Angular Components

What are NgModules ?

Component Basics

Standalone Components vs NgModules

Data Binding

Content Projection

Component Inputs & Outputs

Component Lifecycle

The main.ts file

Every Angular application **prior to standalone components** must have at least **one** root module called AppModule **(app.module.ts file)** by default, which is loaded in the **main.ts** file when the application starts.

```
import { platformBrowserDynamic } from
'@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic()
  .bootstrapModule(AppModule, {
    ngZoneEventCoalescing: true,
  })
  .catch((err) => console.error(err));
```

What are NgModules ?

An **ngModule** (Angular Module) is a way to group related components in Angular, organizing your application into manageable, feature-specific modules.

Application Division into Modules: Angular applications are divided into multiple modules, each implementing a certain feature, enhancing clarity and maintainability.

Similar to Java Packages: ngModules are like Java packages; components within the same module can use each other without explicit import statements.

What are NgModules ?

Defining Dependencies: ngModules define dependencies for their components, such that all components in the same module have access to all the dependencies that the module imports, reducing redundant import statements.

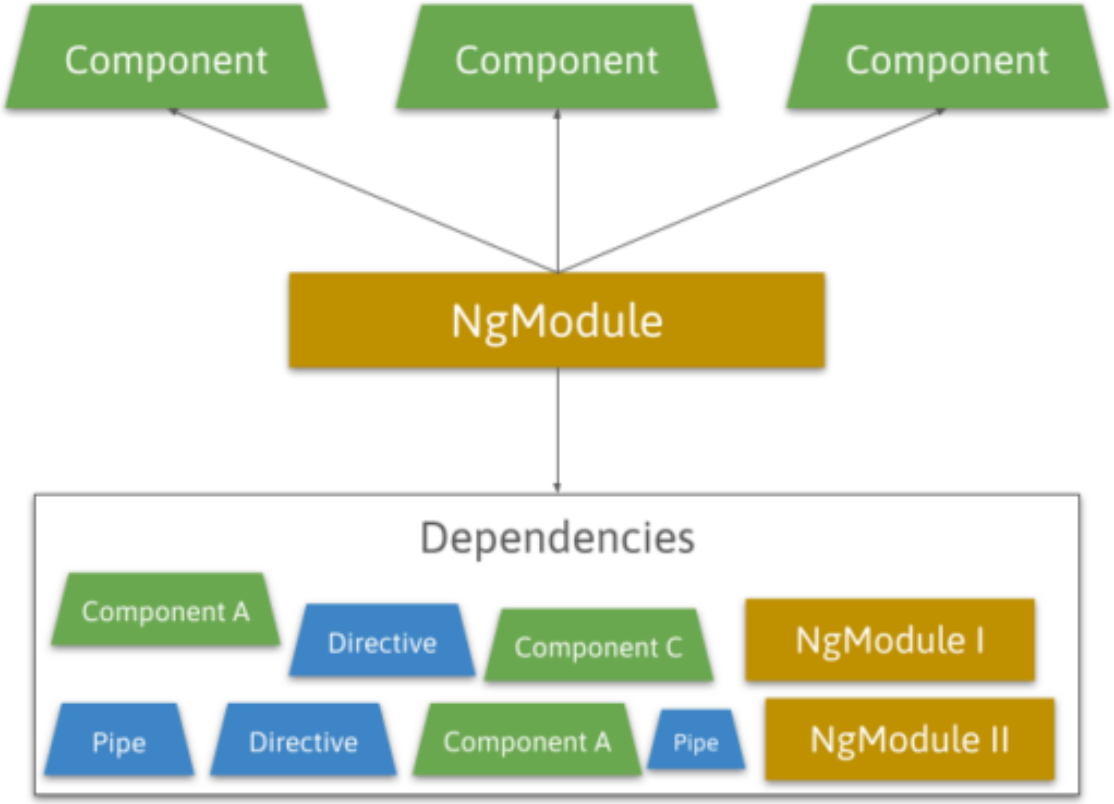
Bootstrap Component: The AppModule specifies a root component (the bootstrap component) that is displayed when the application runs.

