Angular 9

Source

- [Udemy] Angular – The Complete Guide (Link)

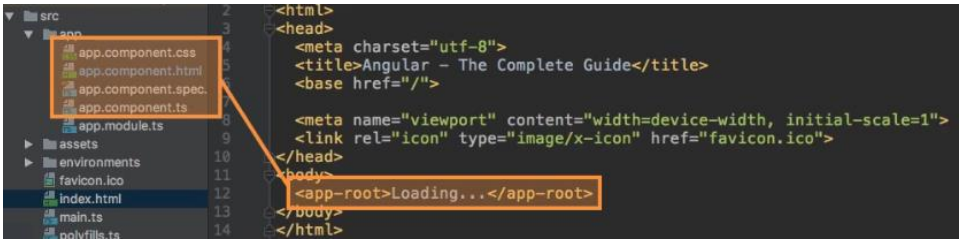
Section 1 – Getting Started

- Angular is a JavaScript Framework which allows you to create reactive single-page applications (SPAs).
    - Angular allows web applications to almost feel like native mobile applications because the JavaScript code runs on client-side rather than server-side.
    - JavaScript changes the Document Object Model (DOM) to make changes within its single page.
- Note that Angular 1 (called AngularJS) is completely different than the version we are learning here (Angular 9).
    - Angular 2 was rewritten in 2016 to fix the issues that Angular 1 had, and all newer versions stem from the Angular 2 version.
    - New versions of Angular are now released every six months.
- Angular, npm, and Node JS must first be installed on the system.
    - To create a new angular project, browse to the folder you want the project in and type `ng new my-first-app`.
    - Note: Check `ng --version` for the local project to make sure that it is the correct version. If not, something like `npm install`.
    - Note that if there is are any questions (i.e. routing) during install, choose 'no' or 'CSS'.
    - The html and css code for an angular project can be found in the src/app folder.
    - The file named `app.component.ts` is the typescript file.
- Angular is built using modules, and we need to import those modules (e.g. FormsModule) if we want to use them.
- TypeScript, which is used in Angular projects, offers types, classes, interfaces, and strong typing.
    - TypeScript does not run in the browser, so it is converted to JavaScript in the end by the Command Line Interface (CLI).
- Bootstrap is commonly used in Angular projects and is installed locally to the individual project using `npm install --save bootstrap@3`.

Section 2 – The Basics

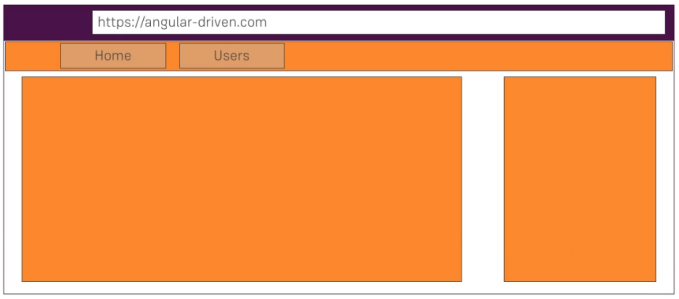
Components

- Development servers are started, or 'spun up', by the CLI when `ng serve` is entered into the command line and the angular application is hosted on this server.
- The `app.component.html` file contains the information that is displayed on the page.
    - However, it is the `index.html` file that is served by the server as the 'single-page' for the application, rather than the `app.component.html` file.
- The `main.ts` file contains a reference to the `app.module.ts` file in the `src/app` folder, which in turn contains a reference to a bootstrap array containing a list of all components that should be known to Angular at runtime.
- The index.html file contains <app-root></app-root> tags, which are replaced at runtime by the angular component noted in the `app.component.ts` typescript file's `selector` property.



    - Ultimately, Angular is a JavaScript framework that changes the DOM (HTML) at runtime.
- Components are a key feature in Angular, and we build our whole application by composing it from a couple of components that we create.

    - Components define views, which are sets of screen elements that Angular can choose among and modify according to the program logic and data.
    - The AppComponent, or root component, will hold our entire application along with any other components.
- Each component has its own template (html code), styling, and business logic.
    - The metadata for a component class associates it with a template that defines a view.
    - A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
    - It allows us to split up the application into reusable parts since components can be reused.



- Any new components that we create will not be added to the index.html page, but rather to the app.component.html page.
    - Each component should have its own folder under the src/app folder.
    - Note that it is also common practice to nest components inside of other components (i.e. create the component folders inside of other component folders).
- A component is simply a TypeScript class so that Angular can instantiate it.
    - Each new component must contain a component decorator and must import a component from the @angular/core package.
    - The component decorator must be configured with a JavaScript object containing selector and templateUrl / template properties.
    - New components must be registered in the declarations array in the app.module.ts file, and imported at the top of the file.

```
import { Component } from '@angular/core';

@Component({
    selector: 'app-server',
    templateUrl: './server.component.html'
})
export class ServerComponent {

}
```

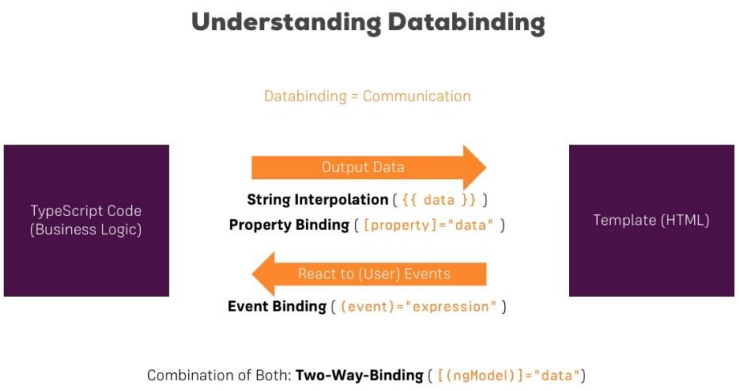
- Recall that Angular uses components to build webpages and uses modules to bundle different pieces of the app into packages.
    - Note that modules can be imported into other modules in the app.module.ts file.
- Once created, new components can be added into the html of other components (e.g. into app.component.html).
    - Note that we can reuse the same component multiple times in a component's html file.

```
<h3>I'm in the AppComponent!</h3>
<hr>
<app-server></app-server>
```

- Components can also be automatically generate by the CLI using the ng generate component <comp_name> command.
- The html code can actually be written inline in the template property of the *.component.ts file.
    - It is considered best practice to write inline code if there are three lines or less, but to specify an external template (component html) file in the templateUrl property for more than three lines.
- Styles are specified for each component in the *.component.css file or can be written inline in the *.component.html file.
    - Styles can also be defined inline in the *.component.ts TypesScript file, inside of a styles property.

Databinding

- Databinding is the communication between the TypeScript code (business logic) and the template (html).
    - Either string interpolation or property binding can be used in the html to allow for data to be passed from the TypeScript code to the front-end template (html).
    - Event binding can be used to react to user events from the front-end.
    - Two-way databinding can combine these two directions.



- For string interpolation, we include the string interpolation syntax ({{ }}) in the html, and include a variable name that is exported from the *.component.ts file.
    - The only condition for string interpolation is that an expression that point to properties be provided that can be turned into a string (i.e. numbers are ok, since they can be converted to a string).
    - Note that you can also call a method from the TypeScript export class inside the double curly braces.

```
<p>Server with ID {{ serverID }} is {{ getServerStatus() }}</p>
```

- Property binding is indicated in Angular by square brackets ([ ]) inside of the html.
    - We can bind to a native html property, such as the disabled property, by including square brackets around it.
    - Angular makes it easy to interact with the DOM at runtime.

```
<button
  class="btn btn-primary"
  [disabled]="!allowNewServer">Add Server</button>
<app-server></app-server>
<app-server></app-server>
```

- We can also bind to directives and components, although this will be explored later.
- When deciding whether to use string interpolation or property binding:
    - We would typically use string interpolation when we are displaying text in the template.
    - If we want to change some property in html, a directive, or a component, we will use property binding.
- Event binding in the html code is shown by parentheses ( ) and we can target any html properties or events (e.g. click event) in this manner.
    - Note that for events, we do not bind to onclick, for example, but rather to click.
    - We can send event information using the $event reserved variable name inside of our TypeScript function call, where the event properties can be accessed in the function call to get information about the event.
    - Using console.log() with the element that we are interested in will list all properties and events that it offers, or refer to the Mozilla Developer Network (MDN).
    - While we could include the code that we want to execute between the parentheses in the template file, this is not advised, and the code should be kept in the TypeScript file for the component.

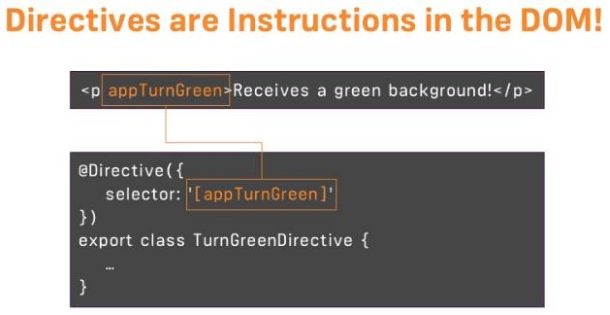
```
<input
  type="text"
  class="form-control"
  (input)="onUpdateServerName($event)">
```

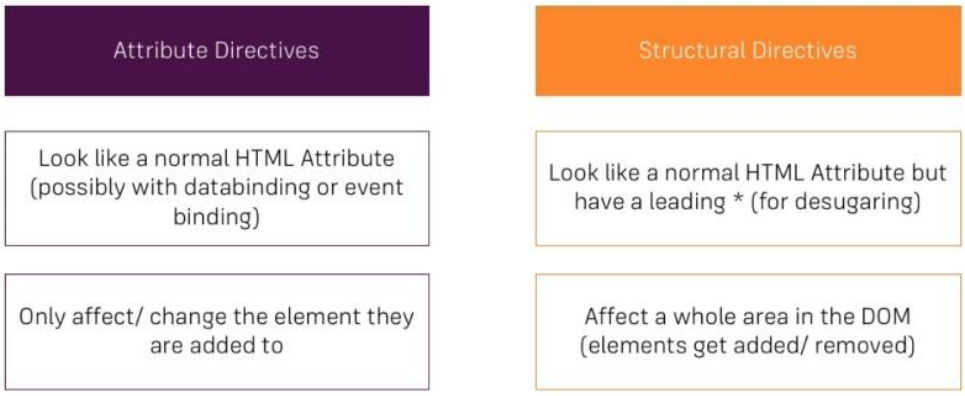
- Two-way databinding combines event binding and property binding and allows for binding in both directions (i.e. from the template to the TypeScript file, and vice versa).
    - Note that for two-way databinding to work, the ngModel directive must be enabled by adding the FormsModule to the imports[] array in the AppModule. Then an import { FormsModule } from '@angular/forms'; line must be added to the app.module.ts file.
    - Using [(ngModel)]="<propertyname>" syntax, we can have two-way binding from a control.

```
<input
  type="text"
  class="form-control"
  [(ngModel)]="serverName">
```

Directives

- Directives are instructions in the DOM.
    - We have already used directives when we used components. When we placed the selector of our component in the template, we are instructing Angular to add the content of our component template (i.e. business logic in the TypeScript code) at that place. Components are an example of a directive with a template.
    - We typically add directives with the attribute selector, rather than the CSS or the element style.





