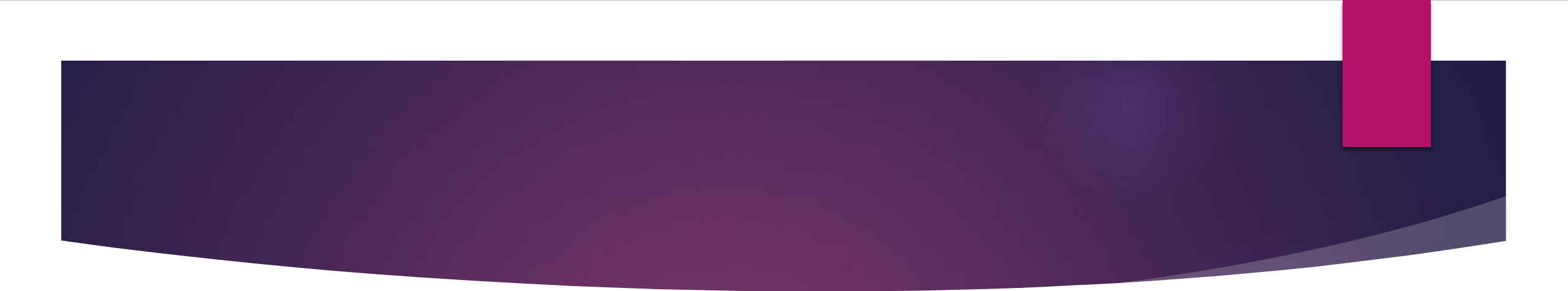
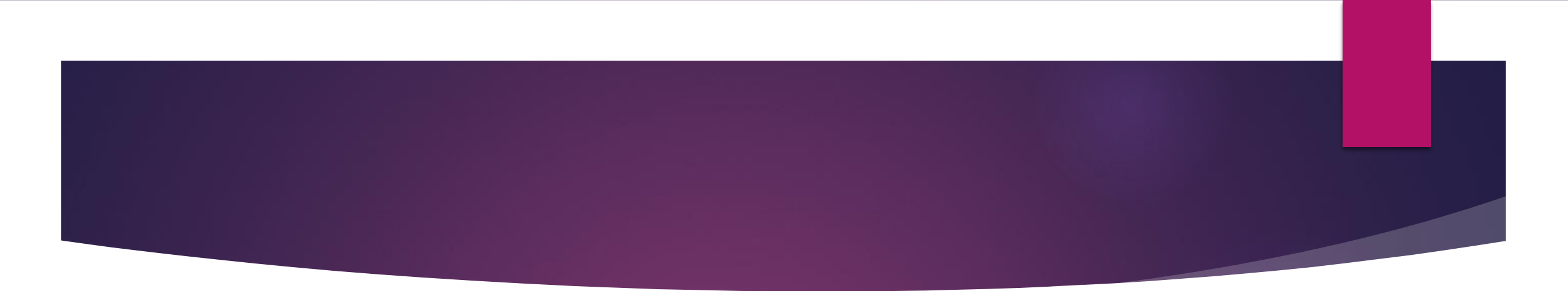


Creación de un sistema de seguimiento de problemas mediante formularios reactivos

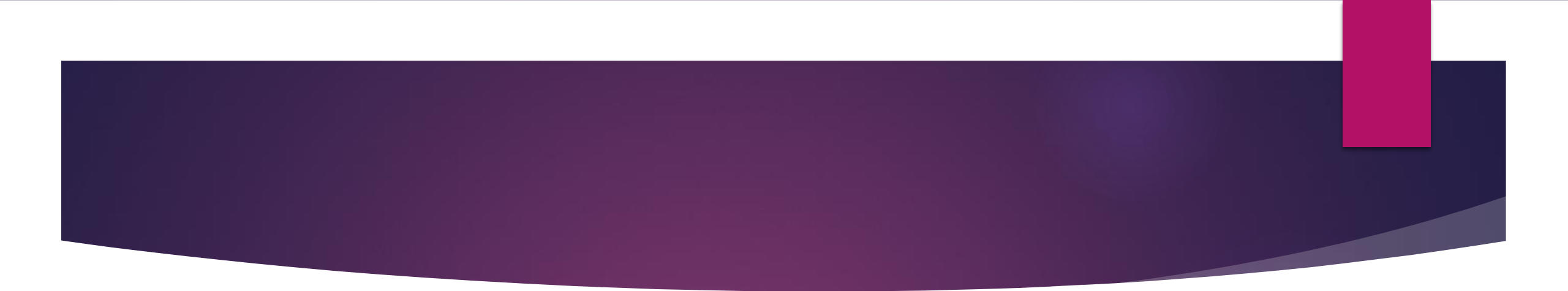
UNIDAD 3

- 
- ▶ Las aplicaciones web utilizan formularios HTML para recopilar datos de los usuarios y validarlos, como al iniciar sesión en una aplicación, realizar una búsqueda o completar un pago en línea.
 - ▶ Angular proporciona dos tipos de formularios, reactivos y basados en plantillas, que podemos usar en una aplicación Angular.
 - ▶ Crearemos un sistema para administrar y rastrear problemas. Usaremos formularios reactivos de Angular para informar nuevos problemas. También usaremos Clarity Design System de VMware para diseñar nuestros formularios y mostrar nuestros problemas.

- 
- ▶ Cubriremos los siguientes temas:
 - ▶ Instalación de Clarity Design System en una aplicación Angular
 - ▶ Visualización de una descripción general de los problemas
 - ▶ Informar nuevos problemas
 - ▶ Marcar un problema como resuelto
 - ▶ Activar sugerencias para nuevos problemas

Teoría y contexto esenciales

- ▶ Angular proporciona dos tipos de formas que podemos usar:
 - ▶ Basado en plantillas(Template-driven): son fáciles y sencillos de configurar en una aplicación Angular. Los formularios basados en plantillas no se escalan bien y son difíciles de probar porque están definidos en la plantilla del componente.
 - ▶ Reactivo(Reactive): Se basan en un enfoque de programación reactiva. Los formularios reactivos operan en la clase TypeScript del componente y son más fáciles de probar y escalar mejor que los formularios basados en plantillas.

- 
- ▶ Ahora nos familiarizaremos con el enfoque de formas reactivas, que es el más popular en la comunidad Angular.
 - ▶ Los componentes angular pueden obtener datos de fuentes externas como HTTP o de otros componentes. En el último caso, interactúan con componentes que tienen datos utilizando una API pública:
 - ▶ @Input (): se utiliza para pasar datos a un componente.
 - ▶ @Output (): se utiliza para recibir notificaciones sobre cambios o recuperar datos de un componente.
 - ▶ Clarity es un sistema de diseño que contiene un conjunto de pautas de UX y UI para crear aplicaciones web. También consta de un marco de HTML y CSS patentado que incluye estas pautas. Afortunadamente, no tenemos que usar este marco ya que Clarity ya proporciona una amplia variedad de componentes de IU basados en Angular que podemos usar en nuestras aplicaciones.

Descripción del proyecto

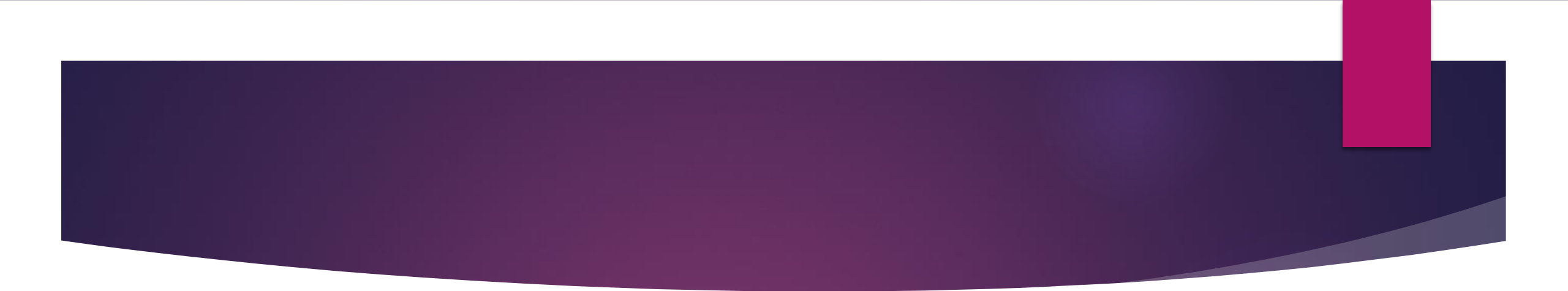
- ▶ En este proyecto, crearemos una aplicación Angular para administrar y rastrear problemas usando formularios reactivos y Clarity.
- ▶ Inicialmente, mostraremos una lista de problemas en una tabla que podemos ordenar y filtrar.
- ▶ Luego crearemos un formulario para permitir a los usuarios informar nuevos problemas.
- ▶ Finalmente, crearemos un diálogo modal para resolver un problema.
- ▶ También haremos un esfuerzo adicional y activaremos las sugerencias al informar un problema para ayudar a los usuarios a evitar entradas duplicadas.
- ▶ Tiempo estimado: 1 hora.

Instalar Clarity en una aplicación angular

- ▶ Comencemos a crear nuestro sistema de seguimiento de problemas con una nueva aplicación Angular.

```
PS C:\Users\Geovany\Desktop\Angular> ng new issue-tracker --defaults
```

- ▶ Usamos el comando `ng new` de Angular CLI para crear una nueva aplicación Angular con las siguientes características:
 - ▶ `issue-tracker`: el nombre de la aplicación Angular.
 - ▶ `--defaults`: esto deshabilita el enrutamiento angular para la aplicación y establece que el formato de la hoja de estilo sea CSS

- 
- ▶ Ahora necesitamos instalar la biblioteca Clarity en nuestra aplicación Angular. El equipo de VMware ha creado un esquema de Angular CLI que podemos usar. Navegue a la carpeta del issue tracker que se creó con el comando anterior y ejecute el siguiente comando add de Angular CLI:

```
PS C:\Users\Geovany\Desktop\Angular\issue-tracker> ng add @clr/angular
```

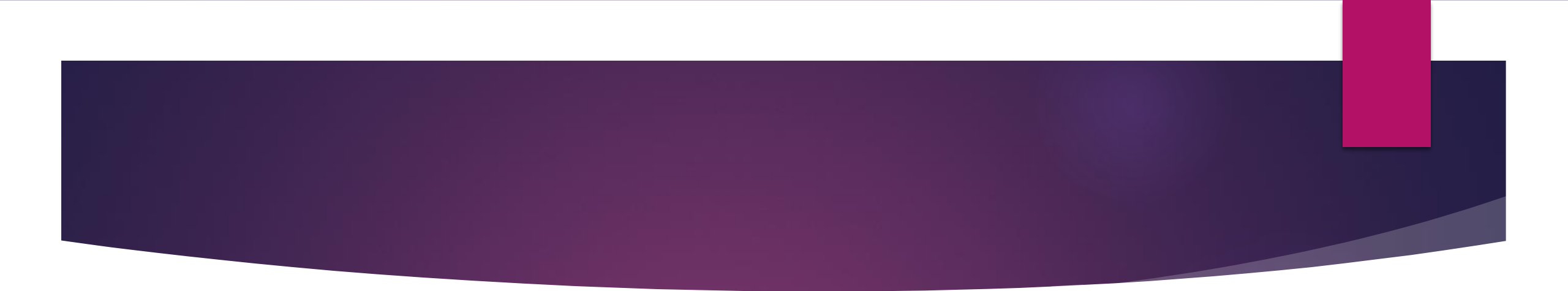

- El comando anterior realizará las siguientes modificaciones en nuestro espacio de trabajo de Angular CLI:
 1. Agregará todos los paquetes npm necesarios de la biblioteca Clarity a la sección de dependencias del archivo package.json.
 2. Agregará los estilos necesarios en el archivo de configuración del espacio de trabajo de Angular CLI, angular.json:

```
29  
30 |  
31 |  
32 |  
33 |  
34 |  
    "styles": [  
      "node_modules/@clr/ui/clr-ui.min.css",  
      "src/styles.css"  
    ],  
    "scripts": []  
  },  
}
```

- Finalmente, importará ClarityModule en el módulo principal de la aplicación, app.module.ts:

```
1  import { NgModule } from '@angular/core';  
2  import { BrowserModule } from '@angular/platform-browser';  
3  
4  import { AppComponent } from './app.component';  
5  import { ClarityModule } from '@clr/angular';  
6  import { BrowserAnimationsModule } from '@angular/platform-browser/animations';  
7  
8  @NgModule({  
9    declarations: [  
10     | AppComponent  
11    | ],  
12    imports: [  
13     | BrowserModule,  
14     | ClarityModule,  
15     | BrowserAnimationsModule  
16    | ],  
17    providers: [],  
18    bootstrap: [AppComponent]  
19  })  
20  export class AppModule { }
```

You, a day ago • initial commit

- 
- ▶ También importará `BrowserAnimationsModule` desde el paquete npm `@angular/platform-browser/animations`.
 - ▶ `BrowserAnimationsModule` se utiliza para mostrar animaciones cuando ocurren acciones específicas en nuestra aplicación, como hacer clic en un botón.
 - ▶ Ahora que hemos completado la instalación de Clarity en nuestra aplicación, podemos comenzar a crear diseños con él. Ahora vamos a crear una lista para mostrar nuestros problemas.

