplicación de escritorio. El código de nuestra aplicación no está incluido en este archivo, sino en el archivo app.asar, que existe en la carpeta de recursos.
 - ▶ Ejecute el archivo ejecutable y la aplicación de escritorio debería abrirse normalmente. Puede tomar toda la carpeta y cargarla en un servidor o distribuirla por cualquier otro medio. Su editor WYSIWYG ahora puede llegar a muchos más usuarios, como aquellos que están desconectados la mayor parte del tiempo.

Resumen

- ▶ En este capítulo, creamos un editor WYSIWYG para el escritorio usando Angular y Electron. Inicialmente, creamos una aplicación Angular y agregamos ngx-wig, una popular biblioteca Angular WYSIWYG. Luego, aprendimos cómo construir una aplicación de Electron.
- ▶ Se implementó un mecanismo de comunicación para intercambiar datos entre la aplicación Angular y la aplicación Electron. Finalmente, aprendimos cómo agrupar nuestra aplicación para empaquetarla y prepararla para su distribución.

Preguntas de práctica

- ▶ ¿Qué clase es responsable de crear una ventana de escritorio en Electron?
- ▶ ¿Cómo nos comunicamos entre los procesos principal y renderizador en Electron?
- ▶ ¿Qué bandera habilita el uso de Node.js en el proceso de renderizado?
- ▶ ¿Cómo convertimos un objeto JavaScript global en uno inyectable Angular?
- ▶ ¿Cómo cargamos Electron en una aplicación Angular?
- ▶ ¿Qué interfaz usamos para interactuar con Electron en una aplicación Angular?
- ▶ ¿Cómo pasamos datos al proceso principal de Electron desde una aplicación Angular?
- ▶ ¿Qué paquete usamos para la manipulación del sistema de archivos en Electron?
- ▶ ¿Qué biblioteca utilizamos para empaquetar una aplicación de Electron?

Lecturas recomendadas

- ▶ Electron: <https://www.electronjs.org/>
- ▶ Inicio rápido Electron: <https://www.electronjs.org/docs/tutorial/quick-start>
- ▶ ngx-wig: <https://github.com/steevermeister/ngx-wig>
- ▶ Configuración webpack: <https://webpack.js.org/configuration/>
- ▶ ts-loader: <https://webpack.js.org/guides/typescript/>
- ▶ Inyectando un objeto en Angular:
<https://angular.io/guide/dependencyinjection-providers#injecting-an-object>
- ▶ Sistema de archivos Node.js API: <https://nodejs.org/api/fs.html>
- ▶ electron-packager: <https://github.com/electron/electronpackager>
- ▶ concurrently: <https://github.com/kimmobrunfeldt/concurrently>