- One useful built-in structural directive example is using ngIf, which allows html elements to be conditionally added to the DOM (i.e. structurally changing the DOM).
  - Note that we must add a star (*) in front of the ngIf attribute since ngIf is a structural directive that changes the structure of our DOM by either adding or not adding the element.

```
<p *ngIf="serverCreated">Server was created, server name is {{ serverName }}</p>
```

  - One constraint is that we can have only one structural directive on any given element.
- Unlike structural directives, attribute directives don't add or remove elements. Rather, they only change properties of the element that they were used on.
  - ngStyle is an example of an attribute directive, where styles applied to elements can be dynamically updated.

```
<p [ngStyle]="{backgroundColor: getColor()}">Server with ID {{ serverID }} is {{ getServerStatus() }}</p>
```

  - The directive is added to an element in the same way as an attribute would be.
- ngClass is another attribute directive example, and it allows us to dynamically add or remove CSS classes.
  - Keep in mind that ngClass only works as intended with property binding in place in the template.

```
<p
  [ngStyle]="{backgroundColor: getColor()}"
  [ngClass]="{online: serverStatus === 'online'}">
  {{ 'Server' }} with ID {{ serverID }} is {{ getServerStatus() }}
</p>
```

- ngFor is another built-in structural directive that lets us loop through data, dynamically adding new elements at runtime.
  - Note that we can output the individual values not only in the element with the ngFor loop, but any nested elements.

```
<app-server *ngFor="let server of servers"></app-server>
```

Section 3 – Course Project: The Basics

- Models can be created as a TypeScript class, which is conceptually identical to what we have used models for in Java.
  - If the model is to be used in another file, it needs to be imported first.
  - Should the model not have any logic, a shortcut way to assign values to member variables is to simply place an accessor (e.g. public, private, etc.) in front of the parameter name in the constructor.

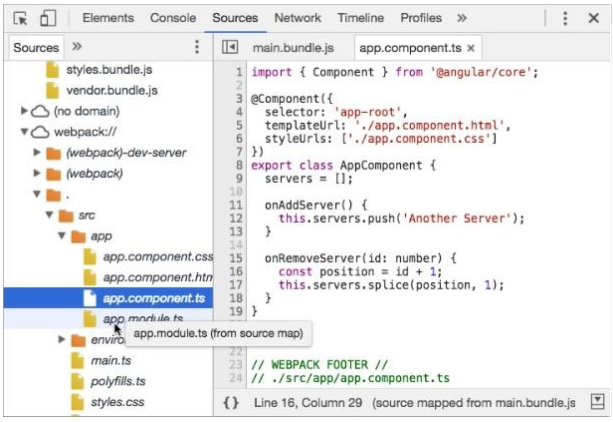
```
export class Ingredient {
  constructor(public name: string, public amount: number) { }
}
```

- The Recipe Book app created as a project for this section is an excellent code and (especially) style reference.
  - The amount of databinding is minimal, but it uses most of the concepts covered and focuses on project organization and bootstrap CSS styling.

Section 4 – Debugging

- One method of debugging errors is simply to read the message, see what file it points to, and figure out where the problem is without having explicit line numbers to point you to the problem.

- Since JavaScript files support source maps, which are maps between our JavaScript and TypeScript files, we can go to Sources in Chrome Developer Tools, choose main.js, find what roughly looks like the line that we want to inspect, click on the line number, and get taken to the actual TypeScript file where we can actually place a debug breakpoint.
    - Now when we interact with the web page, the execution will pause when it hits the debug breakpoint and we can then walk through the code and inspect variables.
    - Alternatively, we can directly access our TypeScript files as shown in the screenshot.
- Using Augery (Chrome Extension) is another option for viewing the state, properties, injector graphs, etc.



Section 5 – Components and Databinding Deep Dive

Passing Data Between Components

- We can use databinding not only on html elements, but also on properties and events within directives and components.
- By default, all properties of components are only accessible inside the components and not from outside.
    - We must be explicit about what properties we want to expose outside of the component to the world.
    - If we want to allow parent components (i.e. components hosting another component through its selector) to be able to have a one-way binding (i.e. sending data to the property) to a property in another component class, we have to use the @Input() decorator. Note that Input also must be added as in import at the top of the TypeScript file.

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

@Component({
  selector: 'app-server-element',
  templateUrl: './server-element.component.html',
  styleUrls: ['./server-element.component.css']
})
export class ServerElementComponent {
  @Input() element: { type: string, name: string, content: string };
}
```

```
<app-server-element *ngFor="let serverElement of serverElements"
        [element]="serverElement"></app-server-element>
```

- We can specify an alias for the component property inside of the input decorator, which then requires that the alias name be used to call the property outside of the component.

```
export class ServerElementComponent {
  @Input('srvElement') element: { type: string, name: string, content: string };
}
```

- If instead of sending data to a property, we want to capture data from a property in a parent component, we can use one-way binding again where we instead output the data using the @Output() decorator from the property to the parent class.
    - We can setup events in the parent component html file to capture data from a child component property. For the child to send the update property data to the event in the parent component, the child component property must emit the data (as shown in the two figures).

```
export class CockpitComponent {
  @Output() serverCreated = new EventEmitter<{serverName: string, serverContent:
string}>();
  @Output() blueprintCreated = new EventEmitter<{serverName: string, serverContent:
string}>();
  newServerName = '';
  newServerContent = '';

  onAddServer() {
    this.serverCreated.emit({
      serverName: this.newServerName,
      serverContent: this.newServerContent
    })
  }

  onAddBlueprint() {
    this.blueprintCreated.emit({
      serverName: this.newServerName,
      serverContent: this.newServerContent
    })
  }
```

    - Just like in the input decorator, we can create an alias for the event outside of the component, which will then always be required to be used anytime the event is referenced outside of its own component.

```
<app-cockpit
  (serverCreated)="onServerAdded($event)"
  (blueprintCreated)="onBlueprintAdded($event)">
</app-cockpit>
```

    - Although we can emit from a child to a parent, we cannot emit two or more levels upwards.

- Note that there are some use cases where we will not want to have this chain where data is emitted from a component to the parent component, to be send down to another child component.
    - There are times when the complexity will increase, and the chain will grow so great that this is not practical.
    - There is another approach that will be discussed later, although it is not necessarily a better approach.

Encapsulation

- Angular supports CSS encapsulation, where the CSS file for each component only applies to that component.
    - This happens in the compiled html, where different paragraphs are assigned different attribute names so that CSS styles are not applied globally to paragraphs (for example).
- We can also disable CSS encapsulation in a component, where the stylings will be instead applied globally.
    - This is done by setting the encapsulation: ViewEncapsulation.None attribute in the @Component decorator.

References

- A local reference can be placed on any html element by placing a #<name> inside of the html tag for the element.
    - This reference will hold a reference to the html element including all the properties.
    - Local references can be used anywhere in the template, but not outside of the template (e.g. not in the TypeScript code).

```
<input
  type="text"
  class="form-control"
  #serverNameInput>
<label>Server Content</label>
<input type="text" class="form-control" [(ngModel)]="newServerContent">
<br>
<button
  class="btn btn-primary"
  (click)="onAddServer(serverNameInput)">Add Server</button>
```

- If we want to access an html element in the TypeScript code for a component, we can use a ViewChild.
    - To do so, we would still set a local reference in the html as before, but this time we would set that local reference as a @ViewChild(..) in the TypeScript file.
    - We can then reference the ViewChild variable and access the native element through the reference in either the parent or child component.
    - As a warning though, we should not go the opposite way to set data in the DOM element. Use either string interpolation or property binding to do this instead.

```
@ViewChild('serverContentInput', { static: true }) serverContentInput: ElementRef;

onAddServer(nameInput: HTMLInputElement) {
  console.log(this.serverContentInput);
  this.serverCreated.emit({
    serverName: nameInput.value,
    serverContent: this.serverContentInput.nativeElement.value
  });
}
```

- One last tool that we can use is the ng-content directive, which has similar syntax to a component.
    - Ng-content is a placeholder tag for dynamic content since the tags will be replaced with content when the template is parsed to html.
    - This is called content projection since we are projecting content from the parent component into the designated child component.

```
<div
  class="panel panel-default">
  <div class="panel-heading">{{element.name}}</div>
  <div class="panel-body">
    <ng-content></ng-content>
  </div>
</div>
```

Component Lifecycle

- Recall that when angular comes across a selector for a component, it is at that point that the component is instantiated and added into the DOM.
- There are several lifecycle hooks in Angular, which are called during certain times of a component object's lifecycle.
- Keep in mind that these will not be used too often, but if needed, the Section 5 code shows when each of these are called as the application runs.
    - Keep in mind that some of these hooks such as ngOnInit() will run before the DOM has been rendered, so we may not get the information that we expect to get.

ngOnChanges	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) has been initialized
ngAfterViewChecked	Called every time the view (and child views) have been checked
ngOnDestroy	Called once the component is about to be destroyed

Section 7 – Directives Deep Dive

Custom Attribute Directives

- Custom Attribute Directives are created in their own TypeScript files named *.directive.ts.
    - Each directive should also have a unique selector, just like components.
    - If we want the directive to be added in an attribute style, we will wrap the selector name in brackets so that whenever the directive name is added to an element without square brackets, the directive is applied to that element.
    - Angular gives us access to the element that the directive is applied to by allowing us to inject the element that the directive sits on into the directive.
    - Although we can add styling to elements like this, it is not recommended to directly access your elements in this way since Angular offers a better way (below).

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

@Directive({
  selector: '[appBasicHighlight]'
})

export class BasicHighlightDirective implements OnInit {
   constructor(private elementRef: ElementRef) { }

   ngOnInit() {
     this.elementRef.nativeElement.style.backgroundColor = 'green';
   }
}
```

```
<p appBasicHighlight>Style me with basic directive!</p>
```

- Once created, we have to add our directive to the app.module.ts file as a declaration, and add it as an import.
    - Directives can also be automatically created using the ng generate directive <directiveName> command in the terminal.
- Using a renderer is a better approach for accessing the DOM, and the type of attribute directive that we should be using.
    - The reason that we should be using a renderer for any DOM manipulations is that Angular does not always have access to the DOM (e.g. when using service workers), and hence would throw an error if we tried to access an element when the DOM is not available.
    - Using a host listener with a renderer is a convenient way of reacting to events on an element.
- Rather than using a renderer, having a host binding is another way to set properties on an element from inside of a directive, as shown in the screenshot.

