Class 24 - Dynamic Components && Angular Modules + Optimizing Angular Apps

Class 24 Course Content

Lesson Outline

Today we will learn:

1. **What Dynamic Components are**
2. **How to create Dynamic Components**
3. **What Modules are in Angular**
4. **How to use Code-Splitting**
5. **How to utilize Lazy-Loading of Routes & Components**
6. **How to Preload your Components**
7. **A new way to load Services**
8. **About Ahead-of-Time-Compliation**

Lesson Notes

- **Dynamic Components:** *Dynamic Components* are components whose location isn't defined at buildtime and are loaded programmatically through code.
- **Modules:** In Angular, *Modules* are a great way to organize an application and extend it's capabilities. They are cohesive blocks of functionality comprised of components, directives, and pipes to focus on a specific feature or area of an application.
- **Code-Splitting:** *Code-Splitting* is the process of splitting code "chunks" into smaller bundles that can be loaded on demand or in parallel.
- **Lazy-Loading:** *Lazy-Loading* is a strategy to identify non-critical resources and than load those features or assets only when needed.
- **Ahead-of-Time-Compliation:** *AOT Compliation* is the act of compiling higher-level programming languages into a lower-level language before execution of a program to reduce the amount of work need to be performed at run time.

Course Project - Dynamic Alert Component Steps

GOAL: Create a (dynamically-loaded) Modal that displays whenever we add a book on the Library page.

STEP 1: Creating the Alert Component

Terminal:

- Use the terminal to create a new component named "AlertComponent" inside the "shared" folder.

```
ng g c shared/alert
```

shared/alert/alert.component.ts file:

- Create an `@Input() alertMsg: string` "alertMsg" Input variable.

```
@Input() alertMsg: string;
```

shared/alert/alert.component.html file:

- Add an empty div with the class of "backdrop". Under that, create another div with a class of "alertBox" that has a paragraph tag displaying the "alertMsg" variable, and below that, a button to close the modal.

```
<!-- Background -->
<div class="backdrop"></div>

<!-- Content -->
<div class="alert-box">
  <p>{{ alertMsg }}</p>

  <div class="alert-box-actions">
    <button>Close</button>
  </div>
</div>
```

shared/alert/alert.component.ts file:

- Create the styles for the `.backdrop` class that adds a dark background to the whole screen.
- Create the styles for the `.alert-box` class that position the message in the center of the screen.

```
.backdrop {
  z-index: 99;
  position: fixed;
  top: 0;
  left: 0;

  width: 100vw;
  height: 100vh;

  background-color: rgba(0, 0, 0, 0.75);
}
```

```
.alert-box {
  z-index: 100;
  position: absolute;
  top: 25%;
  left: 50%;
  transform: translate(-50%, -50%);

  padding: 1em;

  background-color: #fff;
  box-shadow: 0px 8px 17px 2px rgba(0, 0, 0, 0.14), 0px 3px 14px 2px
rgba(0, 0, 0, 0.12),
    0px 5px 5px -3px rgba(0, 0, 0, 0.2);
}
```

STEP 2: Displaying the Alert Component

library/library.component.ts file:

- Add a new variable "alert" of type "string".
- Inject the "BookshelfService" inside the constructor.
- Create a new "Subscription" variable called "selectedBookSub".
- Inside `ngOnInit()` use your new subscription variable and set it equal to a subscription on the `bookshelfService.bookSelectected()` method.
- Inside the `.subscribe()` callback, set the "alert" variable equal to a template literal string that prints out some text along with the book title and author.
- Implement `ngOnDestroy()` to remove your subscription when the component is destroyed.

```
export class LibraryComponent implements OnInit, OnDestroy {
  private selectedBookSub: Subscription;
  alert: string;

  constructor(private bookshelfService: BookshelfService) {}

  ngOnInit(): void {
    this.selectedBookSub = this.bookshelfService.bookSelected.subscribe(
      (book) => {
        this.alert = `Successfully added book: ${book.title} by
${book.author} to personal bookshelf!`;
      }
    );
  }

  ngOnDestroy(): void {
    this.selectedBookSub.unsubscribe();
```

```
  }
}
```

library/library.component.html file:

- Add the `<app-alert>` tag to the bottom of the file and bind to the `[alertMsg]` property, passing in the "alert" variable.
- Add `*ngIf` to the `<app-alert>` tag to only display if we have an "alert" to display.

```
<!-- Alert -->
<app-alert [alertMsg]="alert" *ngIf="alert"></app-alert>
```

bookshelf/bookshelf.service.ts file:

- Inside the `saveBook(book: Book)` method, use the `next(book)` to emit the book to all who subscribe to this variable.
- Inside the `removeBook(book: Book)` method, use the `next(this.myBooks[idx])` to emit the book recently removed to all who subscribe to this variable.

```
saveBook(book: Book) {
    this.myBooks.push(book);
    this.bookSelected.next(book);
    this.bookListChanged.next(this.myBooks.slice());
}

// . . .

removeBook(idx: number) {
    if (idx !== -1) {
      // We found a book at that index
      this.bookSelected.next(this.myBooks[idx]);
      this.myBooks.splice(idx, 1);
      this.bookListChanged.next(this.myBooks.slice());
    }
}
```

STEP 3: Closing the Alert Component

shared/alert/alert.component.ts file:

- Create a new `@Output() closeModal` "EventEmitter" variable.
- Create a new method `onCloseModal()` that emits an event to close the alert message screen.

```

```

```
@Output() closeModal = new EventEmitter<void>();

// . . .

onCloseModal() {
    this.closeModal.emit();
}
```

shared/alert/alert.component.html file:

- Add a `(click)` listener on the close button that calls the `onCloseModal()` method.
- Add that same `(click)` listener on the "backdrop" div.

```
<!-- Background -->
<div class="backdrop" (click)="onCloseModal()"></div>

<!-- Content -->
<div class="alert-box">
  <p>{{ alertMsg }}</p>

  <div class="alert-box-actions">
    <button (click)="onCloseModal()">Close</button>
  </div>
</div>
```

library/library.component.html file:

- Bind to the "closeModal" "EventEmitter" coming from the `<app-alert>` component and set it equal to a local function called `handleCloseModal()`.

```
<!-- Alert -->
<app-alert
  [alertMsg]="alert"
  *ngIf="alert"
  [closeModal]="handleCloseModal()"
></app-alert>
```

library/library.component.ts file:

- Create the `handleCloseModal()` method that sets the "alert" variable to null.
- Inisde the `ngOnInit` "Subscription", add a `setTimeout(() => ,4000)` callback that calls our `handleCloseModal()` method after 4 seconds.

```
ngOnInit(): void {
    this.selectedBookSub = this.bookshelfService.bookSelected.subscribe(
```

```
      (book) => {
        this.alert = `Successfully added book: ${book.title} by
${book.author} to personal bookshelf!`;
        setTimeout(() => this.handleCloseModal(), 4000);
      }
    );
}

handleCloseModal() {
    this.alert = null;
}
```

Programmatic Remove Book Alert ~ (BONUS) ~

bookshelf/bookshelf.component.ts file:

- Create a new subscription variable `private bookSelectedSub: Subscription`.
- Inject the "BookshelfService" and the "ComponentFactoryResolver" into the constructor.
- Inside `ngOnInit()`, set the "bookSelectedSub" equal to a subscription on the `bookshelfService.bookSelected` emitter.
- Inside the subscription, create a varaible and set it equal to a template literal that contains a message about what book is being removed.
- Call a new method on this class called `removeBookAlert(alertMsg)`, passing in the alertMsg.
- Implement `ngOnDestroy()` and unsubscribe from the "bookSelectedSub" when the component is destroyed.
- Create the new `removeBookAlert(msg: string)` method. This method should create/resolve a new "ComponentFactory".

```
export class BookshelfComponent implements OnInit, OnDestroy {
  private selectedBookSub: Subscription;

  constructor(
    private bookshelfService: BookshelfService,
    private cmpFacResolver: ComponentFactoryResolver
  ) {}

  ngOnInit(): void {
    this.selectedBookSub = this.bookshelfService.bookSelected.subscribe(
      (book) => {
        const alertMsg = `Successfully removed ${book.title} from your
personal library.`;
        this.removeBookAlert(alertMsg);
      }
    );
  }
```

```
  ngOnDestroy(): void {
    this.selectedBookSub.unsubscribe();
  }

  removeBookAlert(msg: string) {
    const alertCmpFactory =
      this.cmpFacResolver.resolveComponentFactory(AlertComponent);
  }
}
```

shared/directives/placeholder.directive.ts file:

- Create a new directive inside the "shared/directives" folder called "placeholder.directive.ts".
- This directive should give access to a public "viewContainerRef" variable to whatever component uses the "appPlaceholder" selector.
- *Note*: Ensure that this directive is declared in the *app.module.ts* file.