Visualización de una descripción general de los problemas

- ▶ Nuestra aplicación Angular será responsable de administrar y rastrear problemas. Cuando se inicia la aplicación, deberíamos mostrar una lista de todos los problemas pendientes en el sistema. Las cuestiones pendientes se definen como aquellas cuestiones que no se han resuelto. El proceso que seguiremos se puede analizar más a fondo en lo siguiente:
 - ▶ Obteniendo problemas pendientes
 - ▶ Visualización de problemas usando un grid

Obteniendo problemas pendientes

- ▶ Primero, necesitamos crear un mecanismo para buscar todos los problemas pendientes:
 1. Use el comando generate de Angular CLI para crear un servicio Angular llamado issues:


```
\issue-tracker> ng generate service issues
```

El comando anterior creará un archivo issues.service.ts en la carpeta src \ app de nuestro proyecto CLI de Angular

- Cada problema tendrá propiedades específicas de un tipo definido. Necesitamos crear una interfaz TypeScript para eso con el siguiente comando CLI angular:

```
issue-tracker> ng generate interface issue
```

- Abra el archivo issue.ts y agregue las siguientes propiedades en la interfaz de Issue:

```
src > app >  issue.ts > ...  
1  export interface Issue {  
2      issueNo: number;  
3      title: string;  
4      description: string;  
5      priority: 'low' | 'high';  
6      type: 'Feature' | 'Bug' | 'Documentation';  
7      completed?: Date;  
8  }
```

- Abra el servicio Angular que creamos y agregue una propiedad de problemas para contener nuestros datos de problemas. También cree un método `getPendingIssues` que devolverá todos los problemas que no se hayan completado:

```
src > app > 📁 issues.service.ts > ...
1  import { Injectable } from '@angular/core';
2  import { Issue } from './issue';
3
4  @Injectable({
5    providedIn: 'root',
6  })
7  export class IssuesService {
8    private issues: Issue[] = [];
9    constructor() {}
10   getPendingIssues(): Issue[] {
11     return this.issues.filter((issue) => !issue.completed);
12   }
13 }
```

- Inicializamos la propiedad issues en una matriz vacía. Si desea comenzar con datos de muestra, puede usar el archivo mock-issues.ts e importarlo de la siguiente manera:

```
src > app > A issues.service.ts > IssuesService
1  import { Injectable } from '@angular/core';
2  import { Issue } from './issue';
3  import { issues } from '../assets/mock-issues'
4
```


- Crearemos un componente para mostrar esos problemas.

Visualización de problemas en un grid

- ▶ Vamos a utilizar el componente UI Grid de la biblioteca Clarity para mostrar los datos en formato tabular. Un grid también proporciona mecanismos para filtrar y clasificar desde el primer momento. Primero creemos el componente Angular que albergará el grid.
- ▶ Use el comando generate de Angular CLI para crear el componente

```
issue-tracker> ng generate component issue-list
```

- Abra la plantilla del componente principal de nuestra aplicación, `app.component.html` y reemplace su contenido con el siguiente código HTML:

```
src > app >  app.component.html > ...  
Go to component | You, seconds ago | 1 author (You)  
1 <div class="main-container">  
2   <div class="content-container">  
3     <div class="content-area">  
4       <app-issue-list></app-issue-list>  
5     </div>  
6   </div>  
7 </div>
```

- La lista de problemas se mostrará en el componente principal de la aplicación Angular, tan pronto como se inicie

- Actualmente, el componente app-issue-list no muestra ningún dato de problemas. Necesitamos conectarlo con el servicio Angular que creamos. Abra el archivo issue-list.component.ts e inyecte IssuesService en el constructor de la clase IssueListComponent:

```
src > app > issue-list >  issue-list.component.ts > ...  
1  import { Component, OnInit } from '@angular/core';  
2  import { IssuesService } from '../issues.service';  
3    
4  @Component({  
5    selector: 'app-issue-list',  
6    templateUrl: './issue-list.component.html',  
7    styleUrls: ['./issue-list.component.css']  
8  })  
9  export class IssueListComponent implements OnInit {  
10  
11    constructor(private issueService: IssuesService) { }  
12  
13    ngOnInit(): void {  
14    }  
15  
16  }
```


- Cree un método llamado `getIssues` que llamará al método `getPendingIssues` del servicio inyectado y mantendrá su valor devuelto en la propiedad del componente de problemas:

```
src > app > issue-list > issue-list.component.ts > IssueListComponent
1  import { Component, OnInit } from '@angular/core';
2  import { IssuesService } from '../issues.service';
3  import { Issue } from '../issue';
4  @Component({
5    selector: 'app-issue-list',
6    templateUrl: './issue-list.component.html',
7    styleUrls: ['./issue-list.component.css'],
8  })
9  export class IssueListComponent implements OnInit {
10     issues: Issue[] = [];
11     constructor(private issueService: IssuesService) {}
12     private getIssues() {
13       this.issues = this.issueService.getPendingIssues();
14     }
15     ngOnInit(): void {}
16   }
```

- ▶ Finalmente, llame al método `getIssues` en el método del componente `ngOnInit` para obtener todos los problemas pendientes en la inicialización del componente.

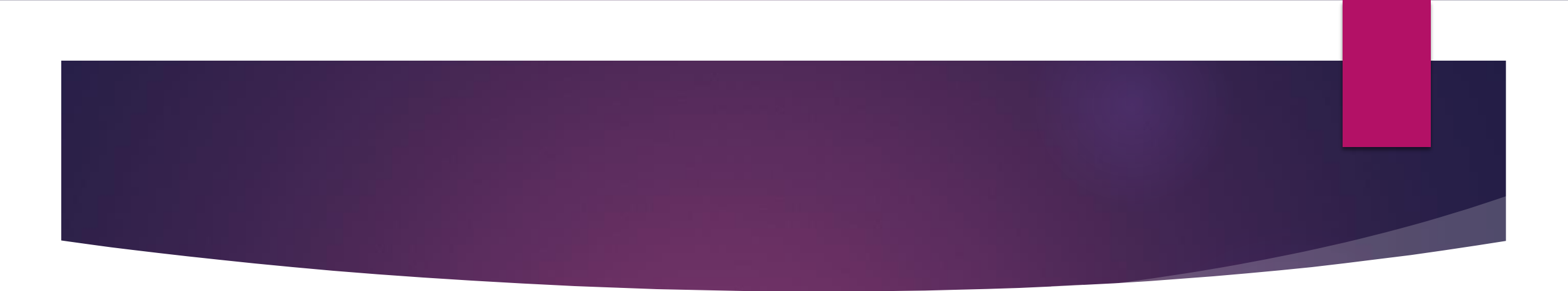
```
14     }  
15     ngOnInit(): void {  
16         this.getIssues();  
17     }
```

- ▶ Ya hemos implementado el proceso para obtener datos de problemas en nuestro componente. Todo lo que tenemos que hacer ahora es mostrarlo en la plantilla. Abra el archivo `issue-list.component.html` y reemplace su contenido con el siguiente código HTML

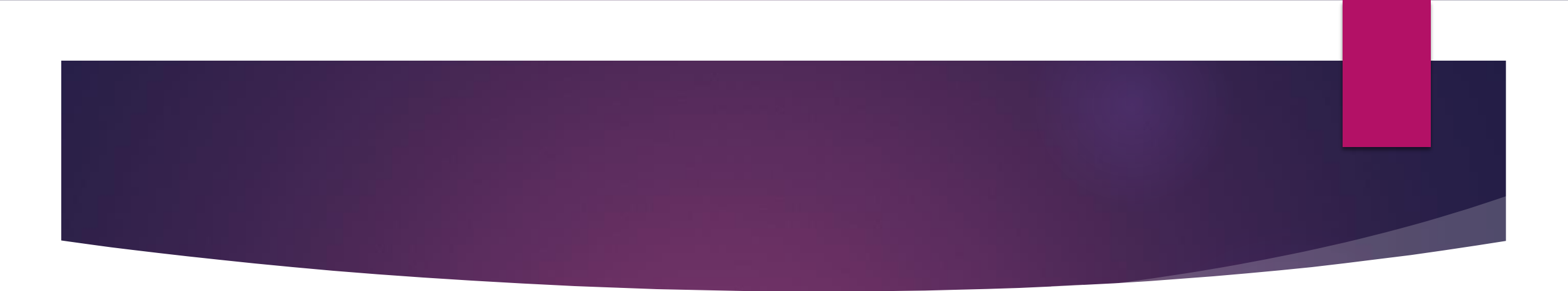
src > app > issue-list >  issue-list.component.html > ...

Go to component

```
1  <clr-datagrid>
2    <clr-dg-column [clrDgField]='issueNo' [clrDgColType]='number'>Issue No</clr-dg-column>
3    <clr-dg-column [clrDgField]='type'>Type</clr-dg-column>
4    <clr-dg-column [clrDgField]='title'>Title</clr-dg-column>
5    <clr-dg-column [clrDgField]='description'>Description
6    </clr-dg-column>
7    <clr-dg-column [clrDgField]='priority'>Priority
8    </clr-dg-column>
9    <clr-dg-row *clrDgItems='let issue of issues'>
10     <clr-dg-cell>{{issue.issueNo}}</clr-dg-cell>
11     <clr-dg-cell>{{issue.type}}</clr-dg-cell>
12     <clr-dg-cell>{{issue.title}}</clr-dg-cell>
13     <clr-dg-cell>{{issue.description}}</clr-dg-cell>
14     <clr-dg-cell>
15       <span class='label' [class.label-danger]='issue.priority === 'high''>{{issue.priority}}</span>
16     </clr-dg-cell>
17   </clr-dg-row>
18   <clr-dg-footer>{{issues.length}} issues</clr-dg-footer>
19 </clr-datagrid>
```

- 
- ▶ En el fragmento anterior, usamos varios componentes de la biblioteca Clarity:
 - ▶ `clr-datagrid`: define una tabla.
 - ▶ `clr-dg-column`: define una columna de una tabla. Cada columna usa la directiva `clrDgField` para enlazar con el nombre de propiedad del problema representado por esa columna. La directiva `clrDgField` nos proporciona capacidades de clasificación y filtrado sin escribir ni una sola línea de código en el archivo de clase de TypeScript. La clasificación funciona automáticamente solo con contenido basado en cadenas. Si queremos ordenar por un tipo primitivo diferente, debemos usar la directiva `clrDgColType` y especificar el tipo particular.
 - ▶ `clr-dg-row`: define una fila de una tabla. Utiliza la directiva `clrDgItems` para iterar sobre los problemas y crear una fila para cada problema.
 - ▶ `clr-dg-cell`: cada fila contiene una colección de componentes `clr-dg-cell` para mostrar el valor de cada columna mediante interpolación. En la última celda, agregamos la clase `label-danger` cuando un problema tiene una prioridad alta para indicar su importancia.
 - ▶ `clr-dg-footer`: define el pie de página de una tabla. En este caso, muestra el número total de problemas.

