Class 14 - Components & Databinding Deep Dive && Course Project

Class 14 Course Content

Lesson Outline

Today we will learn more about:

1. Event-binding.
2. Property-binding.
3. Sending data from component to component. (Parent to child and child to parent.)
4. Writing functions.
5. Taking in data using `@Input()` and outputting data using `@Output()`.
6. Conditionally displaying data based on a variable's state.

Lesson Notes (Review)

- **Component:** A *component* is a section or feature of your application. Every component has its own template, style, and logic. The benefit of components is that they are reusable and controllable.
- **Data-Binding:** *Data-Binding* is how we automatically update our pages template when our application state changes. It is a way to coordinate DOM object properties with data object properties.
- **View Encapsulation:** *View Encapsulation* is a build-in Angular feature where a component's CSS is locally scoped. Changing the paragraph styles inside a child component will not affect any sibling or parent components.
- **Component Lifecycle:** Angular offers many different *Lifecycle Hooks* to run logic at specific points along the *Component Lifecycle*. Every component, once instantiated, will run through a few phases; this is the "lifecycle" of the component.

Course Project Steps

STEP 1: Conditionally Rendering Pages (Using *ngIf)

shared/navigation/navigation.component.html:

- Add a `(click)="onSelectPage('bookshelf')"` listener as a property on the anchor tag with the content: "Bookshelf".
- Add a `(click)="onSelectPage('library')"` listener as a property on the anchor tag with the content: "Library".

```
<ul class="navbar-nav mr-auto mt-2 mt-lg-0">
  <li class="nav-item">
    <a class="nav-link" href="#" (click)="onSelectPage('bookshelf')"
      >Bookshelf</a
    >
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#"
(click)="onSelectPage('library')">Library</a>
  </li>
</ul>
```

shared/navigation/navigation.component.ts:

- Create a new variable `@Output() currentPage = new EventEmitter<string>()`. (Make sure to `import { EventEmitter, Output } from "@angular/core"`.)
- *Note*: The `Output()` and `EventEmitter()` are used to send variables to the parent component.
- Add the `onSelectPage(page: string)` function. This function should emit our `featureSelected` variable.

```
export class NavigationComponent implements OnInit {
  @Output() currentPage = new EventEmitter<string>();
  collapsed: boolean = true;
  show: boolean = false;

  constructor() {}

  ngOnInit(): void {}

  onSelectPage(page: string) {
    // Page Change Logic - Pass Page to Parent
    // console.log("NAV:", page);
    this.currentPage.emit(page);
  }
}
```

app.component.html file:

- We now want to listen to our outputted `currentPage` variable by using "event-binding" on the `<app-navigation>` component tag. This event should trigger a new function we will create `onNavigatePage($event)`. We use the `$event` shortcut to access the data passing through the `currentPage` variable.

```
<app-navigation (currentPage)="onNavigatePage($event)"></app-navigation>
```

app.component.ts file:

- Create the `onNavigatePage(page: string)` function. This function should update a local variable that we will use to render sections of our application conditionally.
- Create the `pageDisplayed = "bookshelf"` variable with a default value equal to "bookshelf" because that is the page we want to render when the user first renders the page.

```
pageDisplayed = "bookshelf"

onNavigatePage(page: string) {
    // console.log("APP COMP:", page)
    this.pageDisplayed = page;
}
```

app.component.html file:

- Now, we can use our `pageDisplayed` variable, paired with an `*ngIf` statement, to render our pages/features conditionally.

```
<app-bookshelf *ngIf="pageDisplayed === 'bookshelf'"></app-bookshelf>
<hr />
<app-library *ngIf="pageDisplayed === 'library'"></app-library>
```

STEP 2: Passing Bookshelf Data via Property Binding

bookshelf/book-list/book-list.component.html:

- Cut the anchor tag with everything inside of it. (This will represent one book.)
- Replace the code we cut with an `<app-book>` tag.
- Move the `*ngFor` loop to the `<app-book>` tag.

```
<div class="row mb-3">
  <div class="col-md-12">
    <app-book *ngFor="let book of myBooks"></app-book>
  </div>
</div>
<!-- . . . -->
```

shared/book/book.component.html:

- Delete all the content and paste the singular book representation we just cut from the `book-list.component.html` file.