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

@Directive({
  selector: "[appPlaceholder]",
})
export class PlaceholderDirective {
  constructor(public viewContainerRef: ViewContainerRef) {}
}
```

bookshelf/bookshelf.component.html file:

- Above everything else, add an `<ng-template>` tag with the "appPlaceholder" directive as an attribute.

```
<ng-template appPlaceholder></ng-template>
```

bookshelf/bookshelf.component.ts file:

- Use `@ViewChild(placeholderDirective) alertHost: PlaceholderDirective` a new "ViewChild" of type "PlaceholderDirective".
- Create a new subscription variable for closing the modal called "closeModalSub" of type "Subscription".
- Inside the `removeBookAlert()` method, use the new "alertHost" "ViewChild" variable to setup the viewContainerRef.

- Create a component using the "alertCmpFactory". Set an instance of the message equal to the message being passed in the function.
- Create a method that clears the alert.
- Subscribe to the "closeModalSub" and call the `clearAlert()` method.
- Create a `setTimeout(())` callback that runs after 3 seconds and calls the `clearAlert()` method.

```
export class BookshelfComponent implements OnInit, OnDestroy {
  @ViewChild(PlaceholderDirective) alertHost: any;
  private modalCloseSub: Subscription;

  // . . .

  removeBookAlert(msg: string) {
    // Create Component Factory
    const alertCmpFactory =
      this.cmpFacResolver.resolveComponentFactory(AlertComponent);

    // Access View Container and Clear it
    const hostViewContainerRef = this.alertHost.viewContainerRef;
    hostViewContainerRef.clear();

    // Create new Alert Component Instance and Set the Message from
Arguments
    const componentRef =
hostViewContainerRef.createComponent(alertCmpFactory);
    componentRef.instance.alertMsg = msg;

    // Clear Alert Method
    const clearAlert = () => {
      this.modalCloseSub.unsubscribe();
      hostViewContainerRef.clear();
    };

    // Close the Modal and Clear the Alert
    this.modalCloseSub = componentRef.instance.closeModal.subscribe(() =>
{
      clearAlert();
    });

    // Close Modal and Clear Alert after 3 seconds
    setTimeout(() => {
      if (this.modalCloseSub) clearAlert();
    }, 3000);
  }
}
```

Course Project - Modules & Optimizations Steps

GOAL: Make the app Faster, More Performant, and Prepared for Deployment.

STEP 1: Creating Dedicated Feature Modules

shared/shared.module.ts file:

- Create a new file called "shared.module.ts" inside the "shared" folder.
- Use the `NgModule({})` decorator and add the "declarations", "imports", and "exports" arrays.
- Import and Declare all the Components, Directives, and Modules you need for the "Shared" Features. Export the ones you need elsewhere.
- *Note*: "BrowserModule" only needs to be imported once, so use "CommonModule" in anything other than *app.module.ts*.

```
// . . .
@NgModule({
  declarations: [
    AlertComponent,
    NotificationComponent,
    PlaceholderDirective,
    DropdownDirective,
    BookComponent,
  ],
  imports: [CommonModule, RouterModule, FormsModule, ReactiveFormsModule],
  exports: [
    AlertComponent,
    PlaceholderDirective,
    DropdownDirective,
    NotificationComponent,
    CommonModule,
    BookComponent,
    FormsModule,
    ReactiveFormsModule,
  ],
})
export class SharedModule {}
```

bookshelf/bookshelf.module.ts file:

- Create a new file called "bookshelf.module.ts" inside the "bookshelf" folder.
- Use the `NgModule({})` decorator and add the "declarations", and "imports" arrays.
- Import and Declare all the Components, Directives, and Modules you need for the "Bookshelf" Feature functionality. Export the ones you need elsewhere.
- *Note*: Be sure to import the "RouterModule" to clear the error about the `<router-outlet>`.

```
// . . .
@NgModule({
  declarations: [
    BookshelfComponent,
    BookListComponent,
    BookDetailsComponent,
    BookshelfHomeComponent,
    BookshelfEditorComponent,
    SortPipe,
  ],
  imports: [SharedModule, RouterModule],
})
export class BookshelfModule {}
```

library/library.module.ts file:

- Create a new file called "library.module.ts" inside the "library" folder.
- Use the `NgModule({})` decorator and add the "declarations", and "imports" arrays.
- Import and Declare all the Components, Directives, and Modules you need for the "Library" Feature functionality. Export the ones you need elsewhere.

