Class 16 - Directives Deep Dive, Using Services and Dependency Injection && Course Project

Class 16 Course Content

Lesson Outline

Today we will learn:

1. The difference between attribute & structural directives.
2. How to implement fundamental attributes & structural directives.
3. How to use the Renderer2 Package.
4. How to react to user events in custom directives.
5. How to build custom directives.
6. How to set up Angular Services.
7. How to use Services for cross-component communication.
8. How to "Inject" Services into other Services

Furthermore, we will accomplish:

1. Creating & Updating our app to use the Bookshelf Service.
2. Creating & Updating our app to use the Library Service.

Lesson Notes

- **Directives:** A *directive* is how we give instructions to the DOM to control certain elements in Angular.
- **Attribute Directives:** An *attribute directive* is what we use to change the behavior or appearance of *elements*. (Built-in Structural Directives: `ngClass`, `ngStlye`, `ngModel`).
- **Structural Directives:** A *structural directive* is what we use to change the *DOM* layout by adding, removing, or altering elements. (Built-in Structural Directives: `ngIf`, `ngFor`, `ngSwitch`).
- **Services:** A *Service* in the context of Angular is a centralized store where you can place code so multiple other components can use it. Services allow our app to have better "communication". (This drastically reduces the complexity of the application state because we no longer need to pass a variable through two or more levels of folders using `@Input()` and `@Output()`.)
- **Dependency Injection:** *Dependency Injection* is when a class uses code from another class or service instead of writing it locally. You "inject" the code from one part of your app into another.
- **D.R.Y. Code:** *D.R.Y. Code* stands for "Don't Repeat Yourself". This meta-programming philosophy advises you to abstract away any logic you use in multiple components to a shared place.

Directive Project Steps

STEP 1: Walk Through the Starting App

- *Typescript*: We have an array that carries all our transactions and a boolean variable called showIncome that we can toggle in our HTML. We can display different content based on this variable as well.
- *HTML*: We have a container with a title, a toggle button, and a list. The button changes our "showIncome" variable, and "showIncome" changes the text inside the button. Our list will eventually display our positive transactions and negative transactions.

STEP 2: The Structural Directions (**ngIf && *ngFor*)

app.component.ts file:

- Split our transactions array into two separate arrays.

```
incomeList = [100, 50, 400];
expenseList = [100, 75];
```

app.component.html file:

- *Goal*: Our goal is to loop through each list respectively and only display one at a time.
- First, add an `*ngIf` directive on both the "Income Items" div and the "Expense Items" div.
- We also want to add an `*ngFor` directive on each `<li>` to loop through either the "incomeList" or "expenseList" and display every transaction.

```
<!-- Income Items -->
<div *ngIf="showIncome">
  <li class="list-group-item" *ngFor="let income of incomeList">
    +{{ income }}
  </li>
</div>

<!-- Expense Items -->
<div *ngIf="!showIncome">
  <li class="list-group-item" *ngFor="let expense of expenseList">
    -{{ expense }}
  </li>
</div>
```

STEP 3: Basic Attribute Directive (*ngClass*)

app.component.css file:

- Add a class for income list items and one for expense list items.

```
.income,
.expense {
  margin: 8px;
}

.income {
  background-color: darkgreen;
  color: white;
}

.expense {
  background-color: crimson;
  color: white;
}
```

app.component.html file:

- Add the `[ngClass]` directive on each list item to display the proper class.

```
<li [ngClass]="{ income: showIncome }"></li>
. . .
<li [ngClass]="{ expense: !showIncome }"></li>
```

STEP 4: Building Our Own Directive

directives/basic-border.directive.ts:

- Create a "directives" folder and a file inside called: `basic-border.directive.ts`
- Export your class, create the directive decorator config, implement `OnInit`, create logic.
- Inform our App we created a new directive by adding it to `app.modules` declarations array.

```
import { Directive, OnInit, ElementRef } from "@angular/core";

@Directive({
  // How we use our Directive
  selector: "[appBasicBorder]",
})
export class BasicBorderDirective implements OnInit {
  // Getting access to the element we put the directive on
  constructor(private elementRef: ElementRef) {}

  ngOnInit(): void {
    // Changing the styles on our element
    this.elementRef.nativeElement.style.border = "4px solid black";
  }
}
```