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

@Directive({
  selector: '[appBetterHighlight]'
})

export class BetterHighlightDirective implements OnInit {
  @Input() defaultColor: string = 'transparent';
  @Input('appBetterHighlight') highlightColor: string = 'blue';
  @HostBinding('style.backgroundColor') backgroundColor: string;

  constructor(private elRef: ElementRef, private renderer: Renderer2) { }

  ngOnInit() {
    this.backgroundColor = this.defaultColor;
  }

  @HostListener('mouseenter') mouseover(eventData: Event) {
    this.backgroundColor = this.highlightColor;
  }

  @HostListener('mouseleave') mouseleave(eventData: Event) {
    this.backgroundColor = this.defaultColor;
  }
}
```

- We can dynamically set attributes within a directive by using custom property binding.
    - To do this, we simply add properties to the directive that we can override from outside of the class using @Input().
    - If we are passing a string into an attribute directive in the html, we do not have to show square brackets around the attribute. This is commonly the case, so be aware that these custom attributes are not actually attributes for an element.

```
export class BetterHighlightDirective implements OnInit {
  @Input() defaultColor: string = 'transparent';
  @Input() highlightColor: string = 'blue';
  @HostBinding('style.backgroundColor') backgroundColor: string = this.defaultColor;
```

```
<p [appBetterHighlight]="'red'" defaultColor="yellow">Style me with basic directive!</p>
```

Custom Structural Directives

- Unlike attribute directives, structural directives require that an asterisk (*) be placed in front of the directive when added as an attribute in an element.

```
<div *appUnless="onlyOdd">
  <li
    class="list-group-item"
    [ngClass]="{odd: even % 2 !== 0}"
    [ngStyle]="{backgroundColor: even % 2 !== 0 ? 'yellow' : 'transparent'}"
    *ngFor="let even of evenNumbers">
    {{ even }}
  </li>
</div>
```

- Behind the scenes, angular desugars this star notation into a marked-up <ng-template> that surrounds the host element and its descendants.

- To create a custom structural directive, we would first create a new directive using ng generate directive <directiveName>.
  - Note that in the provided code, a template reference is created in the constructor. This allows for the code inside of the <ng-template> in the html file to be swapped out dynamically at runtime depending on the custom directive condition.
  - As the code in the if-else condition shows, if the condition is not true, the template reference code

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {
  @Input() set appUnless(condition: boolean) {
    if (!condition) {
      this.vcRef.createEmbeddedView(this.templateRef);
    } else {
      this.vcRef.clear();
    }
  }

  constructor(private templateRef: TemplateRef<any>, private vcRef: ViewContainerRef) { }
}
```

- As an aside, we can include a setter for a property using the set keyword, which will execute a method whenever a property changes.

ngSwitch

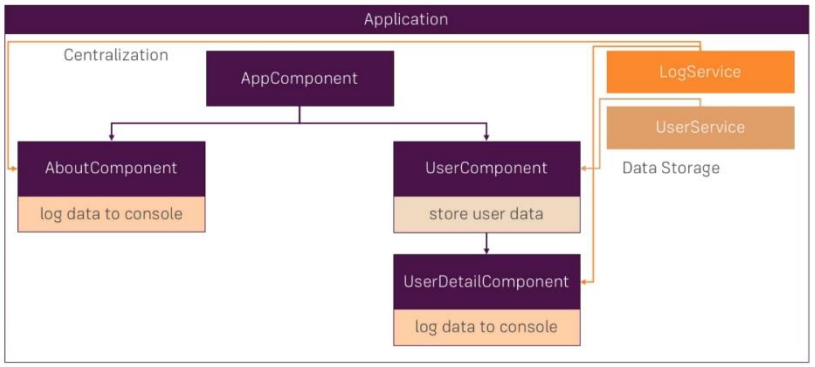
- If we find yourself using a lot of ngIf conditions, ngSwitch may be a better solution.
  - It allows you to use cases and default values, just like switch statements normally do.
  - Only those elements that meet the case conditions are rendered to the DOM.

```
<div [ngSwitch]="value">
  <p *ngSwitchCase="5">Value is 5</p>
  <p *ngSwitchCase="10">Value is 10</p>
  <p *ngSwitchCase="100">Value is 100</p>
  <p *ngSwitchDefault>Value is default</p>
</div>
```

Section 9 – Using Services & Dependency Injection

Services for Code Reuse

- Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making the code modular, reusable, and efficient.
- The use cases involving the duplication of storage and data access across the application are where services are useful.
  - A service is essentially just another class we can add to our application that serves as a central repository for data or code.



- Service files can be placed in any folder, but since they are shared, they are often placed into a shared folder (or maybe the main app).
  - Service files are named *.service.ts and are structured as a typical class without the special decorators that component or directive classes have.

```
export class LoggingService {
  logStatusChange(status: string) {
    console.log('A server status changed, new status: ' + status);
  }
}
```

- Rather than manually instantiating a service, Angular includes a dependency injector that should be used to make the service available to components.
  - Note that a dependency is something that a class depends on. For example, our class could depend on the logging service shown in the screenshot, and hence the LoggingService class would be a dependency.
  - The dependency injector automatically injects an instance of this dependency (i.e. class) into our class when we include the service in class constructor as a type for a variable being passed into the constructor.

- Note that Angular is responsible for constructing new instances of classes as these classes are found during compilation.
- One last step that we have to take for a class that uses a dependency injector is to provide the service to the class, which is done by adding a providers property inside of the @Component decorator.

- Services help considerable with the DRY (Don't Repeat Yourself) software engineering principle.

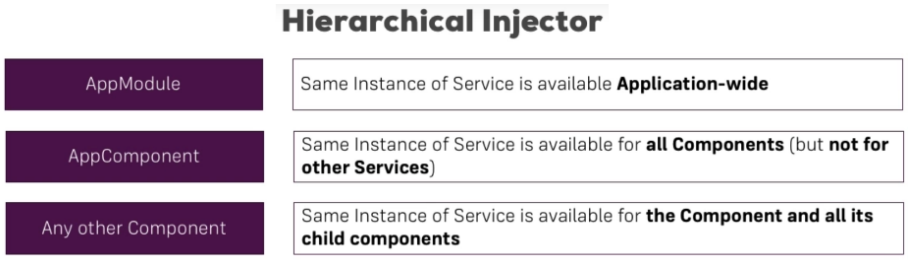
Services for Storing Data

- Storing and managing data is another typical use case for services.
  - Keep in mind that most initialization should not be done in the constructor, but rather in the ngOnInit() method for a component class.
  - Since the service is injected into a class when we include the service in the class constructor, we can then call methods within to the service and access the data members as needed.
- Keep in mind that we should consider encapsulation when creating services for storing data, where the properties should be made private with methods created for manipulating the properties.
  - Getter and setter methods should be used, just like in Java.
  - Max also suggested that we return a copy of the data, rather than the original data.

```
import { EventEmitter, Injectable } from '@angular/core';
import { LoggingService } from './logging.service';

@Injectable()
export class AccountsService {
  accounts = [
    {
      name: 'Master Account',
      status: 'active'
    },
    {
      name: 'Testaccount',
      status: 'inactive'
    },
    {
      name: 'Hidden Account',
      status: 'unknown'
    }
  ];

  statusUpdated = new EventEmitter<string>();

  constructor(private loggingService: LoggingService) { }

  addAccount(name: string, status: string) {
    this.accounts.push({ name: name, status: status });
    this.loggingService.logStatusChange(status);
  }

  updateStatus(id: number, status: string) {
    this.accounts[id].status = status;
    this.loggingService.logStatusChange(status);
  }
}
```

Injector Hierarchy

- The Angular dependency injector is a hierarchical injector, which means that if we create a service on a component in our app, the Angular framework creates an instance of the service for this component along with all its child components (including children of children).
  - This instance of the service will be the same for all the components.
  - Note that instances of services only go down the hierarchical chain of components, and not up.
- This hierarchy of service instances is intentional since not all applications should have the same instance of a service throughout the application.
  - To include an existing instance of a service in a child class then, we need to keep the instance in the constructor and keep the service import statement but will remove it from the providers array in the @Component decorator.
  - In summary, we only include the service in the providers array at the top-level component (or module) when we want to reuse the service instance in the children. The service will still be included in the constructor and imported into each child component, however.



- We can inject a service into another service in generally the same way, where we inject the service into the constructor for a service class. The service should be included at the AppModule (root module) level if we want that service to be injected into another service.
  - One difference though is that we must tell Angular that the service that is to have a new service injected, is in fact injectable. We would include the @Injectable() decorator in the service class that is to have another service injected inside of it, as shown in the screenshot above.
  - The instructor did mention that we should consider always including @Injectable() in our services to ensure forward compatibility since it won't break anything to include it.

- If we want to add a service directly into the app module without having to explicitly change that file, we can include @Injectable({providedIn: 'root'}) above the export statement in the service.

- Services overall allow for code that is cleaner, more centralized, and easier to maintain.
  - Using services means that we do not have to build complex input-output chains that we previously had built with events and properties to provide functionality and move data around an application.
- Services permit cross-component communication.
  - We can subscribe to a property event that is emitted in one component by using the subscribe() syntax in another component. This will ensure that when data in the service property changes, the subscribe method in another class will be triggered.

```
constructor(private loggingService: LoggingService, private accountsService:
AccountsService) {
  this.accountsService.statusUpdated.subscribe(
    (status: string) => alert('New Status: ' + status)
  );
}
```

```
statusUpdated = new EventEmitter<string>();
```

```
onSetTo(status: string) {
  this.accountsService.updateStatus(this.id, status);
  // this.loggingService.logStatusChange(status);
  this.accountsService.statusUpdated.emit(status);
}
```

  - Using an event emitter appears to be common, where components that use this service can emit with the property (e.g. statusUpdated in this case) and can receive using the subscribe method.