- *Note*: To access the book data, we must pass it down from the `book-list.component.html`.

```
<a href="#" class="list-group-item clearfix">
  <div class="float-left">
    <h4 class="list-group-item-heading">{{ book.title }}</h4>
    <p class="list-group-item-text mb-0">{{ book.genre }}</p>
  </div>
  <div class="float-right">
    <img
      src="{{ book.coverImagePath }}"
      alt="{{ book.title }}"
      class="img-responsive rounded"
      style="max-height: 50px"
    />
  </div>
</a>
```

shared/book/book.component.ts:

- Add an `@Input()` decorator to take in the "book" variable coming from the "book-list.component.html" file. It should be named `book` of type `Book`. (Make sure to import)

```
@Input() book: Book;
```

bookshelf/book-list/book-list.component.html:

- Now, we can bind to the book variable in the `book.component.ts` file and pass down the singular `book` variable data.

```
<app-book *ngFor="let bookEl of myBooks" [book]="bookEl"></app-book>
```

STEP 3: Passing Data via Event & Property Binding

shared/book/book.component.html file:

- Create a click listener on the book anchor tag that fires a `onBookSelected()` function.

```
<a href="#" class="list-group-item clearfix" (click)="onBookSelected()">
  <!-- . . . -->
</a>
```

shared/book/book.component.ts file:

- Add the `onBookSelected()` function that updates a local `bookSelected` EventEmitter output variable. Doing this will allow us to "listen" to the current `bookSelected` coming from the parent component.

```
// . . .
@Output() bookSelected = new EventEmitter<void>();

// . . .

onBookSelected() {
    // Tell App that someone clicked on a book!
    this.bookSelected.emit();
}
```

bookshelf/book-list/book-list.component.html:

- Create an event-binding for the `onBookSelected()` function and run a new local function inside the `book-list.component.html` file.

```
<app-book
  *ngFor="let bookEl of myBooks"
  [book]="bookEl"
  (bookSelected)="handleBookSelected(bookEl)"
></app-book>
```

bookshelf/book-list/book-list.component.ts:

- Create the `handleBookSelected(book: Book)` function. This function should update a local output variable to emit the new book that was selected.

```
export class BookListComponent {
  @Output() currentSelectedBook = new EventEmitter<Book>();

  // . . .

  handleBookSelected(book: Book) {
    // console.log('BOOK:', book);
    this.currentSelectedBook.emit(book);
  }
}
```

bookshelf/bookshelf.component.html file:

- Create an event listener for the variable the `handleBookSelected()` function changes, which we defined in the `book-list.component.ts` file. This listener should bind to a local variable set to the book emitted from that `@Output`.

```
<app-book-list (currentSelectedBook)="selectedBook=$event"></app-book-
list>
```

bookshelf/bookshelf.component.ts file:

- Create the `selectedBook: Book` variable.

```
selectedBook: Book;
```

STEP 4: Conditionally Displaying Components

bookshelf/bookshelf.component.html file:

- Add an `*ngIf` directive on the `<app-book-details>` tag to only render if we have a `selectedBook`.
- Use an `<ng-template>` to render some text if we do no't have a `selectedBook`. (This is one way we can simulate "if/then" statements in Angular HTML files.)
- Bind to a variable `book` that we will create in the `book-details.component.ts` file, and pass down the `selectedBook`.

```
<!-- . . . -->
<div class="col-md-5">
  <app-book-details
    *ngIf="selectedBook; else infoText"
    [book]="selectedBook"
  ></app-book-details>
  <ng-template #infoText><p>Please select a Book!</p></ng-template>
</div>
```

bookshelf/book-details/book-details.component.ts:

- Add an `@Input() book: Book` decorator because we expect to get a book passed down to display.

```
@Input() book: Book;
```

bookshelf/book-details/book-details.component.html:

- Use the data we have stored in the local `book` variable to show more details for a specific book.

```
<div class="row">
  <div class="col-md-12">
```

```
    <h2>{{ book.title }}</h2>
  </div>
</div>

<div class="row">
  <div class="col-md-12">
    <h3>{{ book.author }}</h3>
  </div>
</div>

<div class="row">
  <div class="col-md-12">
    <p>{{ book.genre }}</p>
  </div>
</div>

<div class="row">
  <div class="col-md-12">
    <img
      [src]="book.coverImagePath"
      [alt]="book.title"
      class="img-responsive"
    />
  </div>
</div>
<!-- . . . -->
```

- Test the application and walk through the logic one more time.!

Additional Notes

Class Exercise

1. Create a new Angular application.
2. Generate three components using the CLI:
    - `order-dashboard`: a component that displays all customer orders.
    - `first-five-orders`: a component that displays content for the first five orders.
    - `all-other-orders`: a component that displays content for any order that isn't in the first five group.
3. The `OrderDashboardComponent` should contain a list of all orders, a button that starts the workday, and a button that ends the workday.
    - When the workday begins, a new order should be created every 2 seconds. (Orders should be an incrementing number starting from 1.)
    - When the workday ends, no more orders should be placed.

4. The `FirstFiveOrdersComponent` & the `AllOtherOrdersComponent` should be styled differently using colors, sizes, and content.
    - `FirstFiveOrdersComponent` should only show the first five orders.
    - `AllOtherOrdersComponent` should display all orders past the first five.
5. Publish your project to GitHub!

Bonus: Create a third component, `lottery-winning-order`, with a gold background and display this component every seventh order.

Component Lifecycle

ngOnChanges(changes: Simple Changes): Called after a bound input property changes.

ngOnInit(): Called once the component is initialized.

ngDoCheck(): Called during every change detection run.

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.

Resources

- Angular Guide - View Encapsulation
- Angular Guide - Component Lifecycle