Issue No	Type	Title	Description	Priority
1	Feature	Add email validation in registration form	Validate the email entered in the user registration form	high
2	Feature	Display the adress details of a customer	Add a column to display the details of the customer address in the customer list	low
3	Bug	Export to CSV is not working	The export process of a report into CSV format throws an error	high
4	Feature	Locale settings per user	Add settings configure the locale of the current user	low
5	Documentation	Add new customer tutorial	Create a tutorial on how to add a new customer into the application	high
				5 issues

- 
- ▶ El componente grid de la biblioteca Clarity tiene un amplio conjunto de capacidades que podemos usar en nuestras aplicaciones.
 - ▶ Ahora vamos a aprender cómo usar formularios reactivos para informar un nuevo problema.

Informar nuevos problemas

- ▶ Una de las principales características de nuestro sistema de seguimiento de problemas es la capacidad de informar nuevos problemas.
- ▶ Usaremos formularios reactivos para crear un formulario para agregar nuevos problemas. La función se puede subdividir en las siguientes tareas:
 - ▶ Configurar formas reactivas en una aplicación Angular
 - ▶ Creación del formulario de emisión de informes
 - ▶ Visualización de un nuevo problema en la lista
 - ▶ Validar los detalles de un problema
- ▶ Comencemos por introducir formas reactivas en nuestra aplicación Angular.

Configurar formas reactivas en una aplicación angular

- ▶ Las formas reactivas se definen en el paquete npm @angular/forms.
- ▶ Abra el archivo app.module.ts e importe ReactiveFormsModule:

```
import { ReactiveFormsModule } from '@angular/forms';
```

- ▶ Agregue ReactiveFormsModule en la matriz de importaciones del decorador @NgModule:

```
14 imports: [  
15   BrowserModule,  
16   ClarityModule,  
17   BrowserAnimationsModule,  
18   ReactiveFormsModule  
19 ]
```


Creación del formulario de registro de problemas

- ▶ Ahora que hemos introducido formas reactivas en nuestra aplicación Angular, podemos empezar a construir nuestro formulario:

1. Cree un nuevo componente de Angular llamado issue-report:

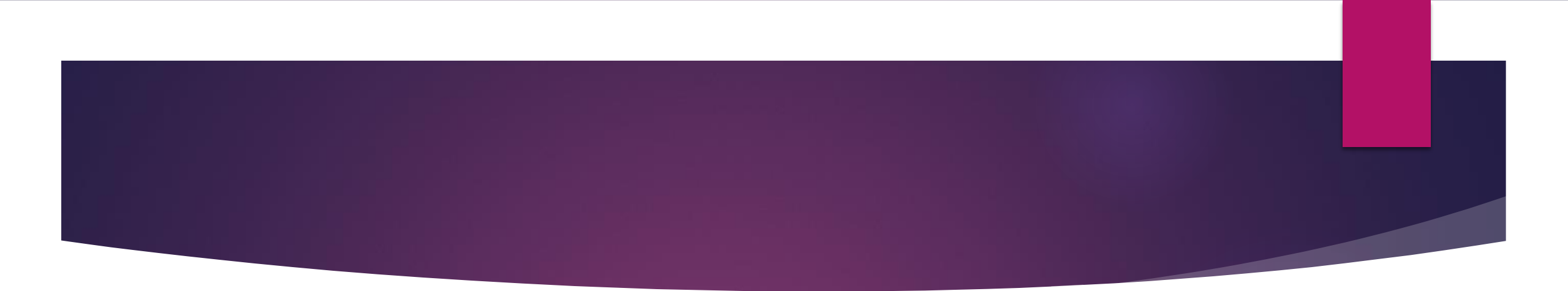
```
issue-tracker> ng generate component issue-report
```

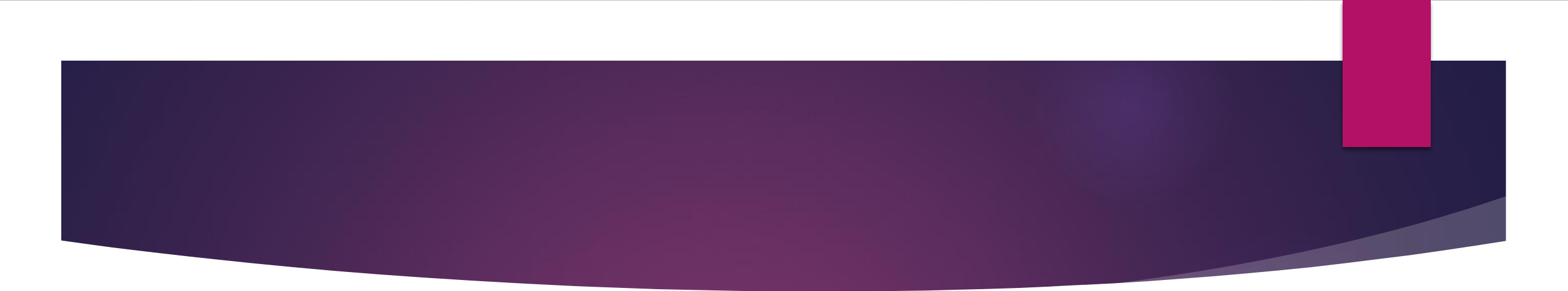
2. Abra el archivo issue-report.component.ts e inyecte FormBuilder en el constructor de la clase IssueReportComponent:


```
src > app > issue-report >  issue-report.component.ts > ...
 1  import { Component, OnInit } from '@angular/core';
 2  import { FormBuilder } from '@angular/forms';
 3  import { FormGroup } from '@angular/forms';
 4  @Component({
 5    selector: 'app-issue-report',
 6    templateUrl: './issue-report.component.html',
 7    styleUrls: ['./issue-report.component.css']
 8  })
 9  export class IssueReportComponent implements OnInit {
10
11    constructor(private builder: FormBuilder) { }
12
13    ngOnInit(): void {
14    }
15
16  }
```

- Declare una propiedad `issueForm` del tipo `FormGroup` e inicialízela dentro del método `ngOnInit`:

```
src > app > issue-report > issue-report.component.ts > IssueReportComponent > ngOnInit
1  import { Component, OnInit } from '@angular/core';
2  import { FormBuilder, FormGroup } from '@angular/forms';
3
4  @Component({
5    selector: 'app-issue-report',
6    templateUrl: './issue-report.component.html',
7    styleUrls: ['./issue-report.component.css']
8  })
9  export class IssueReportComponent implements OnInit {
10     issueForm: FormGroup | undefined
11     constructor(private builder: FormBuilder) { }
12
13     ngOnInit(): void {
14         this.issueForm = this.builder.group({
15             title: [''],
16             description: [''],
17             priority: [''],
18             type: ['']
19         });
20     }
21 }
```

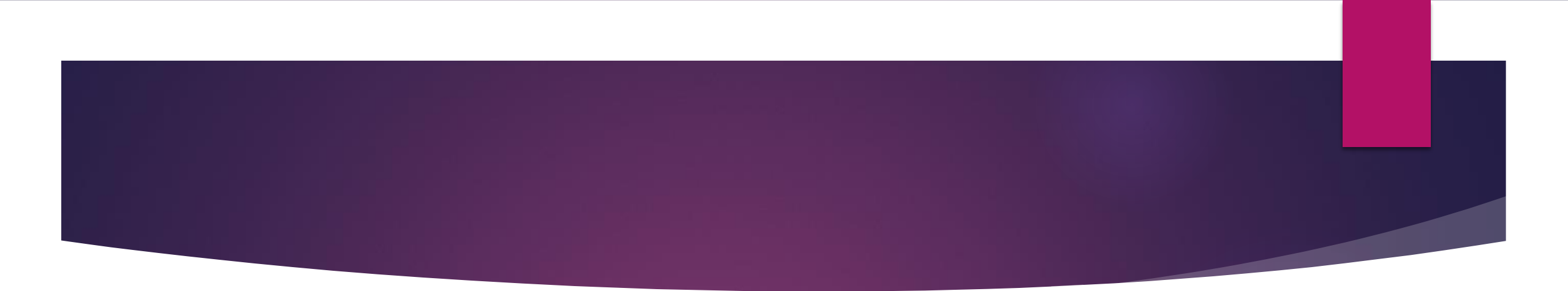
- 
- ▶ FormGroup se utiliza para agrupar controles individuales en una representación lógica de un formulario. El método de grupo de la clase FormBuilder se usa para construir el formulario.
 - ▶ Acepta un objeto como parámetro donde cada clave es el nombre único de un control de formulario y cada valor una matriz que contiene su valor predeterminado. En este caso, inicializamos todos los controles en cadenas vacías porque el formulario se utilizará para crear un nuevo problema desde cero.

- 
- ▶ Ahora necesitamos asociar el objeto FormGroup que creamos con los respectivos elementos HTML. Abra el archivo `issue-report.component.html` y reemplace su contenido con el siguiente código HTML


src > app > issue-report >  issue-report.component.html >

Go to component

```
1  <h3>Report an issue</h3>
2  <form clrForm *ngIf="issueForm" [formGroup]="issueForm">
3    <clr-input-container>
4      <label>Title</label>
5      <input clrInput formControlName="title" />
6    </clr-input-container>
7    <clr-textarea-container>
8      <label>Description</label>
9      <textarea clrTextarea formControlName="description"></textarea>
10   </clr-textarea-container>
11   <clr-radio-container clrInline>
12     <label>Priority</label>
13     <clr-radio-wrapper>
14       <input type="radio" value="low" clrRadio formControlName="priority" />
15       <label>Low</label>
16     </clr-radio-wrapper>
17     <clr-radio-wrapper>
18       <input type="radio" value="high" clrRadio formControlName="priority" />
19       <label>High</label>
20     </clr-radio-wrapper>
21   </clr-radio-container>
22   <clr-select-container>
23     <label>Type</label>
24     <select clrSelect formControlName="type">
25       <option value="Feature">Feature</option>
26       <option value="Bug">Bug</option>
27       <option value="Documentation">Documentation
28       </option>
29     </select>
30   </clr-select-container>
31 </form>
```

- 
- ▶ Las directivas `formGroup` y `clrForm` se utilizan para asociar el elemento del formulario HTML con la propiedad `issueForm` e identificarlo como un formulario Clarity.
 - ▶ La directiva `formControlName` se usa para asociar elementos HTML con controles de formulario usando su nombre. Cada control también se define mediante un elemento contenedor Clarity. Por ejemplo, el control de entrada del título es un componente `clr-input-container` que contiene un elemento HTML de entrada.
 - ▶ Cada elemento HTML nativo tiene una directiva Clarity adjunta según su tipo. Por ejemplo, el elemento HTML de entrada contiene una directiva `clrInput`.

- Finalmente, agregue algunos estilos a nuestro archivo issue-report.component.css:

```
src > app > issue-report >  issue-report.component.css > ...  
1  .clr-input, .clr-textarea {  
2    width: 30%;  
3  }  
4  button {  
5    margin-top: 25px;  
6  }  
7
```

- ▶ Ahora que hemos creado los conceptos básicos de nuestro formulario, aprenderemos cómo enviar sus detalles:
- ▶ Agregue un elemento de botón HTML antes de la etiqueta de cierre del elemento de formulario HTML:

```
31     <button class="btn btn-primary" type="submit">  
32         Create</button>  
33 </form>
```

- ▶ Configuramos su tipo para enviar para activar el envío del formulario al hacer clic en el botón.