- Add `appBasicBorder` to both list items: `<li appBasicBorder . . .>`

STEP 5: A Better Way to Build Custom Directives

Terminal:

- Use the Angular CLI to generate a new directive: `ng g d directives/optimal-border`

directives/optimal-border.directive.ts file:

- Inject the Renderer2 Package & elementRef, implement onInit, write your logic.

```
import { Directive, OnInit, Renderer2, ElementRef } from "@angular/core";

@Directive({
  selector: "[appOptimalBorder]",
})
export class OptimalBorderDirective implements OnInit {
  // Inject the Renderer2 package into our directive and an elementRef
  constructor(private renderer: Renderer2, private elementRef: ElementRef)
{}

  ngOnInit(): void {
    // All Methods Available on Renderer2 Package:
    // https://angular.io/api/core/Renderer2
    this.renderer.setStyle(
      this.elementRef.nativeElement,
      "border",
      "4px solid gold"
    );
  }
}

// This is a better approach because it makes sure we have access to the
DOM first before running.
// Our previous approach would cause errors when using, lets say, a
service worker... and that would be difficult to debug!
```

- Add the "appOptimalBorder" to your list items.

STEP 6: Applying Directives on User Events

directives/optimal-border.directive.ts file:

- Create a new `@HostListener` for a "mouseenter" event and paste the code we used in `ngOnInit`.
- Copy your "mouseenter" listener and create a "mouseleave" listener.

- Set the border to be the defaultBorder on `ngOnInit()`

```
export class OptimalBorderDirective {
  // Inject the Renderer2 package into our directive and an elementRef
  constructor(private renderer: Renderer2, private elementRef: ElementRef)
{}

  ngOnInit(): void {
    this.renderer.setStyle(
      this.elementRef.nativeElement,
      "border",
      this.defaultBorder
    );
  }

  // Triggers when someone hovers over our element
  @HostListener("mouseenter") mouseover() {
    this.renderer.setStyle(
      this.elementRef.nativeElement,
      "border",
      "4px solid gold"
    );
  }

  // Triggers after the cursor leaves our element
  @HostListener("mouseleave") mouseleave() {
    this.renderer.setStyle(
      this.elementRef.nativeElement,
      "border",
      "4px solid transparent"
    );
  }
}
```

STEP 7: Binding to Custom Directives

directives/optimal-border.directive.ts file:

- Create two new `@Input()` declarations at the top for "defaultBorder" and "customBorder"
- Change both host listeners to use their respective `@Input` variable.

```
export class OptimalBorderDirective {
  // Binding to Custom Directives
  @Input() defaultBorder: string = '4px solid white'
  @Input() customBorder: string = '4px solid gold'

  . . .
```

```
  // Triggers when someone hovers over our element
  @HostListener('mouseenter') mouseover() {
    this.renderer.setStyle(
      this.elementRef.nativeElement,
      'border',
      this.customBorder
    )
  }

  // Triggers after the cursor leaves our element
  @HostListener('mouseleave') mouseleave() {
    this.renderer.setStyle(
      this.elementRef.nativeElement,
      'border',
      this.defaultBorder
    )
  }
}
```

app.component.html file:

- Now, we can change/customize our border within our HTML.
- We do this by binding the the "defaultBorder" and "customBorder" properties and setting them to the string we desire.

```
<li
  ...
  appOptimalBorder
  [defaultBorder]="'4px solid purple'"
  [customBorder]="'4px solid aqua'"
></li>
...
<li
  ...
  appOptimalBorder
  [defaultBorder]="'4px solid orange'"
  [customBorder]="'4px solid brown'"
></li>
```

STEP 8: The ngSwitch Structural Directive

app.component.ts file:

- Create a variable `fundraisingGoal = 1000`

app.component.html file:

- Create your switch statement.

```
<!-- Fundraising Switch Case -->
<div [ngSwitch]="fundraisingGoal">
  <p *ngSwitchCase="1000">We want to raise $1,000</p>
  <p *ngSwitchCase="100000">We will raise over $100,000!</p>
  <p *ngSwitchCase="1000000">I must raise $1,000,000 to live another day.
</p>
  <p *ngSwitchDefault>We are trying to raise some money.</p>
</div>
```