```
// . . .
@NgModule({
  declarations: [LibraryComponent, BookSearchComponent,
BookResultsComponent],
  imports: [SharedModule, RoutingModule],
})
export class LibraryModule {}
```

auth/auth.module.ts file:

- Create a new file called "auth.module.ts" inside the "auth" folder.
- Use the `NgModule({})` decorator and add the "declarations", and "imports" arrays.
- Import and Declare all the Components, Directives, and Modules you need for the "Auth" Feature functionality. Export the ones you need elsewhere.

```
// . . .
@NgModule({
  declarations: [AuthComponent],
  imports: [SharedModule, RoutingModule],
})
export class AuthModule {}
```

app.module.ts file:

- Remove all the declarations that are placed in other *.module* files.
- Add the "SharedModule" to the "imports" array.
- Delete all unused imports at the top of the file.
- *Note*: You may need to tempararily import the "AuthModule", "LibraryModule", and "BookshelfModule".

```
// . . .
@NgModule({
  declarations: [AppComponent, NavigationComponent],
  imports: [BrowserModule, AppRoutingModule, HttpClientModule],
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptorService,
      multi: true,
    },
  ],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

STEP 2: Creating Dedicated Routing Modules

bookshelf/bookshelf-routing.module.ts file:

- Create a new file called "bookshelf-routing.module.ts" inside the "bookshelf" folder.
- Create a routes array and place all the "bookshelf" specific paths and components.
- Use the `NgModule({})` decorator and add the "imports" and "exports" arrays.

```
//. . .
const routes: Routes = [
  {
    path: "",
    component: BookshelfComponent,
    canActivate: [AuthGaurd],
    children: [
      { path: "", component: BookshelfHomeComponent },
      { path: "new", component: BookshelfEditorComponent },
      {
        path: ":id",
        component: BookDetailsComponent,
        resolve: [BookResolverService],
      },
      {
        path: ":id/edit",
```

```
        component: BookshelfEditorComponent,
        resolve: [BookResolverService],
      },
    ],
  },
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
export class BookshelfRoutingModule {}
```

bookshelf/bookshelf.module.ts file:

- Replace "RouterModule" import with "BookshelfRoutingModule".

```
imports: [
    SharedModule,
    BookshelfRoutingModule,
],
```

library/library.module.ts file:

- Inside of the "imports" array, add the "RouterModule" and call the `forChild()` method, passing in the path and component to render.

```
// . . .
@NgModule({
  imports: [
    SharedModule,
    RouterModule.forChild([{ path: "", component: LibraryComponent }]),
  ],
})
export class LibraryModule {}
```

auth/auth.module.ts file:

- Inside of the "imports" array, add the "RouterModule" and call the `forChild()` method, passing in the path and component to render.

```
// . . .
imports: [
    SharedModule,
    RouterModule.forChild([{ path: '', component: AuthComponent }]),
],
// . . .
```

STEP 3: Lazy Loading Modules

app-routing.module.ts file:

- For each main feature: "Bookshelf", "Library", and "Auth", create a path to that route and use "loadChildren" to lazy load each feature module.
- In the "imports" array, add a second argument on the `RouterModule.forRoot()` method to configure the "preloadingStrategy".

```
import { BookshelfComponent } from "./bookshelf/bookshelf.component";
import { AuthComponent } from "./shared/auth/auth.component";
import { PreloadAllModules, RouterModule, Routes } from "@angular/router";
import { NgModule } from "@angular/core";

const appRoutes: Routes = [
  { path: "", redirectTo: "/bookshelf", pathMatch: "full" },
  {
    path: "auth",
    loadChildren: () =>
      import("./shared/auth/auth.module").then((m) => m.AuthModule),
  },
  {
    path: "bookshelf",
    loadChildren: () =>
      import("./bookshelf/bookshelf.module").then((m) =>
m.BookshelfModule),
  },
  {
    path: "library",
    loadChildren: () =>
      import("./library/library.module").then((m) => m.LibraryModule),
  },
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes, {
      preloadingStrategy: PreloadAllModules,
      initialNavigation: "enabled",
    }),
  ],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Additional Notes

Resources

- Angular Docs - Dynamic Components
- Angular Docs - NgModules
- Angular Docs - NgModule FAQ
- Angular Guide - Lazy-loading Feature Modules