- Abra el archivo `issues.service.ts` y agregue un método `createIssue` que inserte un nuevo problema en la matriz de problemas:

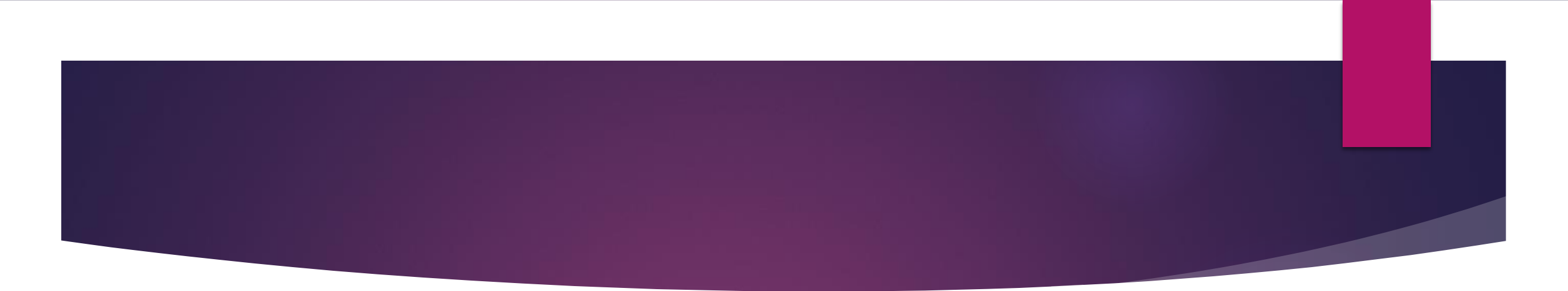
```
14  createIssue(issue: Issue) {  
15      issue.issueNo = this.issues.length + 1;  
16      this.issues.push(issue);  
17  }
```

- Vuelva al archivo `issue-report.component.ts`, importe `IssuesService`, e inyectelo al constructor de la clase TypeScript

```
constructor(private builder: FormBuilder, private issueService: IssuesService) { }
```

- Agregue un nuevo método de componente que llamará al método `createIssue` del servicio inyectado:

```
addIssue() {  
  this.issueService.createIssue(this.issueForm?.value);  
}
```

- 
- ▶ Pasamos el valor de cada control de formulario utilizando la propiedad value del objeto issueForm.
 - ▶ Nota IMPORTANTE
 - ▶ La propiedad de valor de un objeto FormGroup contiene el modelo del formulario.
 - ▶ Las claves del modelo coinciden con los nombres de propiedad de la interfaz Issue, que es el tipo que el método createIssue acepta como parámetro. Si fueran diferentes, deberíamos convertir el modelo de formulario antes de pasarlo al método.

- ▶ Abra el archivo `issue-report.component.html` y vincule el evento `ngSubmit` del formulario al método del componente `addIssue`:

```
<form clrForm *ngIf="issueForm" [formGroup]="issueForm" (ngSubmit)="addIssue()">
```

- ▶ El evento `ngSubmit` se activará cuando hagamos clic en el botón Crear del formulario.
- ▶ Ahora hemos completado todos los procesos involucrados para agregar un nuevo problema al sistema.

Visualización de un nuevo problema en la lista

- ▶ Mostrar problemas y crear nuevos son dos tareas delegadas a diferentes componentes de Angular. Cuando creamos un nuevo problema con `IssueReportComponent`, debemos notificar a `IssueListComponent` para reflejar ese cambio en la tabla. Primero, veamos cómo podemos configurar `IssueReportComponent` para comunicar ese cambio:
- ▶ Abra el archivo `issue-report.component.ts` y use el decorador `@Output()` para agregar una propiedad `EventEmitter`:

```
export class IssueReportComponent implements OnInit {  
  issueForm: FormGroup | undefined  
  @Output() formClose = new EventEmitter();  
}
```

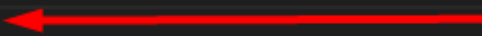
- Llame al método emit de la propiedad de salida formClose dentro del método del componente addIssue, justo después de crear el problema:

```
addIssue() {  
  this.issueService.createIssue(this.issueForm?.value);  
  this.formClose.emit();  
}
```

- Agregue un segundo elemento de botón HTML en la plantilla del componente y llame al método emit en su evento de clic:

```
31   <button class="btn btn-primary" type="submit">  
32     Create</button>  
33   <button class="btn" type="button" (click)="formClose.emit()">Cancel</button>  
34 </form>
```

- ▶ IssueListComponent ahora puede vincularse al evento formClose de issueReportComponent y recibir una notificación cuando se hace clic en cualquiera de los botones.
- ▶ Averigüemos cómo:
- ▶ Abra el archivo issue-list.component.ts y agregue la siguiente propiedad en la clase IssueListComponent:

```
9  export class IssueListComponent implements OnInit {  
10    issues: Issue[] = [];  
11    showReportIssue = false;   
12    constructor(private issueService: IssuesService) {}  
13    private getIssues() {  
14      this.issues = this.issueService.getPendingIssues();  
15    }  
}
```

- Agregue el siguiente método de componente que se llamará cuando el formulario de emisión del informe emita el evento formClose:

```
onCloseReport() {  
  this.showReportIssue = false;  
  this.getIssues();  
}
```

- El método onCloseReport establecerá la propiedad showReportIssue en false para que el formulario de informe de problemas ya no sea visible y, en su lugar, se muestre la tabla de problemas pendientes. También recuperará problemas nuevamente para actualizar los datos en la tabla.

- Abra el archivo `issue-list.component.html` y agregue un elemento de botón HTML en la parte superior de la plantilla. El botón mostrará el formulario de emisión del informe cuando se haga clic en él:

```
src > app > issue-list > issue-list.component.html > button.btn.btn-primary  
  
1 <button class="btn btn-primary" (click)=  
2   "showReportIssue = true">Add new issue</button
```

- Agrupe el botón y la cuadrícula de datos dentro de un elemento `ng-container`. Como lo indica la directiva `*ngIf`, el contenido del elemento `ng-container` se mostrará cuando el formulario de emisión del informe no esté visible:

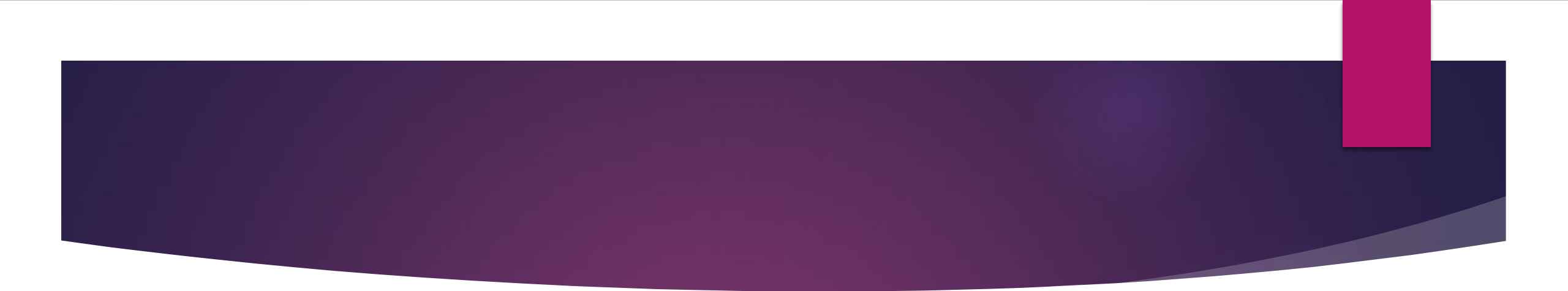
src > app > issue-list >  issue-list.component.html >  ng-container

Go to component

```
1 <ng-container *ngIf="showReportIssue === false">
2   <button class="btn btn-primary" (click)=
3     "showReportIssue = true">Add new issue</button>
4 <clr-datagrid>
5   <clr-dg-column [clrDgField]="issueNo" [clrDgColType]='number'>Issue No</clr-dg-column>
6   <clr-dg-column [clrDgField]='type'>Type</clr-dg-column>
7   <clr-dg-column [clrDgField]='title'>Title</clr-dg-column>
8   <clr-dg-column [clrDgField]='description'>Description
9   </clr-dg-column>
10  <clr-dg-column [clrDgField]='priority'>Priority
11  </clr-dg-column>
12  <clr-dg-row *clrDgItems="let issue of issues">
13    <clr-dg-cell>{{issue.issueNo}}</clr-dg-cell>
14    <clr-dg-cell>{{issue.type}}</clr-dg-cell>
15    <clr-dg-cell>{{issue.title}}</clr-dg-cell>
16    <clr-dg-cell>{{issue.description}}</clr-dg-cell>
17    <clr-dg-cell>
18      <span class="label" [class.label-danger]="issue.priority === 'high'">{{issue.priority}}</span>
19    </clr-dg-cell>
20  </clr-dg-row>
21  <clr-dg-footer>{{issues.length}} issues</clr-dg-footer>
22 </clr-datagrid>
23 </ng-container>
```

- ▶ El elemento ng-container es un componente angular que no se representa en la pantalla y se usa para agrupar elementos HTML.
- ▶ Agregue el componente app-issue-report al final de la plantilla y use la directiva *ngIf para mostrarlo cuando la propiedad showReportIssue sea verdadera. Vincular también su evento formClose al método del componente onCloseReport

```
24 <app-issue-report *ngIf="showReportIssue == true"  
25 | (formClose)="onCloseReport()"></app-issue-report>  
26  
27
```


- 
- ▶ Hemos conectado con éxito todos los puntos y completado la interacción entre el formulario de informe de problemas y la tabla que muestra los problemas. Ahora es el momento de ponerlos en acción:

1. Ejecute la aplicación Angular usando ng serve.
2. Haga clic en el botón ADD NEW ISSUE e introduzca los detalles de un nuevo problema:



localhost:4200

Report an issue

Title

Error cuando intento pasar a un estudiante

Description

al momento de capturar una calificación si pongo 6 o mas me marca un error.

Priority

☒ Low ☐ High

Type

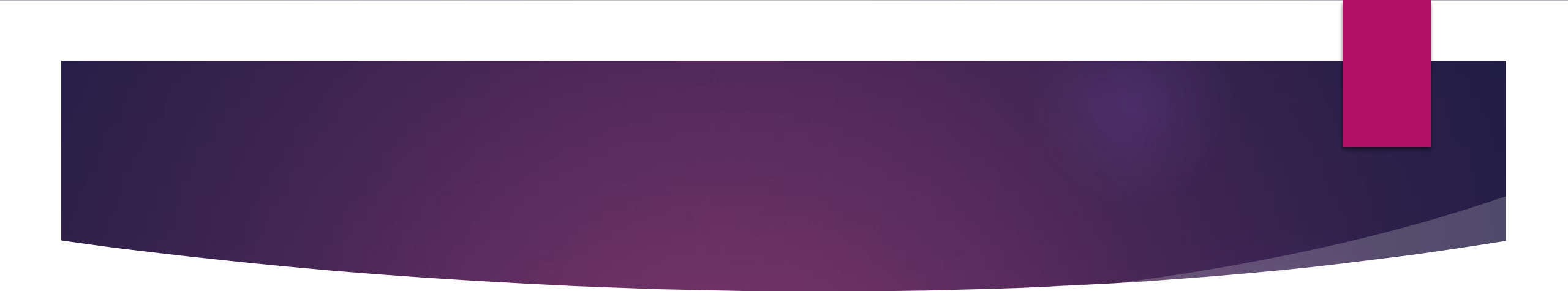
Bug

CREATE

CANCEL

- Haga clic en el botón CREATE y el nuevo problema debería aparecer en la tabla.

</

- 
- ▶ Intente agregar un nuevo problema sin ningún detalle y notará que se agrega un número vacío a la tabla.
 - ▶ Se puede crear un problema vacío porque aún no hemos definido ningún campo obligatorio en nuestro formulario de informe de problema.
 - ▶ Ahora veremos cómo realizar esta tarea y agregaremos validaciones a nuestro formulario para evitar comportamientos inesperados.