Course Project Steps

STEP 1: Creating Dropdown Directive

shared/directives/dropdown.directive.ts:

- Inside the shared folder, create a folder called "directives" and add a file called `dropdown.directive.ts` inside.
- Add your class, decorator, and logic.
- Import this directive in the app.module declarations array.

```
import {
  Directive,
  HostListener,
  HostBinding,
  ElementRef,
  Renderer2,
} from "@angular/core";

@Directive({
  selector: "[appDropdown]",
})
export class DropdownDirective {
  // Inject packages
  constructor(private elementRef: ElementRef, private renderer: Renderer2)
{}

  // When "isOpen" switches to true this will be added and when it's
false, it will be removed
  @HostBinding("class.show") isOpen = false;

  // Click Listener to toggle.
  @HostListener("click") toggleOpen() {
    // Change our "isOpen" variable to the opposite of what it currently
is.
    this.isOpen = !this.isOpen;

    // Grab the dropdown-menu div
```

```
      let dropdownList =
        this.elementRef.nativeElement.querySelector(".dropdown-menu");

      if (this.isOpen) {
        // If "isOpen" is true => ADD the class "show" to our dropdownList
        this.renderer.addClass(dropdownList, "show");
      } else {
        // If "isOpen" is false => REMOVE the class "show" from our
  dropdownList
        this.renderer.removeClass(dropdownList, "show");
      }
    }
  }
```

STEP 2: Implementing our Dropdown Directive

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

- Find the button that says "Edit Book" and replace it with a dropdown div. Bootstrap Dropdowns.
- Add the "appDropdown" directive to the dropdown div.
- Update the text to Edit Book, Update Book, Delete Book. (Also change the button to btn-primary)

```
<div class="row">
  <div class="col-md-12">
    <div class="dropdown" appDropdown>
      <button
        class="btn btn-primary dropdown-toggle"
        type="button"
        id="dropdownMenuButton"
        data-toggle="dropdown"
        role="button"
        aria-haspopup="true"
        aria-expanded="false"
      >
        Edit Book
      </button>
      <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">
        <a class="dropdown-item">Update Book</a>
        <a class="dropdown-item">Delete Book</a>
      </div>
    </div>
  </div>
</div>
```

Course Project Steps (Services)

STEP 1: Creating Our Main Services

bookshelf folder:

- Create a new file in the "bookshelf" folder titled "bookshelf.service.ts".
- Create and export the "BookshelfService" class.
- Inject the service in the "root" of our application.

```
import { Injectable } from "@angular/core";

@Injectable({
  providedIn: "root",
})
export class BookshelfService {}
```

library folder:

- Create a new file in the "library" folded titled "library.service.ts".
- Create and export the "LibraryService" class.
- Inject the service in the "root" of our application.

```
import { Injectable } from "@angular/core";

@Injectable({
  providedIn: "root",
})
export class LibraryService {}
```

STEP 2: Adding Functionality to the Bookshelf Service

- *Goal*: We want to create a "centralized" location that holds our array of books, & we want functions to access and update that array safely.

bookshelf/bookshelf.service.ts file:

- Copy the "myBooks" array from the "bookshelf/book-list.component.ts" file and paste it into the recipe service. (Make sure to import the "Book" model.)

```
// Data sources should be IMMUTABLE!
private myBooks: Book[] = [
    new Book(
        'Book of Testing',
        'Will Wilder',
        'Mystery',
        'https://source.unsplash.com/50x50/?mystery,book'
    ),
```

```
    new Book(
        'Testing Title 2',
        'Nolan Hovis',
        'Science',
        'https://source.unsplash.com/50x50/?science,book'
    ),
    new Book(
        'Fantasy Test',
        'German Cruz',
        'Non-Fiction',
        'https://source.unsplash.com/50x50/?fantasy,book'
    ),
    new Book(
        'Fantasy Test',
        'Lex Pryor',
        'Math',
        'https://source.unsplash.com/50x50/?math,book'
    ),
];
```