Lazy Loading for Performance: ngModules support lazy loading, loading only necessary modules at startup and others when needed, improving performance.

Module Importing and Exporting: ngModules can import other modules and use their exported components, promoting code reuse and separation of concerns.

The AppModule

```
// An NgModule is just a typescript class with the @NgModule decorator.
// The decorator provides metadata about the module.
@NgModule({
  // The declarations array contains a list of components that belong to this module.
  declarations: [AppComponent],
  // The imports array contains a list of other modules that this module depends on.
  // The BrowserModule provides the necessary infrastructure to run your application in a browser environment.
  imports: [BrowserModule],
  // The exports array contains a list of components that this module makes available to other modules.
  exports: [],
  // The providers array contains a list of services that this module provides.
  providers: [],
  // The bootstrap array contains the root component that Angular creates and inserts into the index.html file.
  // This is only used in the root module.
  bootstrap: [AppComponent],
})
export class AppModule {}
// The Module class is exported so that it can be imported in other files.
// Ex: import { AppModule } from './app.module'; in main.ts
```


Component
Component
Component
NgModule
Dependencies
Component A
Directive
Component C
NgModule I
Pipe
Directive
Component A
Pipe
NgModule II

Creating Feature Modules

Each feature in an Angular application usually has its own module, You can create a new module file manually by writing the necessary code yourself or by using the Angular CLI command :

```
ng generate module Cart
```

This command will create a new folder named after your module inside the app folder (the generate module command is relative to the app folder) and generate a cart.module.ts file inside it.

If you want to place the module in a specific directory other than the app folder, include the path:

```
ng generate module ../features/Cart
```

Angular will create the necessary directories if they don't exist.

This will create a new folder called **features** next to the app folder with a new folder named **cart** inside it and generate a cart.module.ts file in it.

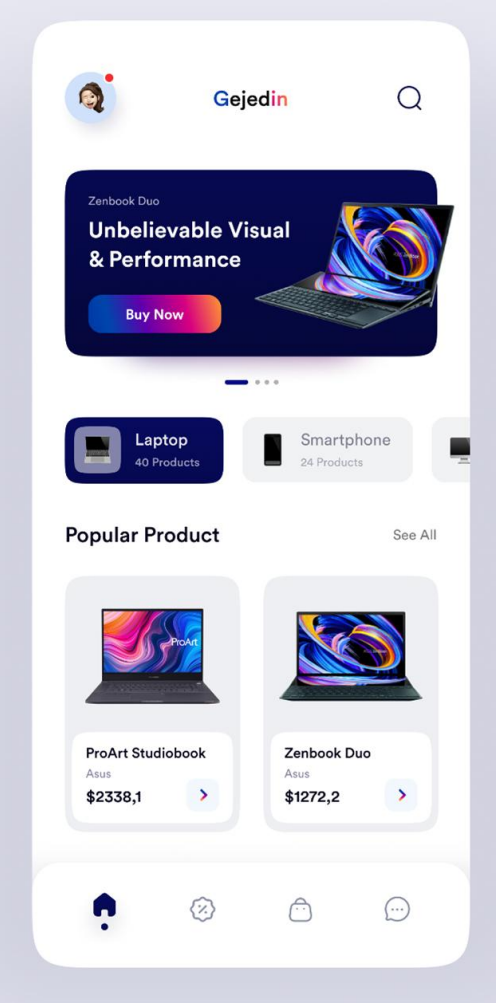
Angular Components

Every Angular application is divided into small, reusable pieces called components.

Each component has its own:

- Template (HTML)
- Behavior (TypeScript)
- Styling (CSS)

This makes each component self-contained easy to manage, and reuse. Example: A product Card



Angular Components

A component is like a custom HTML element that you create with its own template, logic, and styles, usable throughout your application like any standard HTML element.