Validar los detalles de un problema

- ▶ Cuando creamos un problema con el formulario, podemos dejar el valor de un control de formulario vacío ya que aún no hemos agregado ninguna regla de validación. Para agregar validaciones en un control de formulario, usamos la clase `Validators` del paquete `@angular/forms`.
- ▶ Se agrega un validador en cada instancia de control de formulario que construimos usando el servicio `FormBuilder`. En este caso, usaremos el validador requerido para indicar que se requiere que un control de formulario tenga un valor:

- Abra el archivo `issue-report.component.ts` e importe `Validators` del paquete `@angular/forms`:





```
1 import { EventEmitter, Output } from '@angular/core';
2 import { Component, OnInit } from '@angular/core';
3 import { FormBuilder, FormGroup, Validators } from '@angular/forms';
4 import { IssuesService } from '../issues.service';
5
6 @Component({
7   selector: 'app-issue-report'
```

- Establezca la propiedad estática `Validators.required` en todos los controles excepto en la descripción del problema:

```
ngOnInit(): void {  
  this.issueForm = this.builder.group({  
    title: ['', Validators.required],  
    description: [''],  
    priority: ['', Validators.required],  
    type: ['', Validators.required],  
  });  
}
```


- Podemos usar varios validadores para un control de formulario, como `min`, `max` y `email`. Si queremos establecer varios validadores en un control de formulario, los agregamos dentro de una matriz.

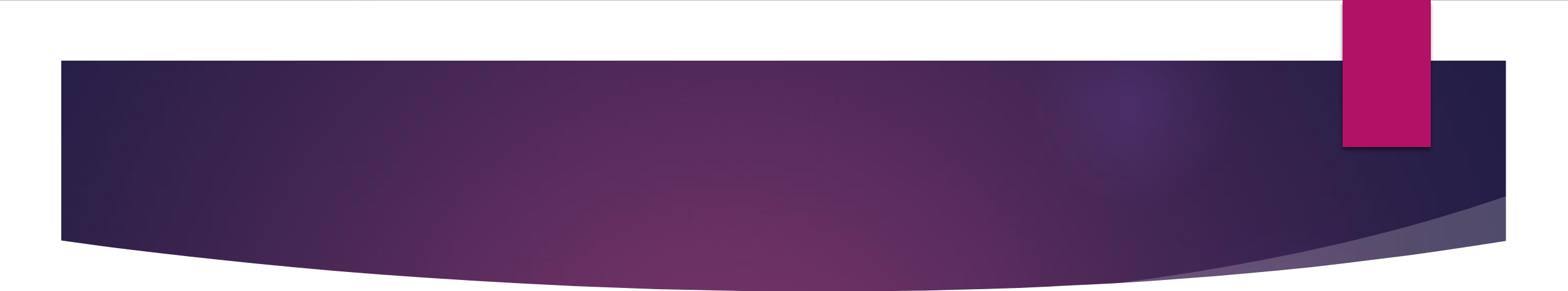
► Cuando usamos validadores en un formulario, necesitamos proporcionar una indicación visual al usuario del formulario. Abra el archivo `issue-report.component.html` y agregue un componente `clr-control-error` para cada control de formulario requerido

```
src > app > issue-report >  issue-report.component.html >  form >  clr-select-container >  clr-control-error
```

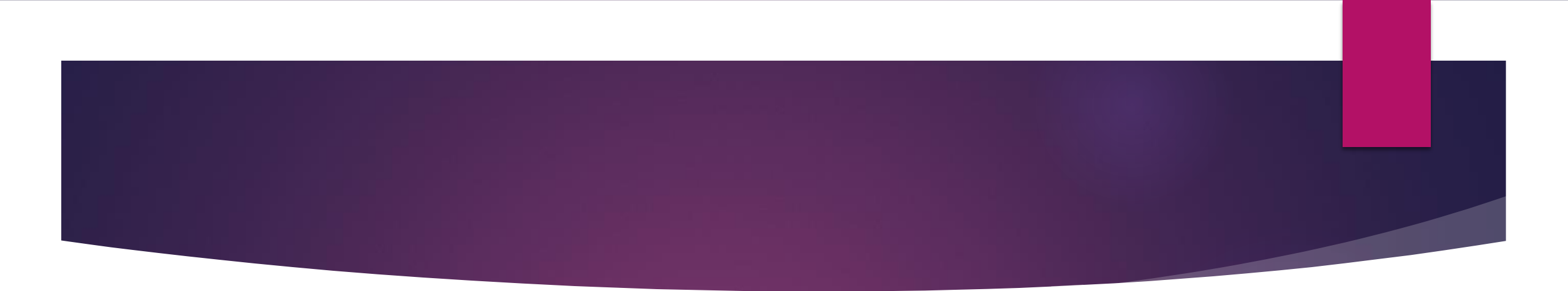
Go to component

```
1 <h3>Report an issue</h3>
2 <form clrForm *ngIf="issueForm" [formGroup]="issueForm" (ngSubmit)="addIssue()">
3   <clr-input-container>
4     <label>Title</label>
5     <input clrInput formControlName="title" />
6     <clr-control-error>Title is required</clr-control-error>
7   </clr-input-container>
8   <clr-textarea-container>
9     <label>Description</label>
10    <textarea clrTextarea formControlName="description"></textarea>
11  </clr-textarea-container>
12  <clr-radio-container clrInline>
13    <label>Priority</label>
14    <clr-radio-wrapper>
15      <input type="radio" value="low" clrRadio formControlName="priority" />
```

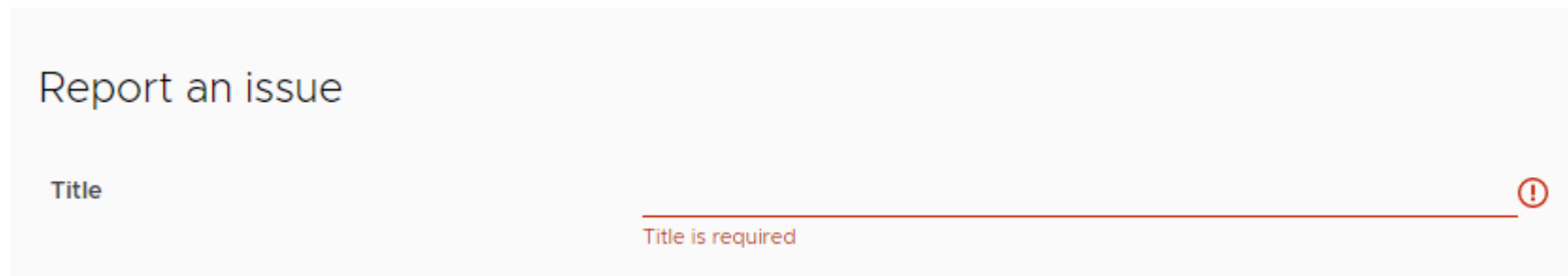


- 
- ▶ `clr-control-error` es un componente de Clarity que se utiliza para proporcionar mensajes de validación en formularios. Se muestra cuando tocamos un control que no es válido. Un control no es válido cuando se infringe al menos una de sus reglas de validación.
 - ▶ Es posible que el usuario no siempre toque los controles de formulario para ver el mensaje de validación. Por lo tanto, debemos tener eso en cuenta al enviar el formulario y actuar en consecuencia. Para superar este caso, marcaremos todos los controles de formulario como `touched` cuando se envíe el formulario:

```
addIssue() {  
  if (this.issueForm && this.issueForm.invalid) {  
    this.issueForm.markAllAsTouched();  
    return;  
  }  
  
  this.issueService.createIssue(this.issueForm?.value);  
  this.formClose.emit();  
}
```

- 
- ▶ En el fragmento anterior, usamos el método `markAllAsTouched` de la propiedad `issueForm` para marcar todos los controles como tocados cuando el formulario no es válido.
 - ▶ Marcar los controles como tocados hace que los mensajes de validación aparezcan automáticamente.
 - ▶ Además, utilizamos una declaración de devolución para evitar la creación del problema cuando el formulario no es válido.

- Ejecute ng serve para iniciar la aplicación. Haga clic dentro de la entrada Título y luego mueva el foco fuera del control de formulario:



The screenshot shows a web form with the title "Report an issue". Below the title is a text input field labeled "Title". A red horizontal line is drawn across the bottom of the input field, and a red exclamation mark icon is positioned at the right end of this line. Below the input field, the text "Title is required" is displayed in red, indicating a validation error.

- Debería aparecer un mensaje debajo de la entrada Título indicando que aún no hemos introducido ningún valor. Los mensajes de validación en la biblioteca Clarity se indican mediante texto y un icono de exclamación en rojo en el control de formulario que se valida.

- Ahora, haga clic en el botón CREATE:

Report an issue

Title

Title is required

Description

Priority

☐ Low ☐ High

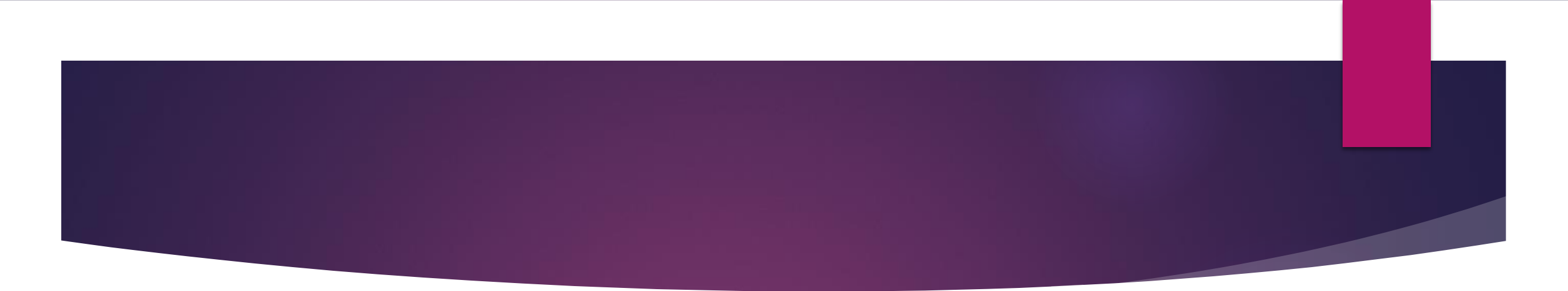
Priority is required

Type

Type is required

CREATE

CANCEL

- 
- ▶ Todos los mensajes de validación aparecerán en la pantalla a la vez y no se enviará el formulario.
 - ▶ Las validaciones en formas reactivas son una parte esencial para garantizar una UX fluida para nuestras aplicaciones.
 - ▶ Ahora vamos a ver cómo crear un diálogo modal con Clarity y usarlo para resolver problemas de nuestra lista.

Resolver un problema

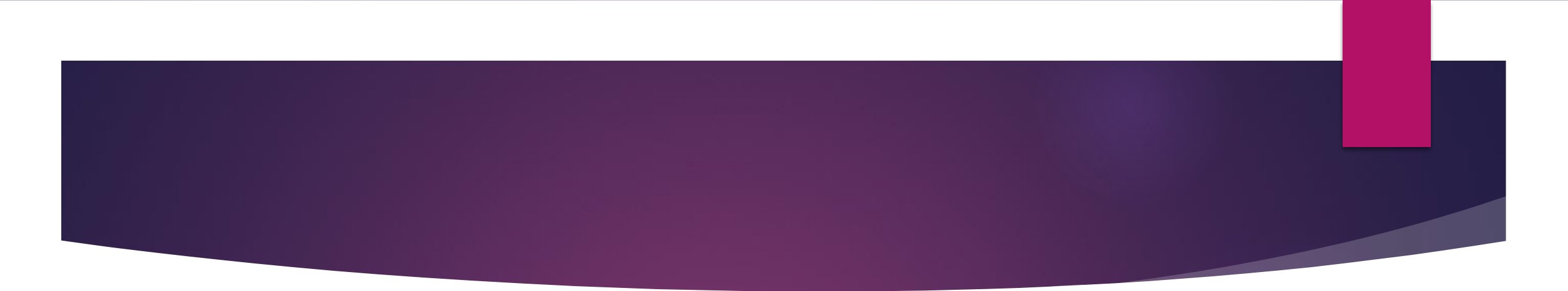
- ▶ La idea principal detrás de tener un sistema de seguimiento de problemas es que un problema debe resolverse en algún momento. Crearemos un flujo de trabajo de usuario en nuestra aplicación para realizar dicha tarea.
- ▶ Podremos resolver un problema directamente desde la lista de problemas pendientes. La aplicación pedirá confirmación al usuario antes de resolver con el uso de un diálogo modal.

