CST-391 Activity 3 Guide

Contents

[Part 1: Basic Angular Components, Events, Routes, and Data Binding 2](#_Toc126840434)

[Create a Simple Application 2](#_Toc126840435)

[Simple Data Binding to Labels and Handling a Form Post 3](#_Toc126840436)

[Building a New Component with Leverage 2-Way Data Binding 4](#_Toc126840437)

[Research 6](#_Toc126840438)

[Deliverables 7](#_Toc126840439)

[Part 2: Creating a Music Application – The Front End 7](#_Toc126840440)

[Create the Music Application 7](#_Toc126840441)

[Add Mock JSON and Album Images to the Application 7](#_Toc126840442)

[Create the Object Models 8](#_Toc126840443)

[Update the Routes Module 8](#_Toc126840444)

[Add a Bootstrap NavBar 8](#_Toc126840445)

[Create a Music Service that will Implement a Façade to Access the Music Data 9](#_Toc126840446)

[Update the App Component to Navigate to the Artist List Component 10](#_Toc126840447)

[Implement the List Artist Component 10](#_Toc126840448)

[Implement the Albums List Component. 11](#_Toc126840449)

[Implement the Display Album Component 12](#_Toc126840450)

[Implement the Album Create Component. 13](#_Toc126840451)

[Research 13](#_Toc126840452)

[Submission 13](#_Toc126840453)

# Part 1: Basic Angular Components, Events, Routes, and Data Binding

**Overview**

In this activity, students will build a simple test application that implements multiple components, handles page events, and binds data to a view template.

**Execution**

Execute this assignment according to the following guidelines.

## Create a Simple Application

1. In the Terminal, navigate to the location where you want the simple application to be created and create a directory named simpleapp. Navigate to the simpleapp directory.
2. Run the following Angular CLI command m,and **select the Angular *route***and CSS options during the installation:

ng new simpleapp

1. Create a VS Code Workspace by performing the following steps:
2. Open VS Code. Close the existing Workspace if one is opened.
3. Click the App Workspace Folder hyperlink from the VS Code start screen.
4. Select the *simpleapp* directory where the Simple application was created.
5. Select the File > Save Workspace As menu options and save your Workspace so VS Code can be opened again in the future.
6. In the Terminal, navigate to the test application directory and run the following command to start the Node server. Your browser should automatically be started with the application running.

ng serve --o

1. Add Bootstrap and Popper Modules to the test application by running the following commands.

npm install bootstrap

npm install @popperjs/core

1. Open the *angular.json* file in the *root* folder and add the following as the first entry in the styles array (keep the quotes):

"node\_modules/bootstrap/dist/css/bootstrap.css"

1. In the same *angular.json* file, add the following entries in the scripts array:

"node\_modules/bootstrap/dist/js/bootstrap.bundle.js"

1. Open *app.component.html* and replace its contents with a Bootstrap grid found [here.](https://getbootstrap.com/docs/5.2/layout/grid/#stacked-to-horizontal)
2. Use Chrome Dev Tools to verify that your grid is responsive. The grid layout should change as you toggle between device emulation:

Graphical user interface, text, application

Description automatically generated

Figure 1 *Chrome's Device View*

1. Take two screenshots, small screen and large, to demonstrate the responsive grid.

## Simple Data Binding to Labels and Handling a Form Post

1. Navigate to the root directory of the application and build a new component using the following command. Inspect the App Module and note that the new Shop Module has been added and registered with the application.

ng generate component shop

1. Add a new property named *question* to the Shop Component and set its value to "What’s your name?"
2. Add a new property named *answer* to the Shop Component and set its value to "unknown".
3. In the App Module, import the following module to support reactive forms:

import { ReactiveFormsModule } from '@angular/forms';

imports: [

BrowserModule,

ReactiveFormsModule,

AppRoutingModule

],

1. Add a Form Group to the Shop Component by adding the following code:

import { FormGroup, FormControl } from '@angular/forms';

1. With the other properties:

appForm = new FormGroup({

answer: new FormControl(''),

});

1. Add a form to the Shop Component view that has a single edit control bound to the *answer* form control and submits the form to the Shop Components onSubmit event handler. This approach is using a Model Driven Form.