Every Angular application must have at least one component called **AppComponent** by default, serving as the entry point displayed when the application runs.

In Angular applications using NgModules and not standalone components, any component must be part of a module, each component must be declared in a module's **declarations** array to be recognized and used within the application.

The AppComponent

```
// A component is just a typescript class with the @Component decorator.
// The @Component decorator provides metadata about the component.
@Component({
  selector: 'app-root', // The selector is the name of the HTML tag that represents the component.
  templateUrl: './app.component.html', // The templateUrl is the path to the HTML file that is
displayed when the component is rendered.
  styleUrl: './app.component.css', // The styleUrl is the path to the CSS file that applies styles
to the component.
})
export class AppComponent { // The class is exported so that it can be imported in other files.
  // The class contains any state or behavior that the component needs.
  title = 'app';
}
```

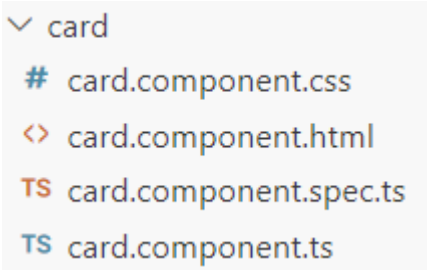
Using The AppComponent

```
@NgModule({
  declarations: [AppComponent], // The AppComponent is included in the declarations array.
  imports: [BrowserModule],
  exports: [],
  providers: [],
  // The AppComponent is the root component that Angular creates and inserts into the
index.html file.
  bootstrap: [AppComponent],
})
export class AppModule {}
```

In the index.html file the **AppComponent** is used by its **selector name**, Angular identifies the custom tag and inserts inside it the content defined in the component's HTML file during rendering.

```
<body>
  <app-root></app-root>
</body>
```

Creating New Components



You can create a new component manually by writing the necessary code files yourself or by using the Angular CLI command :

```
ng generate component Card
```

This will create a new folder named after your component inside the app folder (the generate command is relative to the app folder) and generate 4 files inside it:

- **card.component.ts**: Contains the TypeScript class for the component, defining its logic and data.
- **card.component.html**: Holds the HTML template that specifies how the component's view is rendered.
- **card.component.css**: Includes the CSS styles specific to the component for styling its template.
- **card.component.spec.ts**: Provides a unit test file for writing tests to ensure the component works correctly.

The new component is automatically added to the AppModule’s declarations array.

```
declarations: [AppComponent, CardComponent], // The CardComponent was added to the declarations array.
```

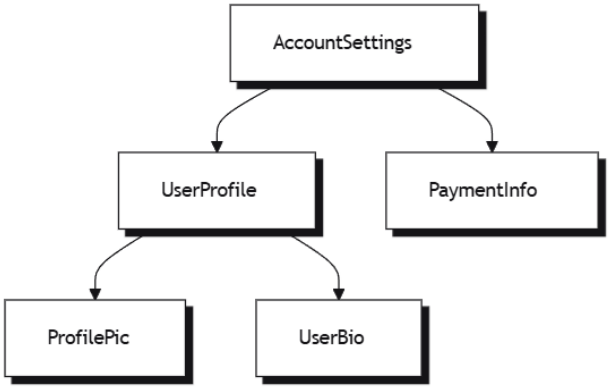
Nesting Components

A component can use another component inside its own template by using the child component's selector.

Ex: The AppComponent can use the CardComponent inside its template.

```
<h1>App Component</h1>

<app-card></app-card>
```



App Component

Card Component

Standalone Components vs NgModules

Standalone components can exist and be used without being part of an NgModule: They don't need to be declared in an NgModule's declarations array and can be imported and used directly in other components or modules.

Standalone components define only the dependencies they need using an imports array similar to modules: Each standalone component specifies its own dependencies in its imports array, ensuring it only has access to what it needs, rather than all the dependencies of a module.