- Cree un componente angular para alojar el diálogo:

```
issue-tracker> ng generate component confirm-dialog
```

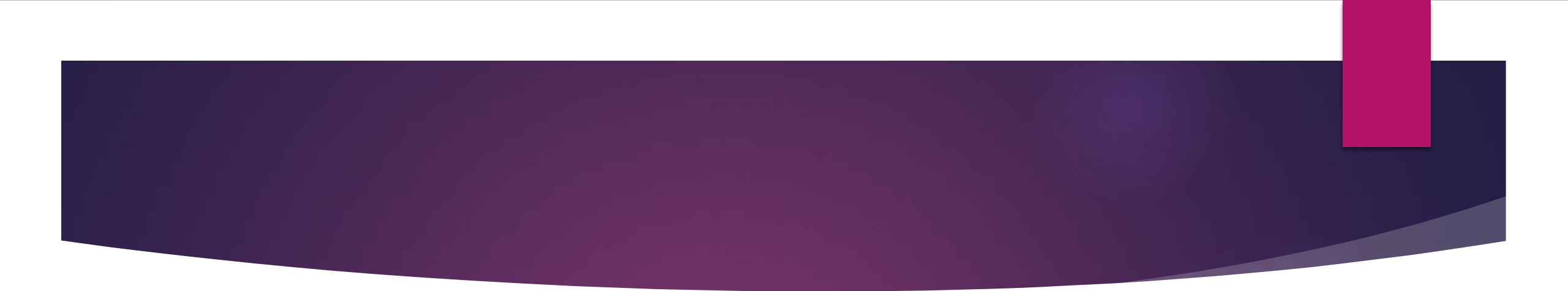
- Abra el archivo confirm-dialog.component.ts y cree las siguientes propiedades de entrada y salida en la clase ConfirmDialogComponent


```
src > app > confirm-dialog > confirm-dialog.component.ts > ...
1  import { Component, EventEmitter, Input, Output } from '@angular/core';
2  @Component({
3    selector: 'app-confirm-dialog',
4    templateUrl: './confirm-dialog.component.html',
5    styleUrls: ['./confirm-dialog.component.css'],
6  })
7  export class ConfirmDialogComponent {
8    @Input() issueNo: number | null = null;
9    @Output() confirm = new EventEmitter<boolean>();
10 }
```


- 
- ▶ Usaremos el decorador `@Input ()` para obtener el número de problema y mostrarlo en la plantilla del componente.
 - ▶ La propiedad `confirm` `EventEmitter` emitirá un valor booleano para indicar si el usuario confirmó que resolvió el problema o no.

- Cree dos métodos que llamarán al método emit de la propiedad de salida de confirmación, ya sea con verdadero o falso:

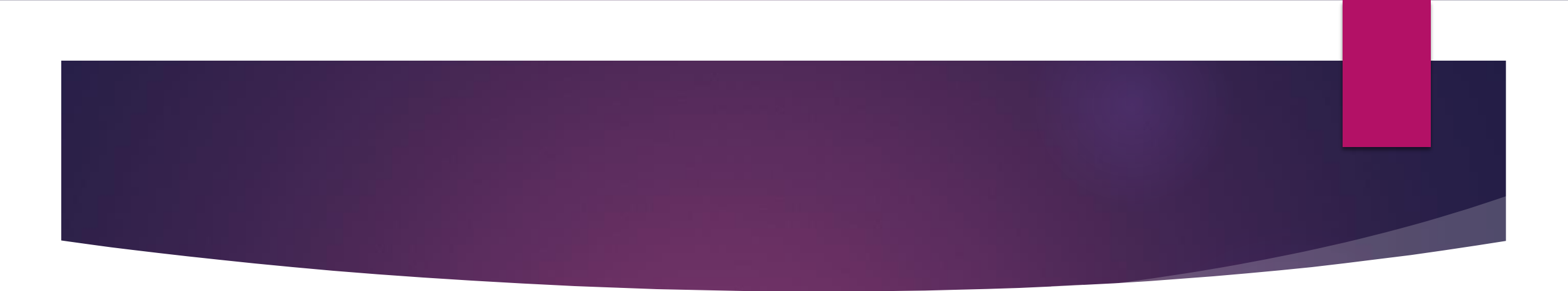
```
10  agree() {  
11    this.confirm.emit(true);  
12    this.issueNo = null;  
13  }  
14  disagree() {  
15    this.confirm.emit(false);  
16    this.issueNo = null;  
17  }
```

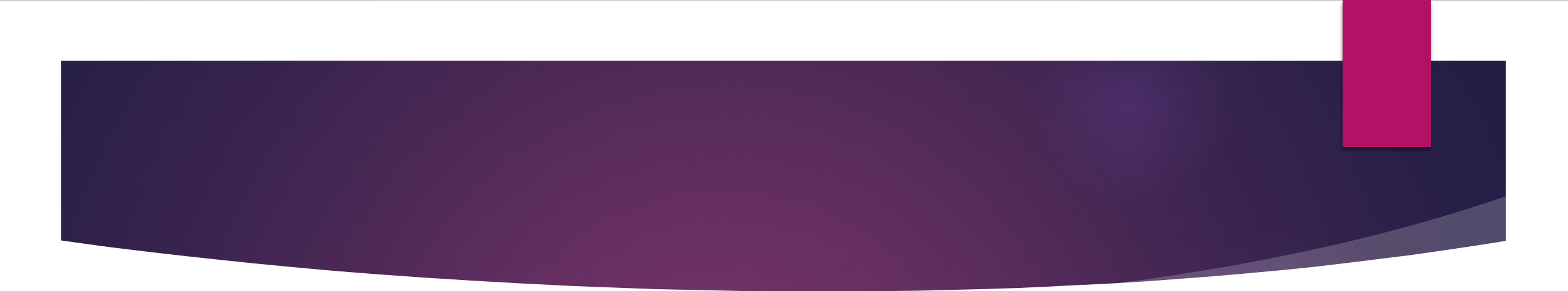
- 
- ▶ Ambos métodos establecerán la propiedad `issueNo` en `null` porque esa propiedad también controlará si el diálogo modal se abre o no.
 - ▶ Entonces, queremos cerrar el cuadro de diálogo cuando el usuario acepta resolver el problema o no.

src > app > confirm-dialog >  confirm-dialog.component.html > ...

Go to component

```
1  <clr-modal [clrModalOpen]="issueNo !== null" [clrModalClosable]="false">
2    <h3 class="modal-title">
3      Resolve Issue #
4      {{issueNo}}
5    </h3>
6    <div class="modal-body">
7      <p>Are you sure you want to close the issue?</p>
8    </div>
9    <div class="modal-footer">
10     <button type="button" class="btn btn-outline" (click)="disagree()">Cancel</button>
11     <button type="button" class="btn btn-danger" (click)="agree()">Yes, continue</button>
12   </div>
13 </clr-modal>
```

- 
- ▶ Un diálogo modal de Clarity consta de un componente `clr-modal` y una colección de elementos HTML con clases específicas:
 - ▶ `modal-title`: el título del cuadro de diálogo que muestra el número de problema actual.
 - ▶ `modal-body`: el contenido principal del diálogo.
 - ▶ `modal-footer`: el pie de página del diálogo que se usa comúnmente para agregar acciones para ese diálogo. Actualmente agregamos dos elementos de botón HTML y vinculamos sus eventos de clic a los métodos del componente de acuerdo y en desacuerdo, respectivamente.


- 
- ▶ Ya sea que esté abierto o cerrado, el estado actual del diálogo se indica mediante la directiva `clrModalOpen` vinculada a la propiedad de entrada `issueNo`. Cuando esa propiedad es nula, el diálogo se cierra. La directiva `clrModalClosable` indica que el cuadro de diálogo no se puede cerrar por ningún otro medio que no sea mediante programación a través de la propiedad `issueNo`.
 - ▶ De acuerdo con nuestras especificaciones, queremos que el usuario resuelva un problema directamente desde la lista. Veamos cómo podemos integrar el diálogo que creamos con la lista de asuntos pendientes.

- Abra el archivo `issues.service.ts` y agregue un nuevo método para establecer la propiedad completa de un problema:

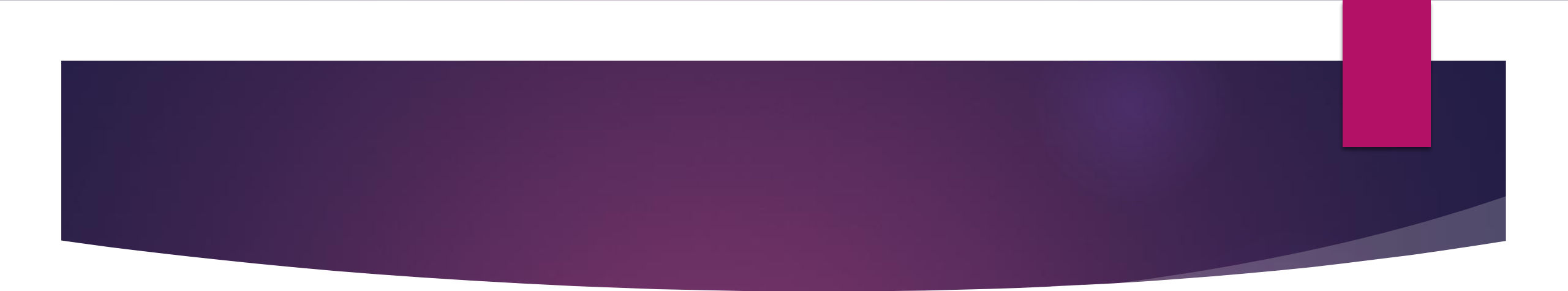
```
completeIssue(issue: Issue) {  
  const selectedIssue: Issue = {  
    ...issue,  
    completed: new Date(),  
  };  
  const index = this.issues.findIndex((i) => i === issue);  
  this.issues[index] = selectedIssue;  
}
```

- El método anterior primero crea un clon del problema que queremos resolver y establece su propiedad de completado a la fecha actual. Luego encuentra el problema inicial en la matriz de problemas y lo reemplaza con la instancia clonada

- Abra el archivo `issue-list.component.ts` y agregue una propiedad `selectedIssue` y un método `onConfirm`:

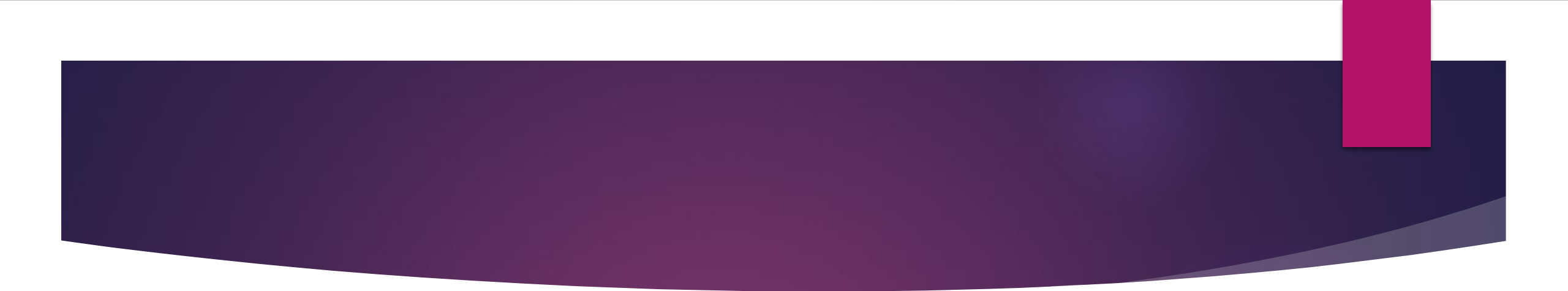
```
9  export class IssueListComponent implements OnInit {  
10     issues: Issue[] = [];  
11     showReportIssue = false;  
12     selectedIssue: Issue | null = null;   
13     constructor(private issueService: IssuesService) {}  
14     private getIssues() {  
15         this.issues = this.issueService.getPendingIssues();  
16     }  
17     ngOnInit(): void {
```

```
onConfirm(confirmed: boolean) {  
    if (confirmed && this.selectedIssue) {  
        this.issueService.completeIssue(this.selectedIssue);  
        this.getIssues();  
    }  
    this.selectedIssue = null;  
}
```


- 
- ▶ El método `onConfirm` llama al método `completeIssue` de la propiedad `issueService` solo cuando el parámetro `confirmado` es verdadero.
 - ▶ En este caso, también llama al método `getIssues` para actualizar los datos de la tabla.
 - ▶ La propiedad `selectedIssue` contiene el objeto de problema que queremos resolver y se restablece cada vez que se llama al método `onConfirm`.