Section 11 – Changing Pages with Routing

Routing Basic Principles

- Note that the Section 12 Course Project for routing is an excellent refresher resource and should be watched as a refresher.
- Angular ships with its own router, which allows a person to change the URL in the URL bar while keeping everything within one single page.
  - The Angular router is used here, and the router needs to know which routes the front-end application has.
  - This is helpful if a user wants to navigate to a certain portion of the page, even if it is a single-page application.

```
const appRoutes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'users', component: UsersComponent, children: [
      { path: ':id/:name', component: UserComponent }
    ]
  },
  {
    path: 'servers', canActivateChild: [AuthGuard], component: ServersComponent, children: [
      { path: ':id', component: ServerComponent, resolve: { server: ServerResolver } },
      { path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }
    ]
  },
  // { path: 'not-found', component: PageNotFoundComponent },
  { path: 'not-found', component: ErrorPageComponent, data: { message: 'Page not found!' } },
  { path: '**', redirectTo: '/not-found' }
];

@NgModule({
  imports: [
    // RouterModule.forRoot(appRoutes, {useHash: true})
    RouterModule.forRoot(appRoutes)
  ],
  exports: [RouterModule]
})

export class AppRoutingModule {

}
```

- To allow for routes, const appRoutes: Routes = [...] holding an array of routes must be included in a module file, and Routes must be imported into the module.
  - Each JavaScript object in the routing array should contain a path for the page, along with a component that will be loaded when the new page is reached.
  - An empty route path is sometimes provided that could direct the user to a home page component.
  - To register these routes with the application, RouterModule.forRoot(appRoutes) must be added to the imports array in the router module.
  - Note that the order of the route items in the array is important, especially once children are added in. The items in the array should be structured so that the more specific ones are at the top so that the less specific routes do not end up routing the provided path incorrectly before the correct route receives the path.
- A <router-outlet></router-outlet> directive should be added to the position in the

```
<div class="row">
  <div class="col-xs-12 col-sm-10 col-md-8 col-sm-offset-1 col-md-offset-2">
    <router-outlet></router-outlet>
  </div>
</div>
```

app.component.html template where the Angular router should load the component for the currently selected route.

  - Routes can also be nested as children, as shown in the screenshot above. Note that additional <router-outlet></router-outlet> directives should be included in the parent router template to accommodate the parent router paths.

- We want to keep in mind that the page should not be reloaded each time that a new route is navigated to. The goal is a single-page application where server requests are kept to a minimum.

```
<li role="presentation"
    routerLinkActive="active">
  <a routerLink="servers">Servers</a>
</li>
```

  - To navigate through the application without reloading the page from the server, router links (routerLink property binding) should be placed into the component template.
  - Router link catches the click on the element that it's placed in, prevents the default action of sending a request to the server, and confirms whether a route was provided in the routing configuration.
  - Using this router link method for page navigation also allows us to keep the app state between pages.

Navigation

- Keep in mind that absolute (e.g. /servers) vs relative navigation paths (e.g. servers) are important when setting up routes, since having the wrong path can cause an error.
  - A relative path will always append to the provided path to the current path.
- As shown in the previous screenshot, the routerLinkActive directive can be included in an element which allows you to add a CSS class to an element when the link's route becomes active.
  - The [routerLinkActiveOptions]="{exact: true}" property binding ensures that classes are added only when the URL matches the link exactly.
- If we want to navigate dynamically at runtime (e.g. when a method is run), we can inject the router into our TypeScript constructor, include the import statement, and call the navigate method.

```
onReload() {
  this.router.navigate(['servers'], {relativeTo: this.route});
}
```

  - When we are calling the navigate method inside of a TypeScript class, keep in mind that the navigate method does not know which route the application is currently on (unlike routerLink, which does know the current route). We can add a relativeTo property with a value of this.route to get around this shortcoming, although we also need to inject the ActivatedRoute for this to work properly.
  - The ActivatedRoute interface provides access to information about a route associated with a component that is loaded in an outlet (i.e. the currently loaded route).

```
export class UserComponent implements OnInit, OnDestroy {
  user: { id: number, name: string };
  paramsSubscription: Subscription;

  constructor(private route: ActivatedRoute) { }

  ngOnInit() {
    this.user = {
      id: this.route.snapshot.params['id'],
      name: this.route.snapshot.params['name']
    };

    this.paramsSubscription = this.route.params.subscribe(
      (params: Params) => {
        this.user.id = params['id'];
        this.user.name = params['name'];
      }
    );
  }

  ngOnDestroy() {
    this.paramsSubscription.unsubscribe();
  }
}
```

- We can dynamically add parameters to our routes by including a colon in the route path (see first screenshot in this section).
  - Once navigated to the page through dynamic routing, we can also extract the dynamic routing content from the URL in the TypeScript file by using the route.snapshot.params[...] syntax shown in the accompanying screenshot.
- If we link to another URL that is already on the component that we are currently on, Angular will not reload the component automatically since that would come with a cost to the performance.
  - We would instead subscribe to the route.params observable, which allows us to react to an event that might happen in the future.
  - When you leave the component, the component is destroyed. It is important to note that the subscription lives on in memory even if the component is destroyed, which is something that Angular does in the background. It is good practice though to unsubscribe from an observable manually (see screenshot).

Query Parameters

- Query parameters are parameters in the URL that are separated by a question mark.
    - Multiple queries can be connected with an & symbol, or a hash symbol # can be included as well to jump to certain positions (i.e. fragments) within the page.
    - These query parameters can be set using the [queryParams] property binding syntax, where queryParams is a property of the routerLink directive.

```
<div class="list-group">
  <a
    [routerLink]="['/servers', server.id]"
    [queryParams]="{allowEdit: server.id === 3 ? '1' : '0'}"
    fragment="loading"
    href="#"
    class="list-group-item"
    *ngFor="let server of servers">
    {{ server.name }}
  </a>
</div>
```

    - The position within the page can be set using the [fragment] property binding syntax.

```
export class HomeComponent implements OnInit {

  constructor(private router: Router, private authService: AuthService) { }

  ngOnInit() {
  }

  onLoadServer(id: number) {
    // complex calculation
    this.router.navigate(['/servers', id, 'edit'], {queryParams: {allowEdit: '1'}, fragment: 'loading'});
  }
```

- We can access query params and fragments from the URL just like we did before, using the route.snapshot syntax.
    - As before, we should be aware that using the route.snapshot syntax inside the ngOnInit() method will not allow us to retrieve any changes in the URL that were made while that same component is active. Therefore, we will need to subscribe to any changes in the query parameters.

```
ngOnInit() {
  console.log(this.route.snapshot.queryParams);
  console.log(this.route.snapshot.fragment);

  this.route.queryParams
    .subscribe(
      (queryParams: Params) => {
        this.allowEdit = queryParams['allowEdit'] === '1' ? true : false;
      }
    );

  this.route.fragment.subscribe();
```

- As an aside, adding a plus-symbol (+) in front of a string return will change the string to a number.
- We can also include logic within query parameter expressions, as shown in the first screenshot under this subsection heading.
- If we want to preserve query parameters when navigating to another page, we can use the queryParamsHandling property in the router.navigate method. This can be useful for loading specific data or determining permissions.

```
onEdit() {
  this.router.navigate(['edit'], { relativeTo: this.route, queryParamsHandling: 'preserve' });
}
```

Redirecting

- If a user enters an address that we do not have routed, we need to allow for redirects to take the user to an error page.
    - Rather than routing to a component to load, we can add a route that redirects to a component. This is useful for route error handling because we can redirect wildcard path: '**' to an error page component. Note that this must be the last path in the array of routes since routes are parsed from top to bottom.

```
{ path: 'not-found', component: ErrorPageComponent, data: { message: 'Page not found!' } },
{ path: '**', redirectTo: '/not-found' }
```

    - Keep in mind that a pathMatch property exists for path routing which ensures that a path exactly matches the URL.

Route Management

- We typically don't add routes to the app.module.ts file, but rather we creating our own routing module to make our files modular and easier to maintain.
    - This could be called app-routing.module.ts or something similar. Note that modules will be covered in detail later.
    - Note that we do not duplicate the declarations found in the app module file.
    - This new routing module is then imported into the app module file.

Route Guards

- Route guards include functionality, logic, and code that is executed before a route is loaded or before leaving a route.
    - Authentication guards are typically created as their own TypeScript files.
- The canActivate interface is then implemented in the AuthGuard class, which decides if a route can be activated or not.
    - If all guards return true, navigation will continue, but if any guard returns false, activation is cancelled.
- CanActivate can return asynchronously (i.e. returning an observable or a promise) or synchronously.
    - Code that runs completely on the client would run synchronously, and code that takes some time to finish or reaches out to a server would run asynchronously.

```
@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot,
              state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return this.authService.isAuthenticated()
      .then(
        (authenticated: boolean) => {
          if (authenticated) {
            return true;
          } else {
            this.router.navigate(['/']);
          }
        }
      );
  }

  canActivateChild(route: ActivatedRouteSnapshot,
                   state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return this.canActivate(route, state);
  }
}
```

- A service called auth.service.ts could also be created to handle the authentication logic.
    - This service should be injected into the auth-guard service, which means that these services should be added as providers to the app module file.
    - The canActivate method can then be applied to paths where authentication should occur, which will also make it apply to all child routes as well. Note that canActivateChild could also be included which will protect only the child routes instead of both the whole route including child routes.
    - Including authService.login() and authService.logout() methods will ensure that the users can log in and out of the authentication service.

```
{
  path: 'servers', canActivateChild: [AuthGuard], component: ServersComponent, children: [
    { path: ':id', component: ServerComponent, resolve: { server: ServerResolver } },
    { path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }
  ]
},
```