<form [formGroup]="appForm" (ngSubmit)="onSubmit(appForm.value)">

<div class="form-group">

<label>{{question}}</label>

<input type="text" class="form-control" formControlName="answer">

</div>

<button type="submit" class="btn btn-success">Tell Me</button>

</form> <br/>

<label>Your name is {{answer}}</label>

1. In the Shop Component, add the onSubmit event handler to save the posted form data to the *question* property and log the answer to the browser's JavaScript console (remember this is code is running as JavaScript in your browser).

Graphical user interface, text

Description automatically generated

Figure 2 *'onSubmit' in the Shop Component*

1. Add the Shop Component to the App Component view by using the Shop Component tags. Start the server.

## Building a New Component with Leverage 2-Way Data Binding

1. Navigate to the root directory of the application and build a new Component using the following command. This is the same folder where app.module.ts resides. Inspect the App Module and note that the new Info Module has been added and registered with the application.

ng generate component info

1. Add three properties to the Info Component. The first property named *quantity*, the second property named *products* as an array, and the third property named *selectectProduct*. In the ngOnInit(), initialize the *quantity, products*, and *selectedProduct* properties as desired. Following is a screenshot:

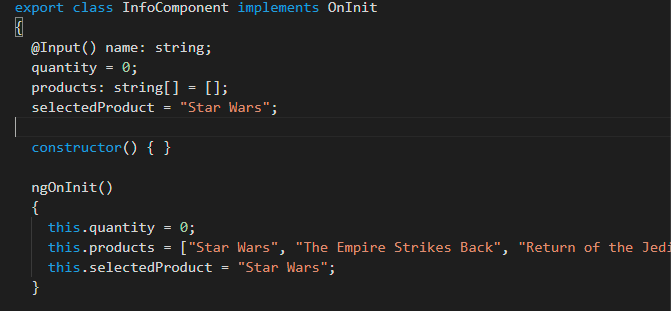


Figure 3 *InfoComponent*

1. In the application Shop Component view, add the selector tags for the Info Component below the form.
2. Start the server and validate that the default Info Component is displayed.
3. In the App Module, import the following module to support template forms:

import { ReactiveFormsModule, FormsModule } from '@angular/forms';

imports: [

BrowserModule,

ReactiveFormsModule,

FormsModule,

AppRoutingModule

],

1. Forms in Angular have two choices; we are using both here. The 'Reactive' version has more code and is more flexible. FormsModule forms work more like traditional html forms and are easier to learn but less flexible than their 'Reactive' counterparts. Traditional html forms are a common feature of web development regardless of framework. It's best to become an expert on the traditional forms approach before studying the reactive version.
2. Update the Info Component view to display a form with an input text control for the quantity, a select list for the products, a submit button, and a reset button. Note the use of the ngModel syntax, which supports 2-way data binding. Add an onSubmit event handler for the submit button and a newInfo event handler for the reset button. The complete code:

<br/>

<form (ngSubmit)="onSubmit()" #infoForm="ngForm">

<div class="form-group">

<label for="quantity">Quantity </label>

<input type="text" class="form-control" id="quantity" required

[(ngModel)]="quantity" name="quantity">

</div>

<div class="form-group">

<label for="products">Star Wars Movies</label>

<select class="form-control" id="products" required

[(ngModel)]="selectedProduct" name="products" #selected="ngModel">

<option \*ngFor="let product of products"

[value]="product">{{product}}</option>

</select>

</div>

<button type="submit" class="btn btn-success" >Buy</button>

<button type="button" class="btn btn-default"

(click)="newInfo()">Reset</button>

</form>

1. Update the Info Component with an event handler for the submit button, which displays the quantity and selected product to the JavaScript console, and an event handler for the reset button that resets the values of the Info Component. Take a screenshot of the browser and JavaScript console.