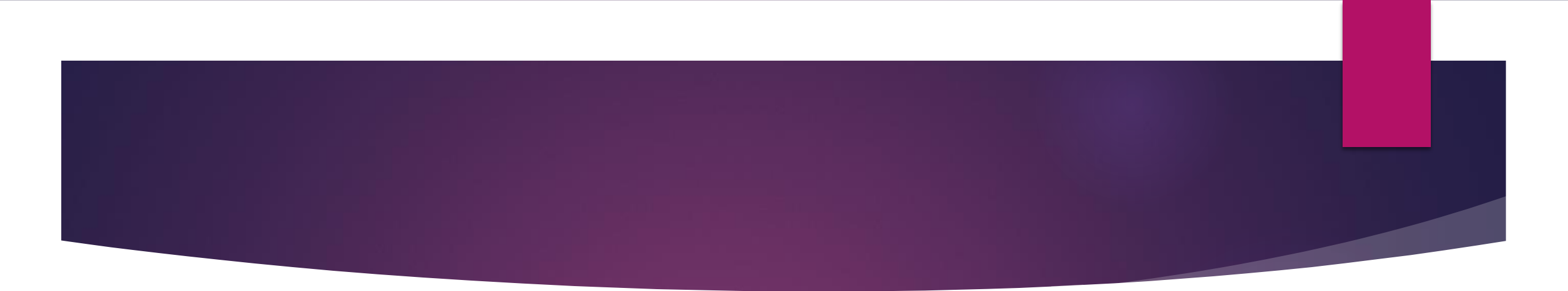
- Abra el archivo issue-list.component.html y agregue un componente de desbordamiento de acción dentro del componente clr-dg-row:

```
12 <clr-dg-row *clrDgItems="let issue of issues">
13   <clr-dg-action-overflow>
14     <button class="action-item" (click)="selectedIssue
15       = issue">Resolve</button>
16   </clr-dg-action-overflow>
17   <clr-dg-cell>{{issue.issueNo}}</clr-dg-cell>
18   <clr-dg-cell>{{issue.type}}</clr-dg-cell>
19   <clr-dg-cell>{{issue.title}}</clr-dg-cell>
20   <clr-dg-cell>{{issue.description}}</clr-dg-cell>
21   <clr-dg-cell>
```

- 
- ▶ El componente `clr-dg-action-overflow` de Clarity agrega un menú desplegable en cada fila de la tabla.
 - ▶ El menú contiene un único botón que establecerá la propiedad `selectedIssue` en el problema actual cuando se haga clic en él.

- Finalmente, agregue el componente app-confirm-dialog al final de la plantilla:

```
<app-confirm-dialog *ngIf="selectedIssue" [issueNo]="selectedIssue.issueNo" (confirm)="onConfirm($event)">
</app-confirm-dialog>
```

- 
- ▶ Pasamos el `issueNo` de la propiedad `selectedIssue` al enlace de entrada del componente de diálogo. Usamos el Operador “?” de navegación segura para evitar errores en nuestra aplicación porque inicialmente no hay ningún problema seleccionado y la propiedad `issueNo` no está disponible.
 - ▶ También vinculamos el método del componente `onConfirm` al evento de confirmación para que podamos ser notificados cuando el usuario esté de acuerdo o no. El parámetro `$event` es una palabra clave reservada en Angular y contiene el resultado de la vinculación del evento, que depende del tipo de elemento HTML. En este caso, contiene el resultado booleano de la confirmación.

- 
- ▶ Hemos puesto todas las piezas en su lugar para resolver un problema. Hagamos un intento:

1. Ejecute ng serve y abra la aplicación en `http: // localhost: 4200`.
2. Si no tiene ningún problema, use el botón ADD NEW ISSUE para crear uno.
3. Haga clic en el menú de acciones de una fila y seleccione Resolver. El menú es el icono de tres puntos verticales junto a la columna Número de problema:

← → ↻ ⓘ localhost:4200

ADD NEW ISSUE

	Issue No	Type
⋮	Resolve	Feature
⋮	3	Bug
⋮	4	Feature

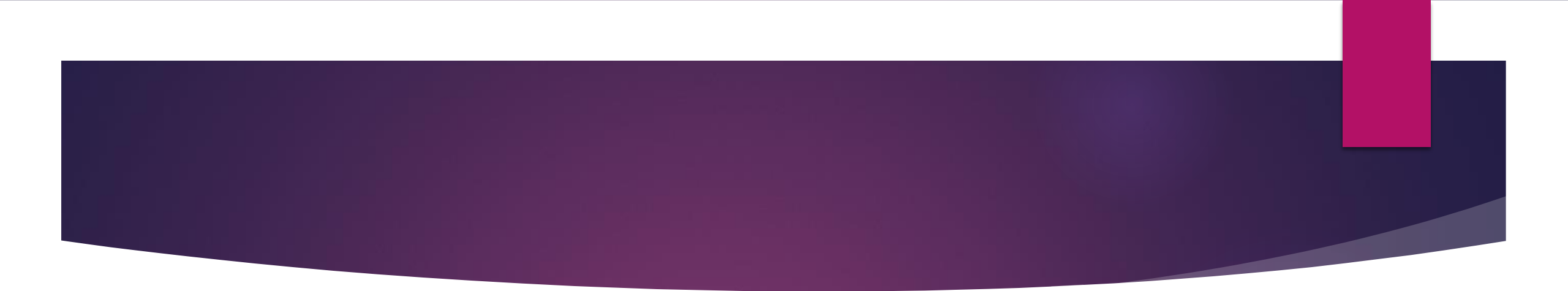
- En el cuadro de diálogo Resolver problema que aparece, haga clic en el botón YES, CONTINUE:

Resolve Issue # 2

Are you sure you want to close the issue?

CANCEL

YES, CONTINUE

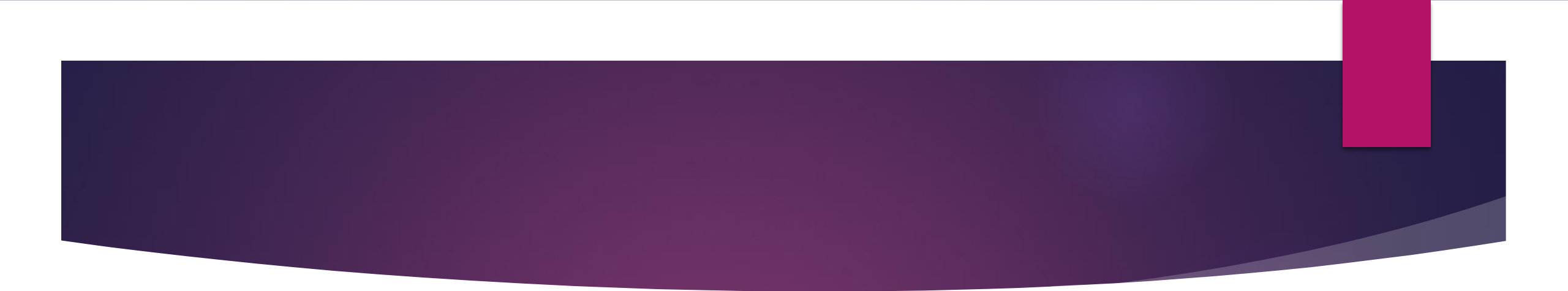
- 
- ▶ Después de hacer clic en el botón, el cuadro de diálogo se cerrará y el problema ya no debería aparecer en la lista.
 - ▶ Hemos proporcionado una forma para que los usuarios de nuestra aplicación resuelvan problemas. ¡Nuestro sistema de seguimiento de problemas ahora está completo y listo para ponerse en acción! A veces, los usuarios tienen prisa y pueden informar un problema ya informado.
 - ▶ En la siguiente sección, aprenderemos cómo aprovechar las técnicas avanzadas de formas reactivas para ayudarlos en este caso.

Activar sugerencias para nuevos problemas

- ▶ La API de formularios reactivos contiene un mecanismo para recibir notificaciones cuando cambia el valor de un control de formulario en particular.
- ▶ Lo usaremos en nuestra aplicación para encontrar problemas relacionados al informar uno nuevo.
- ▶ Más específicamente, mostraremos una lista de problemas sugeridos cuando el usuario comience a escribir en el control del formulario de título.

- Abra el archivo `issues.service.ts` y agregue el siguiente método

```
getSuggestions(title: string): Issue[] {  
  if (title.length > 3) {  
    return this.issues.filter(issue =>  
      issue.title.indexOf(title) !== -1);  
  }  
  return [];  
}
```

- 
- ▶ El método `getSuggestions` toma el título de un problema como parámetro y busca cualquier problema que contenga el mismo título.
 - ▶ El mecanismo de búsqueda se activa cuando el parámetro del título tiene más de tres caracteres para limitar los resultados a una cantidad razonable.

- Abra el archivo `issue-report.component.ts` y agregue la siguiente declaración de importación:

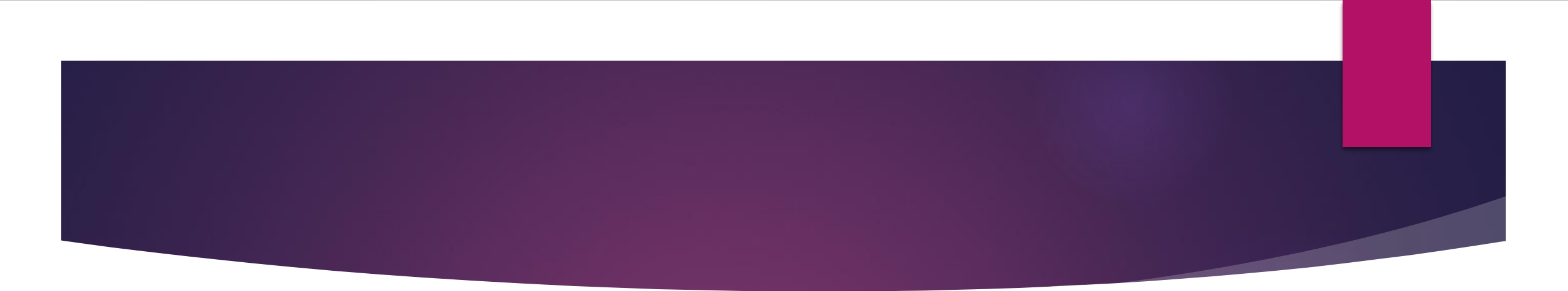
```
src > app > issue-report > TS issue-report.component.ts > IssueReportComponent > issueForm
1  import { EventEmitter, Output } from '@angular/core';
2  import { Component, OnInit } from '@angular/core';
3  import { FormBuilder, FormGroup, Validators } from '@angular/forms';
4  import { IssuesService } from '../issues.service';
5  import { Issue } from '../issue'
6  @Component({
```