Simplifies code with less boilerplate since there's no need to declare the component in an NgModule: By eliminating the need to declare components in modules, standalone components reduce the amount of code and configuration required, making development more straightforward.

Standalone Components vs NgModules

Standalone components make lazy loading easier since they can be lazy loaded individually, loading only the needed component and its dependencies, improving performance by not loading an entire module.

The main.ts file bootstraps the application using a standalone component instead of a module: Instead of bootstrapping an NgModule, the application starts by bootstrapping a standalone component, simplifying the application's entry point.

```
bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

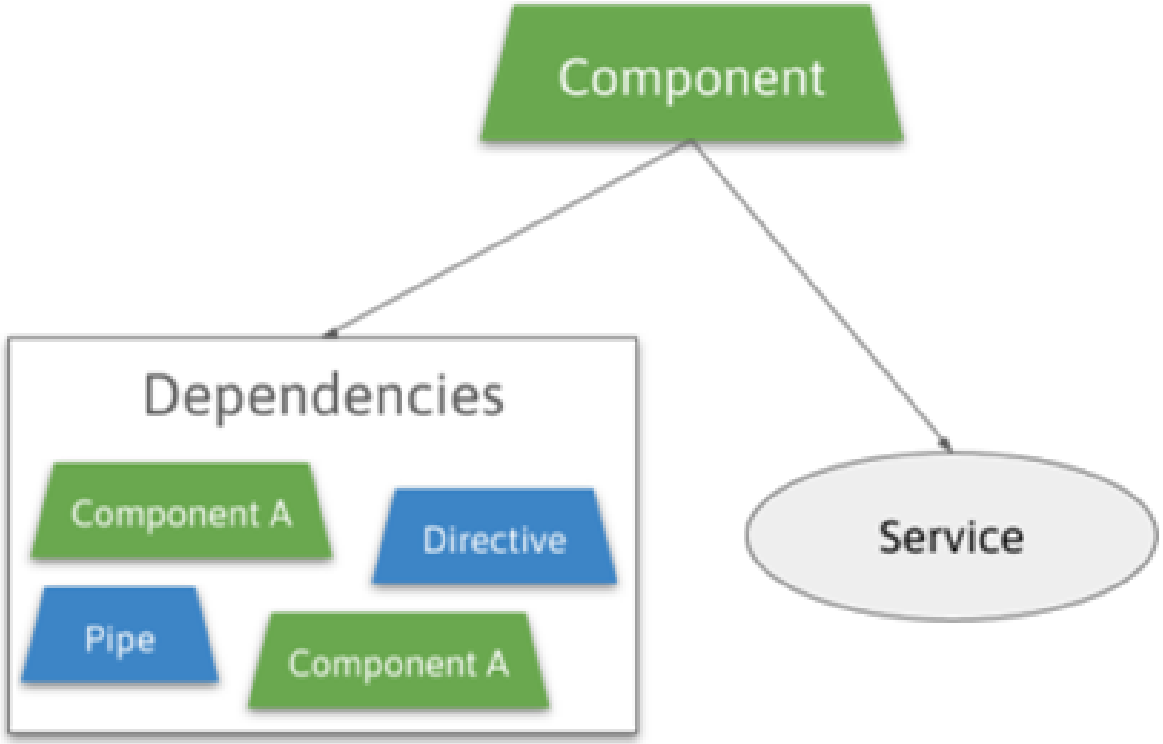
Standalone Components Usage

Starting from Angular 14, you can create standalone components using **standalone: true** in the @Component decorator. You can also use `--standalone` flag when generating components:

```
ng generate component Card --standalone
```

Standalone components can help create Angular applications that are easier to develop, maintain, and optimize for performance. **But as many existing applications still use NgModules we must understand them.**

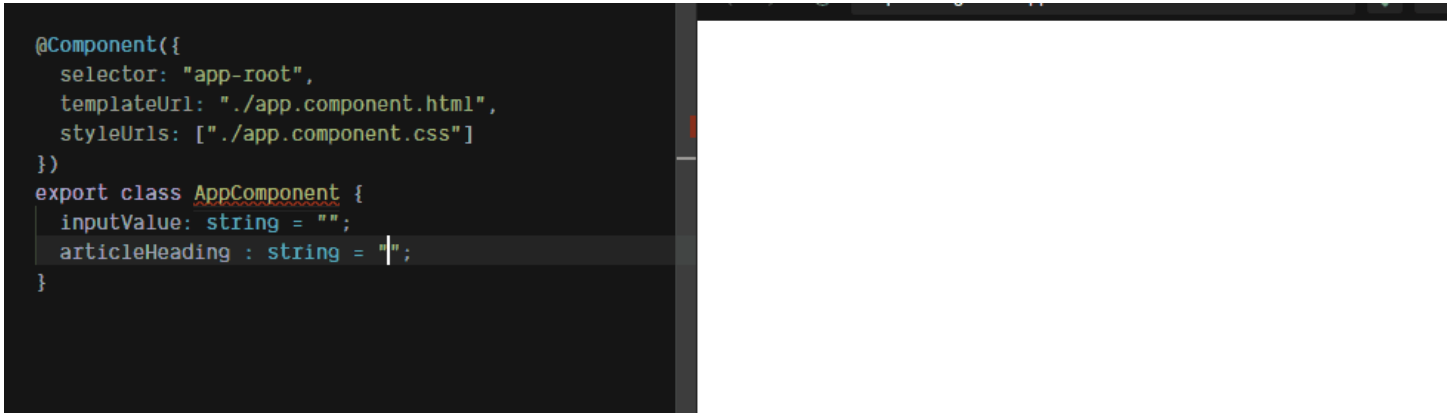
```
@Component({
  selector: 'app-root',
  standalone: true, // The standalone property is set to true.
  imports: [], // The dependencies the component needs are listed in the imports array.
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'app';
}
```


Component
Dependencies
Component A
Directive
Pipe
Component A
Service

Text Interpolation in Components

Text Interpolation is used to display the value of a variable from your component class in the HTML template using its name inside double curly braces like this: `{{ variableName }}`

When the variable's value changes in the component, Angular automatically updates the template to display the new value.



Text Interpolation in Components

```
// The app.component.ts file
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',
})
export class AppComponent {
  title = 'App Title';
  // Private properties are not
accessible in the template.
  private description = 'Description';
}
```

```
// The app.component.html file.
The name of any variable in the
component class can be accessed
in the template using the double
curly braces syntax {{ }}.
<h1>{{ title }}</h1>
This Displays "App Title"
```

App Title

Property Binding in Components

Bind Component Variables to Element Properties: Property binding allows you to set values of HTML element properties using variables from your component class. Use square brackets around the property in the template like this: `[property]="variableName"`

Example Usage: To bind the src attribute of an <img> tag to a component variable imageUrl, write **<img [src]="imageUrl">**

Automatic Updates: When the value of the variable in the component changes, Angular automatically updates the element's property in the view.

Property Binding in Components

```
// The app.component.ts file
@Component({
  selector: 'app-root',
  templateUrl:
'./app.component.html',
  styleUrl: './app.component.css',
})
export class AppComponent {
  imageUrl =
'https://picsum.photos/200';
}
```

```
// The app.component.html file.
Property binding is used to set the
value of an HTML element's property
using the value of a component's
property. The name of the HTML
element's property is enclosed in
square brackets.
Now the value inside the double quotes
can be the name of any property
of the component class.
<img [src]="imageUrl">
```

Event Binding in Components

Respond to User Actions: Event binding allows you to listen for user events like clicks or key presses in your template and execute methods in your component class. Use parentheses around the event name like this: `(event)="methodName()"`

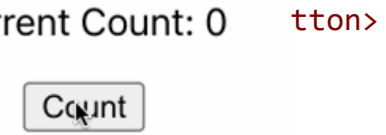
Example Usage: To call a method onClick() when a button is clicked, write `<button (click)="onClick()">Click Me</button>`

Interactive Behavior: When the user triggers the event, Angular calls the specified method, allowing your component to respond to user interactions.

Event Binding in Components