Deactivating Routes

- The canDeactivate interface allows us to confirm with the user when they are looking to leave a route.
    - This can be a convenience to the user in case they accidently click the back button before saving changes, for instance.
- We can create a new service class called can-deactivate-guard.service.ts, or similar, which will export an interface.
    - This canDeactivate interface is implemented within the same class, as well as in any classes that implement the canComponentDeactivate interface (i.e. classes that you want to have this deactivate functionality added to).

```
import { Observable } from 'rxjs/Observable';
import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(component: CanComponentDeactivate,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return component.canDeactivate();
  }
}
```

```
canDeactivate(): Observable<boolean> | Promise<boolean> | boolean {
  if (!this.allowEdit) {
    return true;
  }
  if ((this.serverName !== this.server.name || this.serverStatus !== this.server.status) && !this.changesSaved) {
    return confirm('Do you want to discard the changes?');
  } else {
    return true;
  }
}
```

Passing Static and Dynamic Data with Routes

- We can define static data within our route paths that will be available to each page with the associated route.

```
{ path: 'not-found', component: ErrorPageComponent, data: { message: 'Page not found!' } }
```

- This can be helpful for things like error messages (i.e. page not found errors) that may need to be changed as the application progresses.
- Different components can then inject ActivatedRoute into the constructor and get access to the route's static data members.

```
this.route.data.subscribe(
  (data: Data) => {
    this.errorMessage = data['message'];
  }
);
```

- We can resolve dynamic data with the resolve interface, which basically allows us to asynchronously load data to be used in the router during navigation.

```
export class ServerResolver implements Resolve<Server> {
  constructor(private serversService: ServersService) {}

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Server> | Promise<Server> | Server {
    return this.serversService.getServer(+route.params['id']);
  }
}
```

```
{ path: ':id', component: ServerComponent, resolve: { server: ServerResolver } },
```

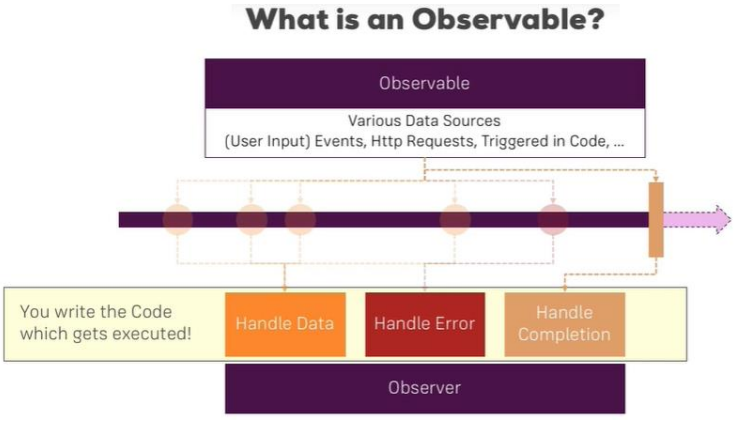
  - Using the resolve interface also ensures that any data is received back from the API call prior to the view being rendered, since the data will be pre-fetched from the server using a resolver so that it's ready the moment the route is activated. This also allows you to handle errors before routing to the component.
  - This data is accessed in a similar way as the static data was accessed above. In the server example above, the server is loaded into the server property of the routes array and can be used for those routes.

Location Strategies

- Routes as we have implemented so far may not always work the same way once hosted on the web.
  - These routes are always handled first by the server that hosts our application.
  - The server hosting the Angular single-page application must be configured so that in the case of a 404 error, it returns the index html file.
- An example problem is if someone wants to go to the /servers route on a hosted Angular application.
  - Since this is a single-page application, there is no /servers route since all that exists is the index.html page containing the Angular app. We need the server in this case to return the index.html file.
  - To get around this problem, we just need to set the useHash configuration in our routing module to true.
  - This will add a # character to the URL, which is called hash routing. This will ensure that the server only parses the URL information before the # tag in the URL, allowing the Angular client to parse the information after the # tag.
- However, it is best to use the HTML History Mode that supports the normal slash routes without the hash tags.

Section 13 – Understanding Observables

Observables Basic Principles



- There could be an entire course on observables, so this is not a deep dive into them.
- An observable can basically be thought of as a data source, and we import this from a 3rd party package called RxJS.
  - The observable is implemented in a way that follows the observer pattern.
  - Observables are generally used for handling events (e.g. button presses), http requests, etc., where the events are asynchronous (i.e. we do not know how long it will take).
  - Promises and call-backs could also be used, but one big advantage of observables is their operators.
- Observables are constructs to what you subscribe to be informed about changes in data.
  - Whenever data is emitted, the subscription will know about it.
  - For example, when we subscribe to params, we are essentially using the params as the observable.
  - It is important to unsubscribe from observables that you are no longer interested in since we can create a memory leak otherwise as the observables will keep emitting.
- While the RxJS package is a 3rd party package, it is automatically added into Angular projects during the dependency install.

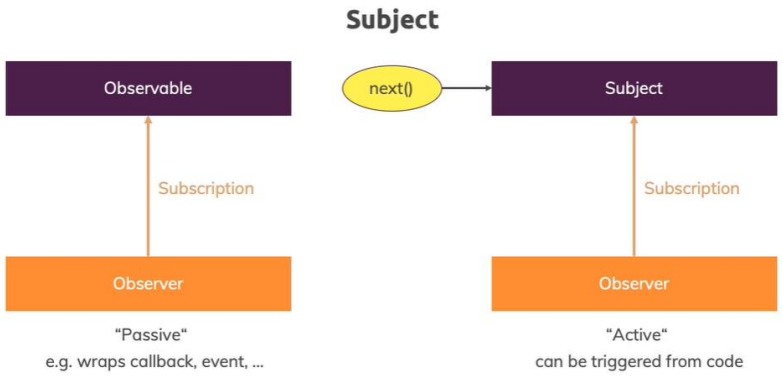
- We should store our subscriptions in a subscriptions variable.
    - It is often a good idea to unsubscribe on the destruction of a component.
    - Keep in mind that any observables handled by Angular (e.g. params) are automatically unsubscribed to by the framework.
- We would instantiate a custom observable by using the Observable.create(...) method call.
    - Once the custom observable is created, it must be subscribed to.
    - We can also throw observer errors on some condition being met and handle them with error handling within the subscription function.
    - The complete method can also be set for an observable, where the observable will stop emitting after a condition is met. This complete condition can be captured upon running and handled as well.

```
const customIntervalObservable = Observable.create(observer => {
  let count = 0;
  setInterval(() => {
    observer.next(count);
    if (count === 5) {
      observer.complete();
    }
    if (count > 3) {
      observer.error(new Error('Count is greater 3!'));
    }
    count++;
  }, 1000);
});


this.firstObsSubscription = customIntervalObservable.pipe(filter(data => {
  return data > 0;
}), map((data: number) => {
  return 'Round: ' + (data + 1);
})).subscribe(data => {
  console.log(data);
}, error => {
  console.log(error);
  alert(error.message);
}, () => {
  console.log('Completed!');
});
```

- It is rare that we build our own observable in Angular, and much more likely that we use and observable that is packaged into the framework.

Operators and Subjects

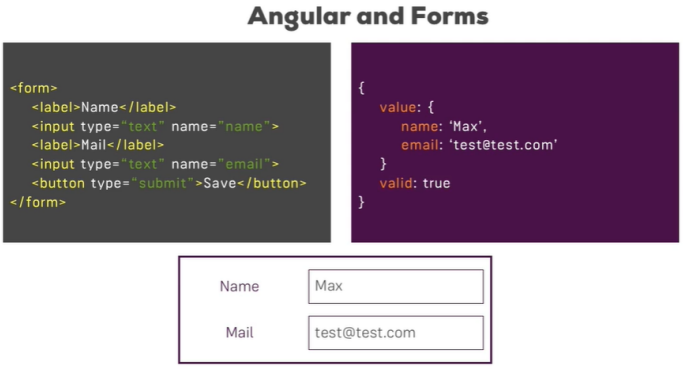
- Operators act between the data and the subscription, performing some operation on the data before it is received by the observer (see screenshot above).
    - We can call a method call pipe() that allows us to apply operators (e.g. map).
- Subjects are similar to observables, but the difference is that we can manipulate it from outside by calling next().
    - Subjects are the recommended way of manipulating elements based on events, rather than using the event emitter that we learned about earlier.
    - Just like observables, we need to make sure that we unsubscribe from subjects when they are not needed.
    - Note that you only use subjects to communicate across components through services. We still use the event emitter when we use @Output.



Section 15 – Handling Forms in Angular Apps

Forms Basic Principles

- Keep in mind that since we have a single page application with Angular, we will not be submitting our forms directly to the server.
    - We instead handle forms with Angular and then reach with Angular's HTTP service to the server.
    - Angular comes packaged with powerful styling and validation functionality.
- Angular offers two approaches when it comes to handling forms.
    - Template-driven approach where Angular infers the form object from the DOM, which makes it easy to get started quickly.



    - Reactive approach where forms are created programmatically and synchronized with the DOM, giving us more control over the form and how it is handled.
- Note that while we could work with html forms as we did earlier in this course, that is not how Angular is meant to work with forms.
    - Using the ngForm directive is a better approach since it adds features such as validation.

Template-Driven Approach

- Ensure first that the forms module has been imported into the app module.
    - ngModel would then have to be added to the input element for the form to be registered with Angular as a form, and a name property would have to be provided to the input element.
    - An onSubmit() method would then need to be added to the component's TypeScript file to handle the submit logic.
    - (ngSubmit)="onSubmit()" event binding would then be added to the form element.
    - A local reference could then be placed on the HTML element to get access to the contents of the form in the onSubmit() method through the use of a @ViewChild. The return data would include the form's entered values.

```
<form (ngSubmit)="onSubmit()" #f="ngForm">
  <div
    id="user-data"
    ngModelGroup="userData"
    #userData="ngModelGroup">
    <div class="form-group">
      <label for="username">Username</label>
      <input
        type="text"
        id="username"
        class="form-control"
        ngModel
        name="username"
        required>
    </div>
```

- While validation should also happen on the server, Angular gives us options to perform validation on the front-end as well.
    - A simple validation feature is to add the required selector to an input. This makes sure that the form will not be empty when submitted.
    - Additionally, an email validator can be added as a selector to a form control, which will set the valid property to false in the NgForm return. Besides the form property, Angular also returns control properties (e.g. ng-invalid) for each control.
    - Refer to the Angular website for a full list of validators, although keep in mind that for the template-driven approach, we need to use directives.

```
@ViewChild('f', { static: false }) signupForm: NgForm;
```

- We can dynamically disable the state of forms (i.e. disable them entirely) if some condition is met.
    - Borders can be added dynamically in CSS to controls based on whether certain form properties are true or false.
    - An additional example would be adding text beneath a form based on some condition (e.g. if a user clicks into a form box but doesn't enter anything).

```
input.ng-invalid.ng-touched {
  border: 1px solid red;
}
```

```
<div class="form-group">
  <label for="email">Mail</label>
  <input
    type="email"
    id="email"
    class="form-control"
    ngModel
    name="email"
    required
    email
    #email="ngModel">
  <span class="help-block" *ngIf="!email.valid && email.touched">Please enter a valid email!</span>
</div>
```

- Default values for forms (i.e. one-way binding) can be added by binding ngModel to a default value provided as an option in the form control. Note that the defaultQuestion is a property defined in the TypeScript file.
    - Keep in mind that two-way binding can also be implemented, where the value added into the form can be outputted in another control.

```
<select
  id="secret"
  class="form-control"
  [ngModel]="defaultQuestion"
  name="secret">
  <option value="pet">Your first Pet?</option>
  <option value="teacher">Your first teacher?</option>
</select>
```

- Groups of controls can be created, which can allow us to validate multiple controls at the same time.
- Radio buttons are another control that is straightforward to implement.
- Note that besides using two-way databinding to set values inside of controls (e.g. add text inside of a textbox), we can also use @ViewChild to pass data into the control through the setValue(...) method.

