Components and Databinding

Splitting Apps with Components

- Splitting components makes your individual components leaner and easier to read
- Doing so can be tricky if you have data that needs to exist in multiple areas
- You should split components on logical topics
    - Example → **Input** goes to **cockpit**, and *output* goes to *server-element*

Property and Event Binding Overview

- We need to either send data to a component or receive data from an event
- We can bind properties to components
- Property Binding and Events for sharing information
    - Native Properties and Events
        - HTML Events
    - Custom Properties and Events
        - Directives
        - Components

Binding to Custom Properties

- By default, all component properties are only accessible by the components themselves
    - You must be explicit with what properties are vulnerable to the world
    - For parent components to bind to a child component's property, you must add the **@Input** decorator to the child component
        - Must be activated with parentheses
        - **@Input()** → **@Input()** element: { type: string }
        - Data flows from the parent component to the child component

Assigning an Alias to Custom Properties

- We may sometimes not want to use the same property in the component anywhere else
- In the parent component, we may bind to another element name
  - This normally wouldn't work as the child has no property of that name
  - To remedy this, we can write an alias as a string in the **@Input(<alias>)** decorator

Binding to Custom Events

- What if we want to pass data from the child component to the parent component?
- To do this, we add an event binding element in the child component element of the parent component's template
  - We can use any property name
  - We must use a parent component method with the **$event** argument
    - **$event** → Tells us the required information of the target element
  - The parent awaits data from the child component
- Whatever property name you use in the parent component must be added as class variables in the child component
  - These variables are decorated with the **@Output()** decorator → Data flows from the child component to the parent component
  - Since these are properties that emit data (aka events), we make them **new EventEmitter<emit-type-data>()** objects
  - **EventEmitter** → Angular object that allows us to emit our own events
- To actually emit data, we must create **emitter** functions in the child component that call the **emit** method from the **EventEmitter** objects
  - We pass an object containing vital data as an argument

Assigning an Alias to Custom Events

- Very similar to assigning aliases to custom properties
- To assign an alias to a custom event, pass a string literal as an argument into the **@Output** decorator

Custom Property and Event Binding Summary

- **@Input** → Transmits data from the parent component to the child component using **property-binding** in the parent and **Input()** in the child
- **@Output** → Transmits data from the child component to the parent component using **event-binding** in the parent and **@Output()** in the child
  - Done so by using the an **EventEmitter** object
- Issue → Chains of inputs and outputs can grow incredibly complex
  - Can only communicate between two adjacent levels of components in a hierarchy
  - Improved with using services in these scenarios

Understanding View Encapsulation

- In Angular, the styles described in a component's stylesheet only applies to that particular component
- Since Angular applies **ngcontent** attributes to elements, we can't generalize styles across all components → without thematic magic, of course
  - All elements in a component get the same **ngcontent-<val>-<val>** attribute
  - Emulates the **ShadowDOM**

More on View Encapsulation

- We can override the aforementioned encapsulation by adding to the **@Component** decorator
  - **encapsulation**: ViewEncapsulation.None
    - These style changes apply globally (all components)
  - **encapsulation**: ViewEncapsulation.Native
    - Gives the same result as above, but through **ShadowDOM** functionality
  - **encapsulation**: ViewEncapsulation.Emulated
    - Works for various browsers

Using Local References in Templates

- Local references can be placed on any HTML element
- Accompany it with a **#** and name of choice → **#serverNameInput**
- These references can be passed elsewhere in the template or the component
    - Minus the **#**
    - Referenced in the component as a parameter of type **HTMLInputElement**
    - Actual value referenced by **<parm-name>.value**

Getting Access to the Template and DOM with @ViewChild

- Another method of getting access to local references, or even any element, from the component
- Arguments
    - The first argument is the targeted element's selector
        - Local references are the most common, component afterward
    - All uses of **@ViewChild()** - same with **@ContentChild()** - require **{ static: true }** as a second argument
- The variable's type is **ElementRef** → Native to Angular
    - Accessed using **<var-name>.nativeElement.value**
- We shouldn't change the element through this
- Typical call
    - **@ViewChild('serverContentInput') serverContentInput: ElementRef;**

Projecting Content into Components with ng-content

- **ng-content** → A directive serving as a hook in the component to mark the place where Angular should add any content between the opening and closing tags of a parent component

Understanding the Component Lifecycle

- **ngOnChanges** → Called at the start and after a bound input (**@input**) property changes
- **ngOnInit** → Called once the component is initialized and after the constructor
- **ngDoCheck** → Called during every change detection run (checks with every event)
- **ngAfterContentInit** → Called after content (ng-content) has been projected into view
- **ngAfterContentChecked** → Called every time the projected content has been checked
- **ngAfterViewInit** → Called after the component's view (and child views) have been initialized
- **ngAfterViewChecked** → Called every time the view (and child views) have been checked
- **ngOnDestroy** → Called once the component is about to be destroyed

Seeing Lifecycle Hooks in Action

- It's good practice to implement used lifecycle hooks
- **ngChanges** Argument →Of type **SimpleChanges**

Lifecycle Hooks and Template Access

- We can't immediately access the value of certain elements if data is asynchronous, or jumping between components
    - This makes **ngOnInit** a no-go in this front
    - Instead, we use **ngAfterViewInit** as it checks later in the lifecycle

Getting Access to ng-content with @ContentChild

- **@ContentChild** allows us to access to content stored in another component, but then passed on via **ng-content**