```ts
// The app.component.ts file
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',
})
export class AppComponent {
  count = 0;
  increment() {
    this.count++;
  }
}
```

```html
// The app.component.html file.
<h1>{{ count }}</h1>
Event binding in Angular is used to listen
for and respond to user actions such as
clicks, key presses, or mouse movements.
The event name is enclosed in parentheses.
The value inside the double quotes is the
method call that will be executed when the
event occurs.
<button
(click)="inc
```



Two Way Data Binding

Synchronize Data Between Component and View: Two-way data binding keeps a component's variable and the UI element in the template in sync. Changes in the input update the variable, and changes in the variable update the input.

Use [(ngModel)] Syntax: Implement two-way binding by using the **[(ngModel)] directive** with both square and parentheses brackets like this: `[(ngModel)]="variableName"`

Example Usage: To bind a text input to a component variable username, write `<input [(ngModel)]="username">`. As the user types, username updates automatically, and if username changes in the component, the input field reflects the new value.

Import FormsModule to Use ngModel: The ngModel directive is part of Angular's FormsModule. To use ngModel, you need to import FormsModule in your module file (e.g., app.module.ts) and add it to the imports array of the module.

Import The FormsModule

```
// Import the FormsModule module
import { FormsModule } from '@angular/forms';
@NgModule({
  declarations: [AppComponent],
  // Add the FormsModule module to the imports array to have access to ngModel
  imports: [BrowserModule, FormsModule],
  exports: [],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Two Way Data Binding


```
// The app.component.ts file
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',
})
export class AppComponent {
When the user types in the input element, the
userName property is updated, and when the
userName changes inside the component the
input element in the template is updated
  userName = '';
}
```

```
// The app.component.html file.
The syntax for two-way data binding is
a combination of property binding and
event binding, enclosed in square
brackets and parentheses respectively.
The value inside the double quotes is
the name of the property in the
component class.
<input [(ngModel)]="userName" />
<h2>{{ userName }}</h2> This will
always display the value of the input
field.
```

Content Projection in Components

Insert External Content into a Component's Template: Content projection allows you to pass content from a parent component into a child component's template.

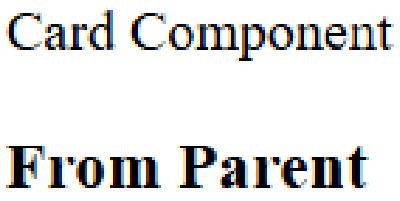
You place custom content inside the child component's selector tags, and the child component uses `<ng-content />` to display it.

Use <ng-content> to Mark Insertion Points: In the child component's template, you include the `<ng-content />` tag where you want the projected content to appear. This acts as a placeholder for the content provided by the parent component.

Content Projection in Components

```
// The app.component.html file
<app-card>
This heading will be displayed in
the card component's template.
  <h3>From Parent</h3>
</app-card>
```

```
// The card.component.html file
<p>Card Component</p>
The ng content tag will be
replaced by the content passed to
the card component.
<ng-content />
```



Component Inputs

Pass Data from Parent to Child Components: Component inputs allow a parent component to pass data to a child component, customizing the child component's behavior or display based on the provided data.

Use @Input() Decorator: In the child component, declare an input property by adding `@Input()` before the property name. This tells Angular that this property can receive data from a parent component.