- Using the form.patchValue(...) method on the @ViewChild variable is a way to set forms on a group without overriding all existing values in the control.
- We can also grab data from forms during a submission process, for instance, and display it onto the page using the *ngIf structural directive.

```
<div class="row" *ngIf="submitted">
  <div class="col-xs-12">
    <h3>Your Data</h3>
    <p>Username: {{ user.username }}</p>
    <p>Mail: {{ user.email }}</p>
    <p>Secret Question: Your first {{ user.secretQuestion }}</p>
    <p>Answer: {{ user.answer }}</p>
    <p>Gender: {{ user.gender }}</p>
  </div>
</div>
```

- If the forms need to be reset in the user interface, we can simply use the reset() method to do so.
- We can check an input against a regular expression by using the [pattern] validator.

Reactive Approach

- For the reactive approach to forms, we need to import the ReactiveFormsModule into the app module for the component that is going to hold the forms. We do not use the ngModel approach that we took in the template-driven approach.

```
<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">
        <div formGroupName="userData">
          <div class="form-group">
            <label for="username">Username</label>
            <input
              type="text"
              id="username"
              formControlName="username"
              class="form-control">
            <span
              *ngIf="!signupForm.get('userData.username').valid && signupForm.get('userData.username').touched"
              class="help-block">
              <span *ngIf="signupForm.get('userData.username').errors['nameIsForbidden']">This name is invalid!</span>
              <span *ngIf="signupForm.get('userData.username').errors['required']">This field is required!</span>
```

  - A property will be created in the TypeScript file to hold the form object.
  - The form needs to be initialized, and therefore it needs to be initialized inside of a lifecycle hook that runs before the template controls are created (so ngOnInit()).
  - A [formGroup] databinding must be created in the template file, where it is assigned to the name of the form group provided in the TypeScript file.

```
ngOnInit() {
  this.signupForm = new FormGroup({
    'userData': new FormGroup({
      'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),
      'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmails)
    }),
    'gender': new FormControl('male'),
    'hobbies': new FormArray([])
  });
```

  - The formControlName property must be then added to each control element in the template file, corresponding to the FormControl object created in the TypeScript file.
- With the reactive approach, it is easy to get access to the form control in the TypeScript file to obtain values, since the form was instantiated in the TypeScript file.

```
onSubmit() {
  console.log(this.signupForm);
  this.signupForm.reset();
}
```

- Since the form is not created in the template file, we won't be assigning the required selector in the template like we did for the template-driven approach.
  - Instead, we set the validation object as required in the FormControl object in the TypeScript file, which Angular checks each time that the input in the form is updated (see screenshot above).
  - We can access the validation for the form in the template file by using *ngIf along with the name of the form object that was instantiated and assigned in the TypeScript file (shown in the template screenshot above).
  - As with the template approach, we can still specify CSS to stylize the controls based on the valid and touched classes.

```
input.ng-invalid.ng-touched {
  border: 1px solid red;
}
```

- Form groups can be nested inside of other form groups, although accessing the form properties will require that a dot-operator be used between the form group and the form property in the template.
- Form arrays can also be used to add form controls dynamically to the user interface.

Validation

- Custom validators can also be created using the reactive workflow.

```
forbiddenNames(control: FormControl): {[s: string]: boolean} {
  if (this.forbiddenUsernames.indexOf(control.value) !== -1) {
    return {'nameIsForbidden': true};
  }
  return null;
}
```

```
'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),
```

- It should receive a control that it should check as an argument.
- If the form control is invalid, null (rather than false) should be returned to the caller. Note that the this keyword in the screenshot will not refer to its containing class in this case since it is Angular that is calling the method, and so the validation method must be bound to this class in the FormControl.
- We would then capture the error code using *ngIf, which would allow us to display a message to the user regarding the error.

```
forbiddenEmails(control: FormControl): Promise<any> | Observable<any> {
  const promise = new Promise<any>((resolve, reject) => {
    setTimeout(() => {
      if (control.value === 'test@test.com') {
        resolve({'emailIsForbidden': true});
      } else {
        resolve(null);
      }
    }, 1500);
  });
  return promise;
}
```

- If we must reach out to a web server for validation, we will have to set up asynchronous validation since the response could take a few seconds to return.
  - An asynchronous validator would wait for the response to come back.
  - While the function would take a control as an input, they will have to return a promise or an observable, both of which handle asynchronous data.

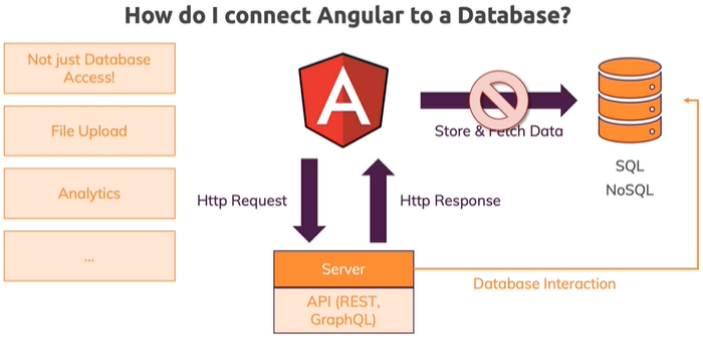
Miscellaneous

```
this.signupForm.setValue({
  'userData': {
    'username': 'Max',
    'email': 'max@test.com'
  },
  'gender': 'male',
  'hobbies': []
});
```

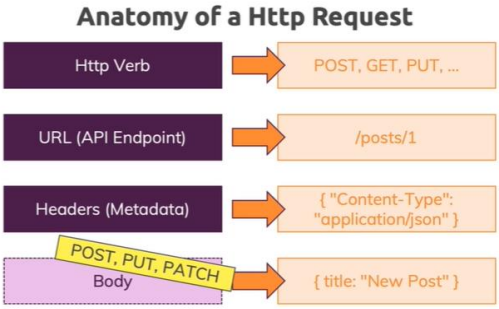
- We can subscribe to status or value changes on a form in the reactive approach by using the statusChanges.subscribe(...) or valueChanges.subscribe(...) methods.
- The values in our forms can be updated in a reactive approach as well, using either the setValue(...) or patchValue(...) methods.

Section 18 – Making Http Requests

HTTP Request Basic Principles



- You would never store credentials inside of the code in your application.
- Http requests and responses and sent to and received from a server that has API endpoints.
  - Typically, data is received back in a JSON format.
- The URL is the endpoint for the API that information is being sent back and forth with.
  - An Http verb consists of POST, GET, PUT, etc., and is how we interact with the API.
  - The verb that can be used with an API endpoint depends on what that endpoint allows.
  - Header information is also sent with an Http request, which is typically optional and is usually done by Angular in this case.
  - The core data attached to the request, which is sometimes provided, is the called the body. For instance, if we send a POST request to a server, the body will contain the information that is to be received by the endpoint.



- For this course, we are using Firebase (made by Google), which will give us both a database and endpoints to work with.
- The Http client module should first be added to the app module file, and then the HttpClient should be added to the component that will handle the Http requests.

- A POST request typically takes two arguments with the first being the URL for the endpoint and the second being the body.
    - The Angular Http client can automatically convert JavaScript objects to JSON data.
    - In order to send the data, the POST method must subscribe to the response which will return whether the POST was successful.
    - Chrome developer tools has the network tab which will show whether the request was sent to the server, and what the response back was.
    - Using observe, as shown in the screenshot, we can get back more information in our response than just the body.

```
createAndStorePost(title: string, content: string) {
  const postData: Post = { title: title, content: content };
  this.http
    .post<{ name: string }>(
      'https://ng-complete-guide-f06a9.firebaseio.com/posts.json',
      postData,
      {
        observe: 'response'
      }
    )
    .subscribe(
      responseData => {
        console.log(responseData);
      },
      error => {
        this.error.next(error.message);
      }
    );
}
```

- GET requests on the other hand, do not have a body since they are not sending any data.
    - To parse the return JSON object, the http.get<> generic method can be used, typically in combination with a pipe to map the data into a property array.
    - It's a good idea to define response data types on the .get<> and .post<> generic methods.
- Once we get a list of items from the endpoint and store them in a post array, they can be displayed in the user interface using the *ngIf structural directive.

```
<p *ngIf="loadedPosts.length < 1 && !isFetching">No posts available!</p>
<ul class="list-group" *ngIf="loadedPosts.length >= 1 && !isFetching">
  <li class="list-group-item" *ngFor="let post of loadedPosts">
    <h3>{{ post.title }}</h3>
    <p>{{ post.content }}</p>
  </li>
</ul>
```

    - It is a good idea to display a loading message if items exist in the database, but the response hasn't been received yet.

Additional Http Notes

- The Http methods should ideally be split out into a separate service, and a model should be created to handle the data.
- As shown in the TypeScript screenshot above, we receive and handle any errors in the second argument.
    - The catchError(...) method can also be used to handle errors in the response.
- Events are rarely used but can provide fine-grained control of a request status.
- While we typically want the responseType to be json, we can also specify type text if we want to parse the data elsewhere.
- It is important when receiving data that all expected items are instantiated, which can be done through a pipe (map) with a ternary operator.