- Cree una nueva propiedad de componente para contener los problemas sugeridos

```
export class IssueReportComponent implements OnInit {
  issueForm: FormGroup | undefined;
  @Output() formClose = new EventEmitter();
  suggestions: Issue[] = [];
  constructor(
    private builder: FormBuilder,
    private issueService: IssuesService
```

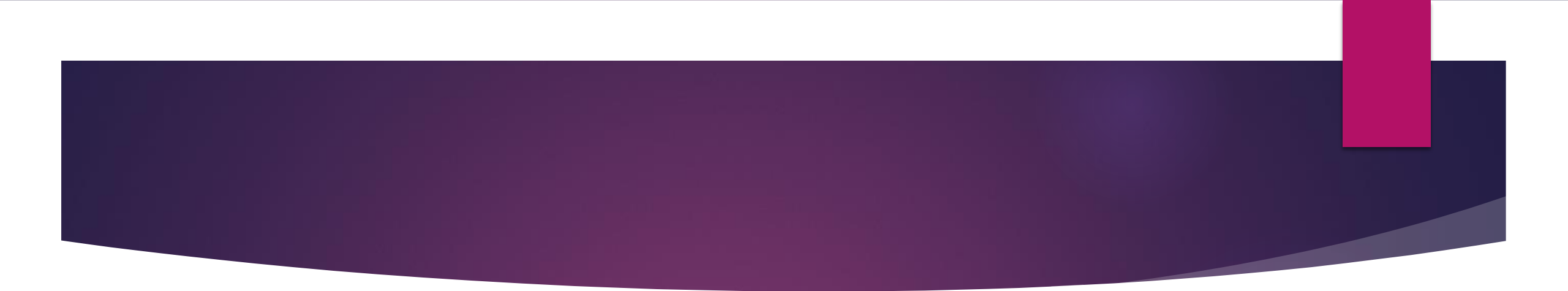
- ▶ La propiedad de controles de un objeto FormGroup contiene todos los controles de formulario como un par clave-valor.
- ▶ La clave es el nombre del control y el valor es el objeto de control de formulario real del tipo AbstractControl.
- ▶ Podemos recibir notificaciones sobre cambios en el valor de un control de formulario accediendo a su nombre, en este caso el título, de la siguiente manera:

```
ngOnInit(): void {  
  this.issueForm = this.builder.group({  
    title: ['', Validators.required],  
    description: ['', Validators.required],  
    priority: ['', Validators.required],  
    type: ['', Validators.required],  
  });  
  this.issueForm.controls.title.valueChanges.subscribe(  
    title: string => {  
      this.suggestions =  
        this.issueService.getSuggestions(title);  
    }  
  );  
}
```

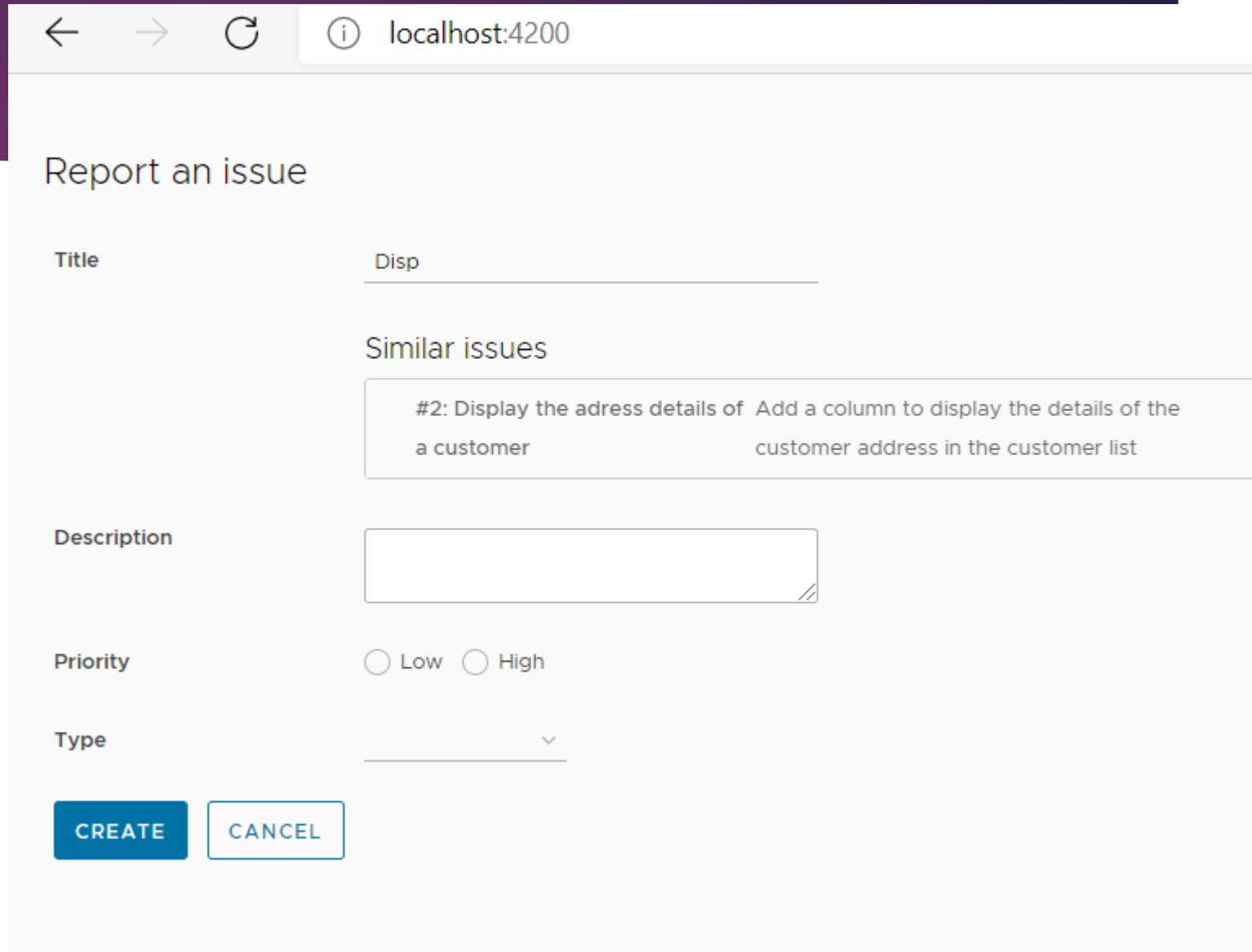
- 
- ▶ Cada control expone un `valueChanges` observable al que podemos suscribirnos y obtener un flujo continuo de valores.
 - ▶ El observable `valueChanges` emite nuevos valores tan pronto como el usuario comienza a escribir el control de título del formulario.
 - ▶ Establecemos el resultado del método `getSuggestions` en la propiedad del componente de sugerencias cuando eso sucede

- Para mostrar los problemas sugeridos en la plantilla del componente, abra el archivo `issue-report.component.html` y agregue el siguiente código HTML justo después del elemento `clr-input-container`:

```
<div class="clr-row" *ngIf="suggestions.length">
  <div class="clr-col-lg-2"></div>
  <div class="clr-col-lg-6">
    <clr-stack-view>
      <clr-stack-header>Similar issues
      </clr-stack-header>
      <clr-stack-block *ngFor="let issue of
suggestions">
        <clr-stack-label>#{{issue.issueNo}}:
        | {{issue.title}}</clr-stack-label>
        <clr-stack-content>{{issue.description}}
        </clr-stack-content>
      </clr-stack-block>
    </clr-stack-view>
  </div>
</div>
```


- 
- ▶ Usamos el componente `clr-stack-view` de la biblioteca Clarity para mostrar los problemas sugeridos en una representación de par clave-valor.
 - ▶ La clave está indicada por el componente `clr-stack-header` y muestra el título y el número del problema.
 - ▶ El valor lo indica el componente `clr-stack-content` y muestra la descripción del problema.

- Ejecute ng serve y abra el formulario de informe de problemas para crear un nuevo problema. Cuando empiece a escribir la entrada Título, la aplicación le sugerirá cualquier problema relacionado con el que está intentando crear:



← → ↻ ⓘ localhost:4200

Report an issue

Title

Similar issues

#2: Display the address details of a customer

Add a column to display the details of the customer address in the customer list

Description

Priority ☐ Low ☐ High

Type

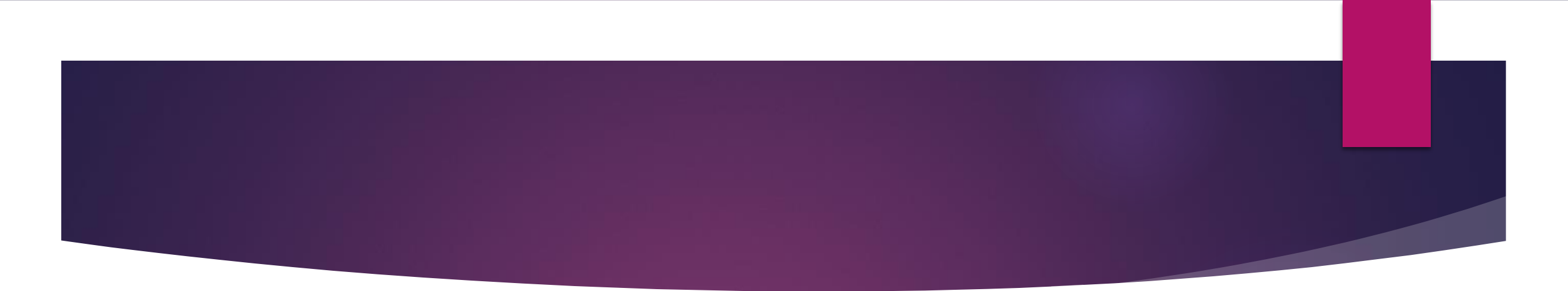
CREATE **CANCEL**

Resumen

- ▶ Creamos una aplicación Angular para administrar y rastrear problemas usando formularios reactivos y Clarity Design System.
- ▶ Instalamos Clarity en una aplicación Angular y usamos un componente grid para mostrar una lista de problemas pendientes. Luego, presentamos formularios reactivos y los usamos para crear un formulario para informar un nuevo problema. Agregamos validaciones en el formulario para brindarles a nuestros usuarios una indicación visual de los campos requeridos y protegerlos contra comportamientos no deseados.
- ▶ Un sistema de seguimiento de problemas no es eficiente si nuestros usuarios no pueden resolverlos. Creamos un diálogo modal usando Clarity para resolver un problema seleccionado. Finalmente, mejoramos la UX de nuestra aplicación sugiriendo problemas relacionados al informar sobre uno nuevo.

Ejercicio

- ▶ Cree un componente angular para editar los detalles de un problema existente. El componente debe mostrar el número del problema y permitir al usuario cambiar el título, la descripción y la prioridad. El título y la descripción deben ser campos obligatorios.
- ▶ El usuario debería poder acceder al componente anterior utilizando el menú de acciones en la lista de problemas pendientes. Agregue un nuevo botón en el menú de acciones que abrirá el formulario de edición de problemas.
- ▶ Una vez que el usuario ha completado la actualización de un problema, el formulario debe cerrarse y la lista de problemas pendientes debe actualizarse.

- 
- ▶ Angular Forms: <https://angular.io/guide/forms-overview>
 - ▶ Reactive forms: <https://angular.io/guide/reactive-forms>
 - ▶ Validación de formularios reactivos: <https://angular.io/guide/form-validation#validating-input-in-reactive-forms>
 - ▶ Pasar datos a un componente: <https://angular.io/guide/component-integration#pass-data-from-parent-to-child-with-input-binding>
 - ▶ Obtener datos de un componente: <https://angular.io/guide/component-integration#parent-listens-for-child-event>
 - ▶ Introducción a Clarity: <https://clarity.design/documentation/get-started>