Component Inputs

```
// The card.component.ts file
// Import the Input decorator
import { Input } from '@angular/core';
@Component({
  selector: 'app-card',
  templateUrl: './card.component.html',
  styleUrl: './card.component.css',
})
export class CardComponent {
 The message property will be passed to this
component from the parent component's template
  @Input()
  message: string = '';
}
```

```
// The app.component.html file
<app-card message="From Parent"></app-card>

// The card.component.html file
<h3>{{ message }}</h3>
```

From Parent

Component Outputs

Send Data from Child to Parent Components: Component outputs allow a child component to emit events to its parent component, so the parent can respond to actions or data changes in the child.

Use @Output() and EventEmitter: In the child component, declare an output property by adding `@Output()` before the property name and assign it a new `EventEmitter` instance. This lets the child component emit events that the parent can listen to.

Component Outputs

```
// Import the EventEmitter class
import { Component, EventEmitter } from '@angular/core';
// Import the Output decorator
import { Output } from '@angular/core';
@Component({
  selector: 'app-card',
  templateUrl: './card.component.html',
  styleUrl: './card.component.css',
})
export class CardComponent {
  Declare an output property
  @Output() notify = new EventEmitter<string>();
  notifyParent() {
    Emit event with data
    this.notify.emit('Message from Child');  }
}
```

```
// The app.component.html file
The event name is the same as the name
of the EventEmitter property in the
child component.
$event is the data that is passed from
the child component to the parent
component.
onNotify() is the function that is
called when the child component emits
an event.
<app-card
(notify)="onNotify($event)"></app-card>
```

The Component Lifecycle

The component lifecycle refers to the series of stages a component goes through from creation to destruction, allowing you to tap into key moments in its existence.

Lifecycle Hooks are Special Methods: Angular provides lifecycle hook methods that you can implement in your component class to run code at specific stages.

The Component Lifecycle Hooks

constructor(): Initializes the component instance when Angular creates it.

ngOnInit(): Called once after the component's inputs are initialized; used for component setup.

ngOnChanges(changes: SimpleChanges): Called when any input property changes; used to react to those changes.

The Component Lifecycle Hooks

ngAfterContentInit(): Runs once after content projection is initialized; used for initializing projected content.

ngAfterViewInit(): Runs once after the component's view and child views are initialized; used for view setup.

ngOnDestroy(): Invoked just before the component is destroyed; used for cleanup tasks.

The Constructor vs ngOnInit

```
@Component({
  selector: 'app-card',
  templateUrl: './card.component.html',
  styleUrl: './card.component.css',
})
export class CardComponent {
  @Input()
  message: string = '';
  // The constructor is called when Angular creates a new instance of the component
  constructor() {
    console.log(this.message); // message is still an empty string
  }
  // The ngOnInit method is called after Angular has set the input properties
  ngOnInit() {
    console.log(this.message); // message has the value passed from the parent component
  }
}
```

Hands On

1. **Design a Login Form**: Create input fields for email and password, and a login button in your template.
2. **Two-Way Data Binding**: Use `[(ngModel)]` to bind the input values to email and password properties in your component.
3. **Handle Login Logic**: Implement a login() method that is called when the login button is clicked to check if the entered credentials are correct.
4. **Display Messages**: Show a success message if the login is successful or an error message if it fails.
5. **Import FormsModule:** Don't forget to import FormsModule in your module to use `[(ngModel)]`.

The app.component.ts File

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css',
})
export class AppComponent {
  email: string = '';
  password: string = '';
  loginState: string = '';
  onLoginClick() {
    if (this.email === 'Basem@gmail.com' && this.password === '123456') {
      this.loginState = 'Login successful';
    } else {
      this.loginState = 'Login failed';
    }
  }
}
```

The app.component.html File

```
<form (submit)="onLoginClick()">
  <input type="email" [(ngModel)]="email" name="email"
placeholder="Email" required />
  <br />
  <input type="password" [(ngModel)]="password" name="password"
placeholder="Password" required />
  <br />
  <button type="submit">Login</button>
</form>
<h3>{{ loginState }}</h3>
```

Thank You ;)

Any Questions ?