- Make the array a private variable and add a `getBooks()` method to access to this array from outside of the service. This is a much safer approach!
- Create a `saveBook()` method that pushes a new book to the array.

```
// . . .
// Read
getBooks() {
    return this.myBooks.slice();
}

// Create
saveBook(book: Book) {
    this.myBooks.push(book);
}
```

- Create the `removeBook(idx: number)` method that takes in an index and removes it from the array.

```
// . . .
// Delete
removeBook(idx: number) {
    if (idx !=== -1) {
        // We have a book at that index
        this.myBooks.splice(idx, 1)
    }
}
```

- Add a "bookSelected" variable that emits an event when a book is selected.
- Add a "bookListChanged" emitter that fires whenever we update our array.

```
bookSelected = new EventEmitter<Book>();
bookListChanged = new EventEmitter<Book[]>();
```

- Now, we want to emit that event after pushing to our book array on the "saveBook" and "removeBook".

```
saveBook(book: Book) {
    this.myBooks.push(book)
    this.bookListChanged.emit(this.myBooks.slice())
}

removeBook(idx: number) {
    if (idx !=== -1) {
        // We have a book at that index
        this.myBooks.spice(idx, 1)
        this.bookListChanged.emit(this.myBooks.slice());
    }
}
```

STEP 3: Using the Bookshelf Service

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

- Inject our new service into the constructor.
- Remove all the Books inside the local "myBooks" array.
- Inside the "ngOnInit()" function, set the local "myBooks" array equal to our "bookshelf.service.ts" files "getBooks()" method. Also add a subscription to the "bookListChanged" emitter.
- Remove the "handleBookSelected()" function.
- Create an "onRemoveBook(idx)" function that removes a book from our service!

```
export class BookListComponent implements OnInit {
  @Input() book: Book;
  myBooks: Book[] = [];

  constructor(private bookshelfService: BookshelfService) {}

  ngOnInit(): void {
    // Use the Service to set local "myBooks" array to Service/Global
"myBooks" array
```

```
    this.myBooks = this.bookshelfService.getBooks();
    // Listen for changes on the global "myBooks" array and update the
local version
    this.bookshelfService.bookListChanged.subscribe((books: Book[]) => {
      this.myBooks = books;
    });
  }

  onRemoveBook(idx) {
    this.bookshelfService.removeBook(idx);
  }
}
```

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

- Move the `*ngFor` to the column div and add the "index" as well.
- Create a button below the `<app-book>` tag that is a "-" sign and calls the `onRemoveBook(i)` method.

```
<div class="row mb-3">
  <div class="col-md-12" *ngFor="let bookElement of myBooks; let i =
index">
    <app-book [book]="bookElement"></app-book>
    <button
      class="float-right"
      style="border: none; font-size: 16px"
      (click)="onRemoveBook(i)"
    >
      &minus;
    </button>
  </div>
</div>
<!-- . . . -->
```

bookshelf/bookshelf.component.ts file:

- Inject our new service into the constructor.
- Inside the "ngOnInit()" function, subscribe to the "bookshelfService" "bookSelected" emitter and set the local "selectedBook" variable equal to that.

```
export class BookshelfComponent implements OnInit {
  selectedBook: Book;

  constructor(private bookshelfService: BookshelfService) {}

  ngOnInit(): void {
    // Subscribe to the bookshelfService to get all the global updates
```

```
  inside this component
      this.bookshelfService.bookSelected.subscribe((book: Book) => {
        this.selectedBook = book;
      });
    }
  }
```

bookshelf/bookshelf.component.html file:

- Remove the click bindings on the `<app-book-list>` tag & `<app-book-details>`.

```
<div class="row justify-content-between">
  <!-- Left Side - Book List -->
  <div class="col-md-6">
    <h1>My Saved Books</h1>
    <app-book-list></app-book-list>
  </div>

  <!-- Right Side - Book Details -->
  <div class="col-md-6">
    <app-book-details
      *ngIf="selectedBook; else infoText"
      [book]="selectedBook"
    ></app-book-details>
    <ng-template #infoText><p>Please select a book!</p></ng-template>
  </div>
</div>
```

bookshelf/shared/book.component.ts:

- Inject the "bookshelfService" and when the "onBookSelected()" function runs, emit the currently selected book.

```
export class BookComponent implements OnInit {
  @Input() book: Book;

  constructor(private bookshelfService: BookshelfService) {}

  ngOnInit(): void {}

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

STEP 4: Adding Functionality to the Library Service

library/library.service.ts file:

- Copy the "allBooks" array from the *library/book-results.component.ts* file. (Delete the books inside the BookResultsComponent array.) Make the variable private in the service file.
- Add a new function "getBooks()" that returns a new copy of our "allBooks" array.
- Create an event emitter to signal when the "allBooks" array changes. We will use this later.

```
export class LibraryService {
  bookListChanged = new EventEmitter<Book[]>();

  private allBooks: Book[] = [
    new Book(
      "API Book 1",
      "Will Wilder",
      "Mystery",
      "https://source.unsplash.com/50x50/?mystery,book"
    ),
    new Book(
      "API Book 2",
      "Nolan Hovis",
      "Non-Fiction",
      "https://source.unsplash.com/50x50/?serious,book"
    ),
    new Book(
      "API Book 3",
      "German Cruz",
      "Mystery",
      "https://source.unsplash.com/50x50/?mystery,book"
    ),
    new Book(
      "API Book 4",
      "Lex Pryor",
      "Non-Fiction",
      "https://source.unsplash.com/50x50/?serious,book"
    ),
  ];

  getBooks() {
    return this.allBooks.slice();
  }
}
```

library/book-results/book-results.component.ts:

```
export class BookResultsComponent implements OnInit {
  allBooks: Book[] = [];

  constructor(private libraryService: LibraryService) {}
```

```
  ngOnInit(): void {
    this.allBooks = this.libraryService.getBooks();
  }
}
```

STEP 5: Sending Data from Library to Bookshelf

library/book-results/book-results.component.ts:

- Inject the "bookshelfService" into this component to add a book to that global array.
- Create the `onSaveBook(book: Book)` function that will use the "bookshelfService" to save the current book.

```
export class BookResultsComponent implements OnInit {
  allBooks: Book[] = [];

  constructor(
    private bookshelfService: BookshelfService,
    private libraryService: LibraryService
  ) {}

  ngOnInit(): void {
    this.allBooks = this.libraryService.getBooks();
  }

  onSaveBook(book: Book) {
    return this.bookshelfService.saveBook(book);
  }
}
```

library/book-results/book-results.component.html:

- Remove the anchor tag and replace it with our book component.
- Move the "*ngFor" to the column div.
- Bind to the `[book]="bookEl"` variable to pass it through.
- Create a button that calls our `onSaveBook(bookEl)` function, passing in the current bookEl

```
<div class="row mb-3">
  <div class="col-md-9" *ngFor="let bookEl of allBooks">
    <app-book [book]="bookEl"></app-book>
    <button
      class="float-right"
      style="border: none; font-size: 16px"
      (click)="onSaveBook(bookEl)"
    >
```

```
        &plus;
      </button>
    </div>
  </div>
```

Additional Notes

Class Exercise

- Demo Video

1. Create a new Angular application.
2. Generate two components using the CLI:
    - "navbar": a component that displays an input box and a search button.
    - "search-results": a component that displays a list of your search history.
3. The "NavbarComponent" should contain an input and a button.
    - The input should use two-way-binding to update a variable in your typescript file.
    - The button should have a click listener that runs a function that adds the currentSearchTerm to an array of searches.
4. The "SearchResultsComponent" should loop through the searchHistory array and display all the searches you have previously inputed.
    - Display the text of the search on its own line
5. Create a new Directive called "RandomBGColorDirective".
    - This directive should take in an elementRef and give it a random background color.
6. Publish your project to GitHub!

Bonus: Listen for a hover event inside your "RandomBgColorDirective" that updates the background color of that element to a new color.

Resources

- Angular Docs - Attribute Directives
- Angular Docs - Structural Directives
- Blog Article - What are 'Directives' in Angular
- Angular Docs - Intro to Services & Dependency Injection
- Blog Article - Dependency Injection in Angular