Interceptors

- If, for instance, we need to authenticate our user, we can add a custom header to every outgoing request by using interceptors.
    - We should make an interceptor service if interceptors are determined to be needed, and that service should implement the HttpInterceptor interface.

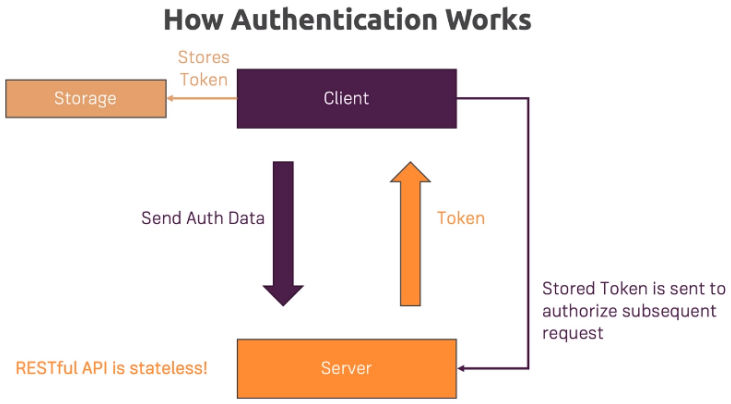
```
export class AuthInterceptorService implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const modifiedRequest = req.clone({
      headers: req.headers.append('Auth', 'xyz')
    });
    return next.handle(modifiedRequest);
  }
}
```

    - The intercept() method will have to be implemented, which will take an HttpRequest and HttpHandler as arguments.
    - Finally, we can append information to a clone of the header, for instance, and append that to the request.
- We also need to make sure that we add provide, useClass, and multi key-value pairs to the providers array in the app module.
- Besides intercepting a POST, we can also intercept a GET request and make changes.
- Multiple interceptors can be used as well, for instance if you need an authentication interceptor as well as a logging interceptor.
    - Be aware of the order that they are added to the app module file, since that order determines the order that they will run.

Section 20 – Authentication & Route Protection in Angular

Authentication Basic Principles

- Any validation of authentication data must be done in the server, since client-side code is exposed publicly on the internet.
    - In a traditional web application where the server generates each HTML page, the authentication would take place with a session.
    - With a single-page application, the client-side is decoupled with the server-side so sessions will not take place.
- Instead, the server will validate a user's credentials and will provide a back a token if the authentication was successful.
    - A token is an JSON encoded string containing meta-data, although it is important to note that the token is not encrypted so that it can be unpacked by the client.
    - The token is generated on the server with a certain algorithm and secret that only the server knows.



- Only the server can validate incoming tokens for their validity.
    - The token is stored in the meantime in local browser storage and is attached to any request that goes to the server for which authentication is required.
    - Token information is typically sent to the server either in the header or as query parameters.
- The server is then able to validate the token because it created the token in the first place with the algorithm and private (secret) key.
    - The token cannot be generated or edited on the client-side because it would not match the server's expected token for a request.

Working with Authentication Files

- When creating new authentication components, remember to add them to the app module file.
    - In the template file for the component, setting the password control to type="password" ensures that the characters are hidden.
    - Ensure that a route is added in the app-routing module to the new authentication page.
- A switch can be added in the AuthComponent to keep track of whether the user is logged in or not, which can then be used to dynamically set the buttons in the user interface.
    - The login button can easily be disabled as well by using the local reference syntax on the form controls.
- We can use Google Firebase if we want for testing token-based authentication.
- An authorization service should be created in the project to be responsible for signing users up, signing users in, and managing the token information.
- The authentication API will require some properties to be provided along as a payload (e.g. email, password, and retrunSecureToken).
    - Keep in mind that the Http client in Angular requires that subscribe be with POST in order to receive the response.
    - We should also define the data the we expect to receive back as an interface, which can be provided to the generic POST method in order to define the expected return data.

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  user = new BehaviorSubject<User>(null);
  private tokenExpirationTimer: any;

  constructor(private http: HttpClient, private router: Router) { }

  signup(email: string, password: string) {
    return this.http
      .post<AuthResponseData>(
        'https://www.googleapis.com/identitytoolkit/v3/relyingparty/
        {
          email: email,
          password: password,
          returnSecureToken: true
        }
      )
      .pipe(
        catchError(this.handleError),
        tap(resData => {
          this.handleAuthentication(
            resData.email,
            resData.localId,
            resData.idToken,
            +resData.expiresIn
          );
        })
      );
  }
```

Loading and Error Handling

- A good resource for loading spinners is the loading.io/css website.
    - We can set an isLoading property in the auth.component.ts file and use *nglf in the template to show the loading icon.
- Errors should be caught if for invalid login or signup attempts.
    - Refer to the sample code for this section to figure out implementation patterns for error handling.

```
private handleError(errorRes: HttpErrorResponse) {
  let errorMessage = 'An unknown error occurred!';
  if (!errorRes.error || !errorRes.error.error) {
    return throwError(errorMessage);
  }
  switch (errorRes.error.error.message) {
    case 'EMAIL_EXISTS':
      errorMessage = 'This email exists already';
      break;
    case 'EMAIL_NOT_FOUND':
      errorMessage = 'This email does not exist.';
      break;
    case 'INVALID_PASSWORD':
      errorMessage = 'This password is not correct.';
      break;
  }
  return throwError(errorMessage);
}
}
```

Handling Tokens

- A user model should be created to store the user and token information that is received from the REST query.
    - The model can also provide some basic token validation. Having the token data sent to private in the model will ensure that any token information is validated before being used elsewhere in the application.
    - Using the tap operator along with the POST method allows us to run some code whenever an event happens, which is this case is a user logging in or signing up. The tap method (shown in the first code screenshot) will allow us to perform some action without changing the response (e.g store the received data into user properties).

```
export class User {
  constructor(
    public email: string,
    public id: string,
    private _token: string,
    private _tokenExpirationDate: Date
  ) {}

  get token() {
    if (!this._tokenExpirationDate || new Date() > this._tokenExpirationDate) {
      return null;
    }
    return this._token;
  }
}
```

- Redirection once logged in can take place in either the authentication component or the service.
    - Upon successful login, the user should be directed.
- Creating the user in the auth.service.ts file as a subject will let us be notified whenever the user changes.
- Always test all routes to ensure that they are not reachable without a user token.
- It is critical that the authorization token be included with any request to the endpoints.
    - A behaviour subject can be used as well to store the user data in a property, which behaves like a subject except that it gives us access to the previous value even if that value was not emitted. This means that that we can get the currently active user even if we subscribe after the user has emitted. So even if the user logged in previously, we can still get access to the user.
    - Then, we can access a single user data using an exhaustMap method inside of the pipe. This will allow us to essentially first confirm that authentication took place, and then get the data for that user from the server.
    - Some databases require that the token be provided in the header, while others require it as a parameter.
- We can also use an interceptor service to add the token to all outgoing requests.
    - The service needs to be added to the app module for the interceptor to take effect.

```
fetchRecipes() {
  return this.http
    .get<Recipe[]>(
      'https://ng-course-recipe-book-1f9c6.firebaseio.com/recipes.json'
    )
    .pipe(
      map(recipes => {
        return recipes.map(recipe => {
          return {
            ...recipe,
            ingredients: recipe.ingredients ? recipe.ingredients : []
          };
        });
      }),
      tap(recipes => {
        this.recipeService.setRecipes(recipes);
      })
    );
}
```

```
@Injectable()
export class AuthInterceptorService implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return this.authService.user.pipe(
      take(1),
      exhaustMap(user => {
        if (!user) {
          return next.handle(req);
        }
        const modifiedReq = req.clone({
          params: new HttpParams().set('auth', user.token)
        });
        return next.handle(modifiedReq);
      })
    );
  }
}
```

- Note that we also must add a check for a user login request, since there will not be a token associated with that request so it will need to be handled differently within the interceptor.
- The nice thing about using an interceptor is that it will work for all requests, and therefore will allow for code reuse.

```
logout() {
  this.user.next(null);
  this.router.navigate(['/auth']);
  localStorage.removeItem('userData');
  if (this.tokenExpirationTimer) {
    clearTimeout(this.tokenExpirationTimer);
  }
  this.tokenExpirationTimer = null;
}
```

- To logout, we simply add a logout() method to the auth.service.ts file and set the user to null.
  - Additionally, we need to clear the user information from local storage.
- To persist login tokens across multiple sessions, either local storage or cookies in the browser can be used.
  - The user object will simply have to be converted to a JSON string and stored as the value in a key-value pair.
  - To view the item in local storage, we can browse to the application tab in Chrome Developer Tools and view the local storage data.

```
localStorage.setItem('userData', JSON.stringify(user));
```

- With local storage being used for authentication tokens, we can create an autologin() method within auth.service.ts to automatically check for a token when the user browses to the site.
  - This method will check whether a user token exists in storage, whether it is expired if so, and will log the user in if the checks have passed successfully.
  - The auto-login method should then be called in the app.component.ts file on ngOnInit() in order to check upon initial load whether a valid user token exists.

```
autoLogin() {
  const userData: {
    email: string;
    id: string;
    _token: string;
    _tokenExpirationDate: string;
  } = JSON.parse(localStorage.getItem('userData'));
  if (!userData) {
    return;
  }

  const loadedUser = new User(
    userData.email,
    userData.id,
    userData._token,
    new Date(userData._tokenExpirationDate)
  );

  if (loadedUser.token) {
    this.user.next(loadedUser);
    const expirationDuration =
      new Date(userData._tokenExpirationDate).getTime() -
      new Date().getTime();
    this.autoLogout(expirationDuration);
  }
}
```

- An auto-logout will be needed to ensure that the application reflects an expired token.
  - The auto-logout method should take the token's expiration duration as an argument, and will then set a timeout for the application (which is a built-in function) based on the duration, automatically calling the logout() method after the expiration time has passed.