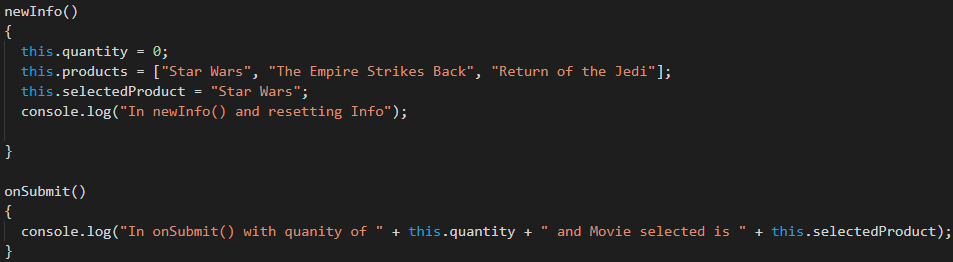


Figure 4 *Sample Handler*

1. Update the Info Component to receive as input the answer from the Shop Component and display this on the Info Component. Also update the application to dynamically render parts of the page based on if a person's name has been entered. Make the following changes to the Info Component:

Update the Shop Component view to pass in the answer property of the Shop Component to the Info Component view:

<app-info [name]="answer"></app-info>

Add the Input import and a new Input property to the Info Component:

import { Component, OnInit, Input } from '@angular/core';

@Input() name: string;

Update the Info Component view to display the name property in a Welcome message at the top of the view.

Welcome {{name}}, please place your Order

Update the Shop Component view to dynamically render the name text and info form based on if a person's name has been entered. Note the use of the ngIf directive. Take a screenshot with no name entered (the default state of the screen) and with a person's name entered.

<label \*ngIf="answer != 'unknown'">Your name is {{answer}}</label>

<app-info [name]="answer" \*ngIf="answer != 'unknown'"></app-info>

## Research

1. Describe @Input decorator used in *info.component.ts*
2. Describe [value] used in *info.component.html*
3. Describe [(ngModel)] also used in *info.component.html*

## Deliverables

In a Microsoft Word document, compile:

1. All screenshots (with captions):
   1. Two screenshots, small screen and large, to demo the responsive grid.
   2. Initial screenshot, before name is entered
   3. Screenshot after the name is entered.
   4. Screenshot of browser with dev tools with quantity and product. On the 'Buy Button'.
2. Research questions and write-up.
3. Save this document to be submitted as directed by the instructor.

# Part 2: Creating a Music Application – The Front End

**Overview**

In this activity, students will build the front end for the Music application that implements multiple components, handles page events, and binds data to a view template.

A zip file called "CST-391 Activity 3 Part 2 Resources" is provided for data and code that would take a long time to reproduce. This resource will be referred to when relevant. The folder structure of the resources will follow the project structure. In most cases, instructions to recreate the file follow directions to copy the file. These are to help you understand the code necessary for Milestone work.

**Execution**

Execute this assignment according to the following guidelines:

## Create the Music Application

1. This Angular application (*musicapp*) will eventually connect with the MusicAPI developed in an earlier activity. For Activity 3, we will use mocked data to develop most of the required UI.
2. Follow the same initial instructions from Part 1 of this activity to create an Angular music application called *musicapp.*
   1. Make sure you add routing support when you create the Angular application.
   2. Add Bootstrap and Popper dependency libraries.

## Add Mock JSON and Album Images to the Application

1. Create a directory called *data* in the *src* directory.
2. Copy the *sample-music-data.json* file from "CST-391 Activity 3 Part 2 Resources" to the *src/data* directory.
3. Copy the provided file t*ypings.d.ts* in the root of your source directory. This will allow a file of json data to be treated as a TypeScript module.
4. In the *tsconfig-app.json* file add the following line to the compiler options:

"resolveJsonModule": true,

"esModuleInterop": true,

1. For future development, create the following components:

ng generate component list-artists

ng generate component list-albums

ng generate component create-album

ng generate component display-album

ng generate component edit-album

ng generate component delete-album

## Create the Object Models

1. Create a *models* directory under the *src/app* directory. Copy the three object model classes from the Topic 1 Activity into the *src/app/models* directory
2. These models are available in the "CST-391 Activity 3 Part 2 Resources" zip folder.
3. Our client object models match exactly to the server's API object models.

## Update the Routes Module

1. Update the Routes Module in *app-routing-module.ts*, adding the routes for the all the Components to the *routes* array:

const routes: Routes = [

{ path: 'create', component: CreateAlbumComponent },

{ path: 'list-artists', component: ListArtistsComponent },

{ path: 'list-albums', component: ListAlbumsComponent },

{ path: 'display/:id', component: DisplayAlbumComponent },

{ path: 'edit/:artist/:id', component: EditAlbumComponent },

{ path: 'delete/:artist/:id', component: DeleteAlbumComponent }

];

**Note:** you will need to add imports for all components listed here. VSCode can add imports for you.

## Add a Bootstrap NavBar

*app.component.html* is included in the "CST-391 Activity 3 Part 2 Resources" zip folder.

Spend some time studying the NavBar and how it interacts with Angular.

1. Change the *title* property value to "My Music Collection."
2. Add a new *version* property and set its value to 1.0. Add a function displayVersion() that displays a JavaScript alert box with the version, which is called on the click of a button in the Bootstrap Navbar (setup below).
3. Add a function displayArtistList() that will pop an alert with a "display list here" message. Of course, this is a placeholder. This will be developed later.
4. Remove all the default content in the view except the router-outlet tags.
5. Display the application title:

<div style="text-align:center">

<h1>

Welcome to {{ title }}

</h1>

</div>

1. Add a Bootstrap NavBar with the following links. Note, do not use the href attribute in the anchor tags but use the routerLink attribute from Angular:
   1. Brand should navigate to # (i.e., self) and set to ‘My Music’.
   2. GCU Home Page should navigate to <https://www.gcu.edu/>.
   3. Create should navigate to /create.
   4. Artist List that calls the Components displayArtistList(). This will be implemented in a few steps, so for now, just log a message to the console.
   5. About that calls the Components displayVersion().

## Create a Music Service that will Implement a Façade to Access the Music Data

1. Create a *service* directory under the *src/app* directory.
2. You can find a copy of *music-service.service.ts* in the "CST-391 Activity 3 Part 2 Resources" zip folder. Follow the description below and study the code. Copy the provided file into the newly created service directory. Spend your time studying the code.
3. Navigate to the *src/app/service* directory.
4. Run the following command in the Terminal:

ng generate service music-service

1. Navigate back to the *root* directory and start the server.
2. Implement the Music Service in the service/music-service.service.ts class:
3. Import the Music data file:

import \* as exampledata from '../data/sample-music-data.json';

1. Import the object models:

import { Artist } from './../models/artists.model';

import { Album } from '../models/albums.model';

1. Create the following property:

albums: Album[] = exampledata;

1. Add a public method getArtists(): Artist[] that returns the *artists* property.
2. Add a public method *getAlbums(artist:string):Album[]* that returns the *albums* property.
3. Add a public method *getAlbum(artist:string, id:number):Album* that returns loops over the *albums* property, finds the matching *Artist* and *Album ID*, then returns the found *Album* else returns null.
4. Add a public method *createAlbum(album:Album):number* that adds the album parameter to the list of *albums*, using the array push()), and returns the Album's ID if OK, otherwise -1 if an error.
5. Add a public method *updateAlbum(album:Album):number* that returns loops over the *albums* property, finds the matching *Album ID*, then updates the Album in the *albums* property, using the array splice(), returns 0 if OK, otherwise returns -1 if the Album is not found.
6. Add a public method *deleteAlbum(id:number, artist:string):number* that returns loops over the *albums* property, finds the matching *Album ID*, then deletes the Album in the *albums* property, using the array splice(), returns 0 if OK, otherwise returns -1 if the Album is not found.