```
autoLogout(expirationDuration: number) {
  this.tokenExpirationTimer = setTimeout(() => {
    this.logout();
  }, expirationDuration);
}
```

  - The token expiration timer must be also cleared on manual logout by the user so that it does not timeout after the user has manually logged out.

Auth Guard

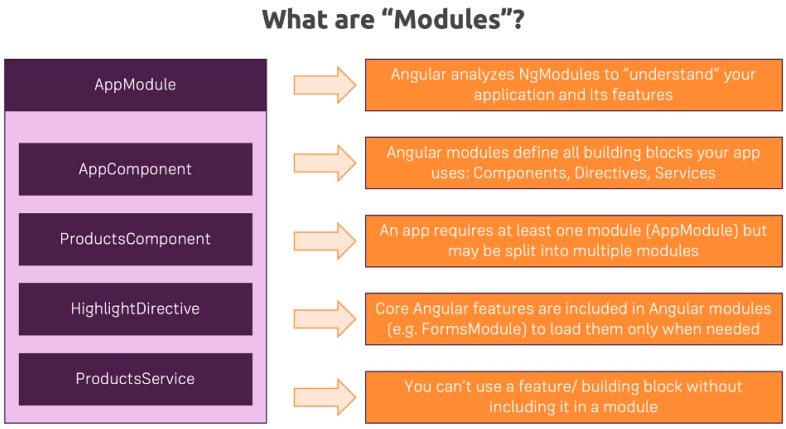
- A route guard allows us to run logic right before a route is loaded, and it can deny access if a certain condition is not met.
  - An auth.guard.ts file should be created, with an AuthGuard class implementation.
- The auth guard should implement the CanActivate interface by creating the canActivate(...) method.
  - This method can return a URLTree, which would provide the path to the redirect page if the user in not logged in. This is a relatively new feature that has been added to Angular but solves some edge cases that were previously occurring.
  - We need to get only the latest user value and then unsubscribe to the observable, so take(1) should be used in this method so that we don't have an ongoing user subscription.

Section 22 – Angular Modules & Optimizing Angular Apps

Modules Basic Principles

- Modules are a way of bundling Angular building blocks – components, directives, services, and pipes – together.

- These blocks of the program must be bundled together into modules so that Angular is aware of these features, since Angular does not automatically scan all files in your project.
- Every Angular app must have at least one module, which is the AppModule, but can also include many modules.



- Angular comes bundled with a bunch of modules, such as the FormsModule, which provides a bunch of forms directives.
- Declarations inside a module is a list of all components, pipes, and directives, which must be declared so that they can be used in templates and routes.
- The imports array allows for other modules to be imported into that module, allowing your app to become modular.
  - By importing a module into another module, we avoid having to declare all the components, pipes, and directives into the parent module directly.
- The providers array allows us to define all the services that we want to provide.
  - Any service needs to be provided either in the app module or by including the providedIn: 'root' key in the @Injectable decorator inside the service.
- The bootstrap array is important for starting your app and defines what component is available right in the index.html file.
  - There is typically only one component – AppComponent – included in this array.
- The entryComponents array defines components that are created programmatically.
- Every module works on its own in Angular and does not know about other modules.
  - We must export the module inside of itself so that we can import it into another module and use everything the module exports.
- As applications grow, it is a good idea to split the app into multiple modules.

```
module.ts   app-routing.module.ts ×   auth.component.html
      resolve: [RecipesResolverService]
    }
  ]
},
{ path: 'shopping-list', component: ShoppingListComponent },
{ path: 'auth', component: AuthComponent }
];

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

```
[@divJ="state"
(@divState.start)="animationStarted($event)"
(@divState.done)="animationEnded($event)">
</div>
```

Section 26 – Angular Animations

Animations Basic Principles

- It is hard to handle transitions and animations when components are being added dynamically to the DOM, which is why Angular includes animations.
- Animations are added into the component TypeScript file, to be used by the template elements.
  - Specifically, an animations array is setup in the @Component decorator.
- Each animation has a trigger function, which is provided name and animation (array) arguments.
  - Animations work between states, where we define two states and then set the animation that will transition between the two states. One state will be the normal state, while the other will be the transformed state.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  animations: [
    trigger('divState', [
      state('normal', style({
        'background-color': 'red',
        transform: 'translateX(0)'
      })),
      state('highlighted', style({
        'background-color': 'blue',
        transform: 'translateX(100px)'
      })),
      transition('normal <=> highlighted', animate(300)),
      // transition('highlighted => normal', animate(800))
    ]),
```

- The div in the template is bound the state property in the component, which can then be changed by event (e.g. a button press).

```
trigger('list2', [
      state('in', style({
        opacity: 1,
        transform: 'translateX(0)'
      })),
      transition('void => *', [
        animate(1000, keyframes([
          style({
            transform: 'translateX(-100px)',
            opacity: 0,
            offset: 0
          }),
          style({
            transform: 'translateX(-50px)',
            opacity: 0.5,
            offset: 0.3
          }),
          style({
            transform: 'translateX(-20px)',
            opacity: 1,
            offset: 0.8
          }),
          style({
            transform: 'translateX(0px)',
            opacity: 1,
            offset: 1
          })
        ]))
```

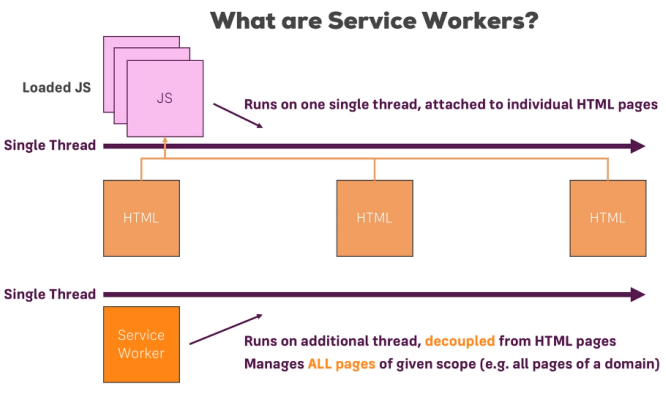
- Inside of each trigger element in the animations array, a transition method should be defined to determine how the two states will transition between one another, along with how much time it will take for the transition.
  - In the transition method, having something like 'shruken ⇔ *' means that the transition will take place between any state.
  - Same as styles can be passed for state, they can also be applied during animation and will allow for multiple phases of transition to take place between the start and end states.
- When adding lists to the DOM, we would start with the void state in the transition method since void is a reserved state name for elements that do not start in the DOM.
  - When animating items being added into the DOM (e.g. new list items), we don't really care about defining a property for the state, since the starting state is generally void and the end state would be any (*).
- Keyframes can also be added to the animations method and will allow us to finely tune which part of multiple styles should occur at which time.
  - The offset property can be set which will precisely define what portion of the total animation time each style will take to animate.
- The group method will allow us to group animations togetheter to be performed at the same time.
- Animation callbacks can be used to run some logic in the TypeScript file either before the animation starts or after it ends.

```
trigger('wildState', [
  state('normal', style({
    'background-color': 'red',
    transform: 'translateX(0) scale(1)'
  })),
  state('highlighted', style({
    'background-color': 'blue',
    transform: 'translateX(100px) scale(1)'
  })),
  state('shrunken', style({
    'background-color': 'green',
    transform: 'translateX(0) scale(0.5)'
  })),
  transition('normal => highlighted', animate(300)),
  transition('highlighted => normal', animate(800)),
  transition('shrunken <=> *', [
    style({
      'background-color': 'orange'
    }),
    animate(1000, style({
      borderRadius: '50px'
    })),
    animate(500)
  ])
])
```

Section 27 – Adding Offline Capabilities with Service Workers

Service Workers Basic Principles

- We can turn an online-only Angular application into one that will work offline by adding service workers.
  - Offline mode can be simulated by click on the Application tab in Chrome Developer Tools and choose Offline in the Service Workers section.
- JavaScript typically runs on a single thread when loaded in an application.
  - However, we can run a service worker on an additional thread which can do things like handle notifications and rest calls.
- Adding the @angular/pwa service worker package will allow us to take advantage of service workers.
  - A ServiceWorkerModule must be added to the app module as an import.
- When we load an Angular app that uses service workers, all the Angular implementation minus the dynamic content will be displayed.
- The service worker config file (ngsw-config.json) can be changed to define how the Angular service worker behaves.
  - It defines what the root page of the app that is to be cached and loaded is.
  - An asset group array is included in this file lists which static assets are to be cached and how they should be cached.
- Setting the installMode to prefetch means that the service worker will immediately cache the specified assets.



- An installMode of lazy will cache assets as they are used for the first time, but this means that the asset may not be available when the user needs it.

- The updateMode property can be set to prefetch or lazy and determines how any updates to the page will be cached.
- To cache data from an API, we can include dataGroups in the service worker config file.
  - We can set properties such as maxSize, which is how many responses we want to cache.
  - maxAge defines how long we want to keep the data in the cache for before refreshing it.
  - The strategy property can be set to freshness, which means that the application always first tries to reach out to the API first, and cache only if the API is not available, or performance, which will load whatever data is fastest to load.

Section 28 – A Basic Introduction to Unit Testing in Angular Apps

Unit Testing Basic Principles

- Unit testing allows us to test whether parts of the application are working as intended.
- An app.component.spec.ts file would typically contain the tests for the application.
  - The beforeEach(...) method will run code before running each test, which are the it(...) methods defined in the file.
- The describe(...) method will be executed by the test runner (CLI in this case).
  - Each it(...) method is executed totally independent of the it(...) methods around it.
- TestBed is the main Angular testing utility object, and we declare within the configureTestingModule(...) method which components we want to include in the testing environment.
  - The expect(...) method is what condition we are testing against.
- Running ng test in the terminal will allow us to run our tests and will display a Karma results page.
- Each component has a test file included when the component is automatically created by the CLI.
  - We can inject services into our testing script to test dynamic data.
  - Keep in mind that we may need to use fixture.detectChanges() in order to update our properties after the data was injected into the component from the service.

```
describe('Component: User', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [UserComponent]
    });
  });

  it('should create the app', () => {
    let fixture = TestBed.createComponent(UserComponent);
    let app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  });
```

- We can test asynchronous data from a server as well, but we won't be reaching out to the endpoint every time we run the test.
  - Instead, we will essentially fake our data inside of the test and use the spyOn(...) method provided with testing framework to get informed whenever the target method gets executed.

```
it('should fetch data successfully if called asynchronously', async(() => {
  let fixture = TestBed.createComponent(UserComponent);
  let app = fixture.debugElement.componentInstance;
  let dataService = fixture.debugElement.injector.get(DataService);
  let spy = spyOn(dataService, 'getDetails')
    .and.returnValue(Promise.resolve('Data'));
  fixture.detectChanges();
  fixture.whenStable().then(() => {
    expect(app.data).toBe('Data');
  });
}));
```

  - We aren't running that asynchronous method but will instead return data that would simulate a return from the method that we are testing, which would be provided back in an asynchronous fashion.
  - The tick() method can also be run to simulate the asynchrounous tasks finishing up when called.