## Update the App Component to Navigate to the Artist List Component

1. In app.*component*.ts, inject a Router into the constructor with an import:

import { Router } from '@angular/router';

constructor(private router: Router)

Notice the handy constructor syntax. When you add a visibility keyword—private in this case—to a constructor's parameter, the compiler automatically creates the member variable.

1. In the displayArtistList() method, add a navigation route to the Artist List Component that passes a timestemp in the query params (this is required to get the query params to changes to the Artist List Component gets reinitialized):

this.router.navigate(['list-artists'], { queryParams: { data: new Date()} });

## Implement the List Artist Component

1. In *list-artists.component.ts*, inject the Active Route and the Music Service into the constructor:

constructor(private route: ActivatedRoute, private service: MusicServiceService) {}

1. Create a *selectedArtist* property of type Artist | null.

Create a *artists* property of type Artist[].

selectedArtist: Artist | null = null;

artists: Artist[] = [];

1. In the *ngOnInit()* method, subscribe to the queryParams changes from the route and call the Music Service getArtists() method, which will set the artists property and set the *selectedArtist* property to null.

ngOnInit()

{

this.route.queryParams.subscribe(params => {

console.log("Getting data....");

this.artists = this.service.getArtists();

this.selectedArtist = null;

});

}

1. Create a *onSelectArtist(artist: Artist)* method that updates the *selectedArtist* property, which will be used as the button click event handler in the Artist List view.

onSelectArtist(artist: Artist) {

this.selectedArtist = artist;

}

1. Display the Artists List in the view:

Text

Description automatically generated

Figure 5 *list-artists.component.html*

## Implement the Albums List Component.

1. In *list-albums.component.ts*, inject the Music Service into the constructor:

constructor(private service: MusicServiceService) {}

1. Create an *artist* property of type Artist, which takes as an input an Artist into the Component using the @Input decorator.

@Input() artist: Artist;

1. Create a *albums* property of type Album[], which is used to hold the currently list of Albums, that is initialized in ngOnInit() by calling the Music Service.

*this.albums = this.service.getAlbumsOfArtist(this.artist!.artist);*

1. Create a *selectedAlbum* property of type Album, which is used to hold the currently selected Album, that is initialized to null.

selectedAlbum: Album | null = null

1. Create a public *onSelectAlbum(album: Album)* method that updates the *selectedAlbum* property, which will be used as the button click event handler in the Album List view.

public onSelectAlbum(album: Album)

{

this.selectedAlbum = album;

}

1. Display the Albums List in the view:

Text

Description automatically generated

Figure 6 *list-albums.component.html*

## Implement the Display Album Component

1. In *display-album.component.ts*, create an *album* property of type Album, which takes as an input into the Component using the @Input decorator on the *album* property.

@Input() album: Album;

1. Display the Album in the view:

Text

Description automatically generated

Figure 7 *display-album.component.html*

## Implement the Album Create Component.

For this component, copy the code found in "create-album" folder. Make sure the UI component displays and that a new album is added to the album list when submitted. Study and understand the code.

New Artist

Add a new artist

**Optional: Complete the additional features for the application.**

These features are optional for this activity. Later activities will require these components.

1. Album Update Component.
   1. Consider reusing the *create-album* component in the *edit-album* component. Think about making *create-album* a child component of *edit-album*.
2. Implement the Album Delete Component.

## Research

1. Add complete comments to *music-service.service.ts.* Include this commented code in a Microsoft Word document or a text document. Make sure your submission reads like a code file.

## Submission

Submit all deliverables in a Word document as directed by the instructor.

**Deliverables**

1. Captioned screenshots:
   1. The initial application page
   2. GCU homepage
   3. Create Album page
   4. Artist List page showing your added album/artist
   5. About Box
2. Research questions (the fully commented code).