**Angular 2+ Notes  
Feb. 2019**

These notes are from the Pluralsight course titled Angular: Getting Started by Deborah Kurata. The project built while going through this course can be found on GitHub at:  
<https://github.com/DeborahK/Angular-GettingStarted>

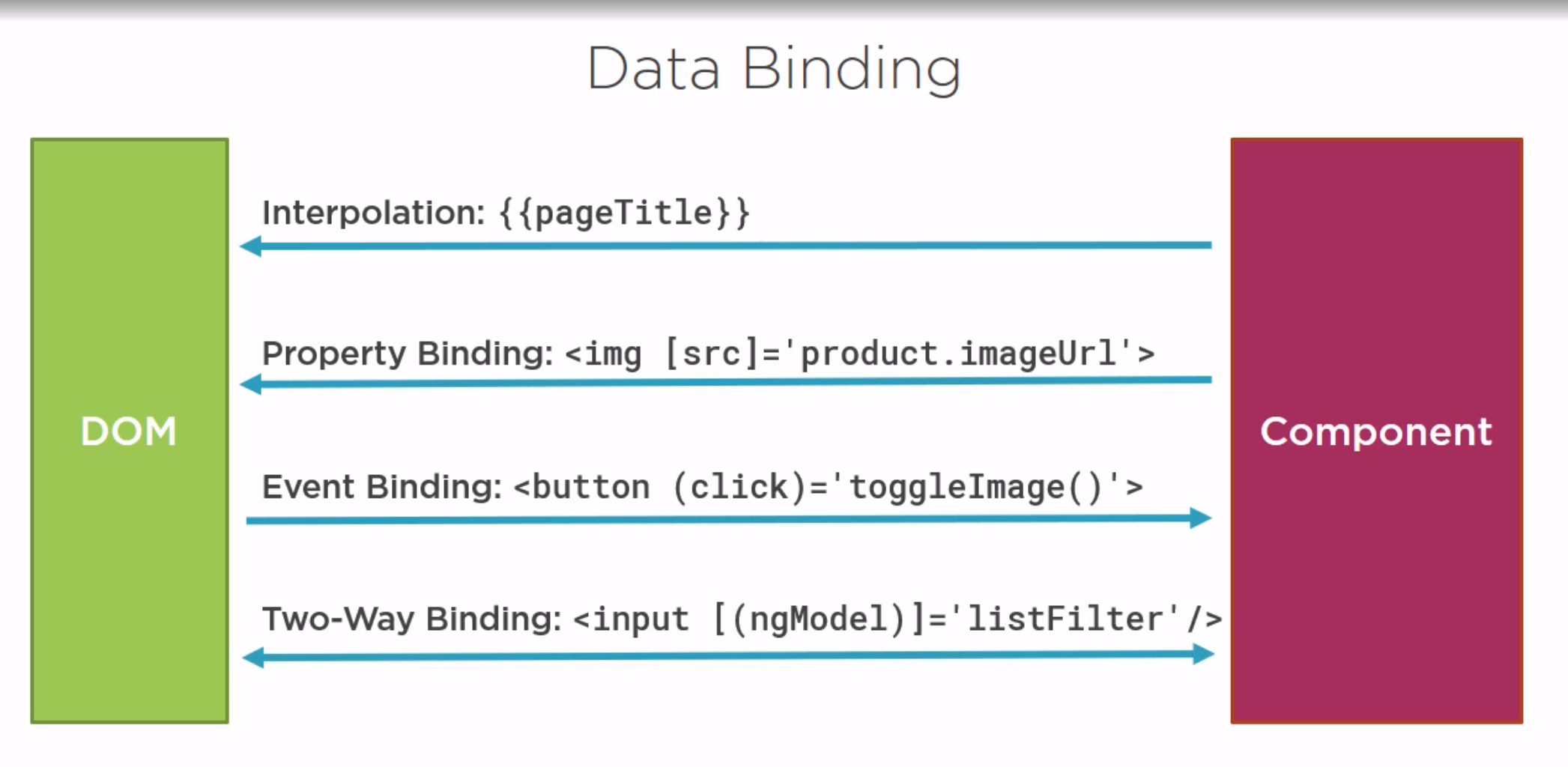
**Components**

* Angular is made up of components.
* Must have at least Angular Core import(s)
* @Component Decorator tells Angular that this file contains an Angular Component
  + The Decorator contains metadata
  + selector: defines the Component name for use in HTML
  + template: inline HTML syntax describing what this Component looks like in HTML
    - templateUrl: HTML syntax is in own file
  + styles: inline CSS syntax to style the HTML associated with this Component
    - stylesUrl: CSS syntax is in own file(s)
* export class required so that other Components can import and use this Component
  + Contains the Typescript code for this Component
* Append “**Component**” to the name of the Component file
* The **index.html** can reference a single Component by using its’ selector (if not using Routing). By default this is set to <pm-root> which is the selector for the **app.componet.ts**.

**Templates**

* The back-ticks (`multi-line text`) are used by ES 2015 (Typescript) to define a multi-line string. If you have a short single line string then simple double quotes (“”) will work.
  + Back-ticks are also used to define a template string that lets you embed properties directly into a string like this:  
    *console.log(`The rating ${this.rating} was clicked!`);*
* NOTE: **npm install bootstrap font-awesome**  
  This CLI command line action installs both Bootstrap and Font-Awesome packages for use in our Angular program.

**Data Binding**

* 
* The ? safe navigation operator after the end of a property binding name will insure that your page does NOT error out in the event of a NULL or Undefined object or property. This can also pertain to and object within an object leading to syntax like this:  
  *{{event?.location?.address}}*This will simply leave the code block blank and continue to render all other elements on the page without breaking.
* ngModel requires that the FormsModule be imported in the Angular app.module.ts
* Pipes can be used to change the display of data bound items.

**Interfaces**

* Interfaces can define custom types
  + i.e. ***export interface IProduct {…}***
* **implements** along with the interface name is used to implement interfaces with methods that must be defined

**Pipes**

* Import **Pipe** and **PipeTransform**
* Use **@Pipe** decorator with name: attribute
* ***export class [PipeName] implements PipeTransform {…}***
* Implement PipeTransform by writing code in the **transform()** method
* To use custom Pipe:
  + Declare and import Pipe class in Angular module (app.module.ts)
  + Pipe then becomes available to all Templates associated to Components that are declared in the same Angular module
  + Use Pipe in templates like standard Pipes using:  
    **| PipeName: ‘optional arguments separated by colons’**
* Repeat data with \*ngFor
* Remove elements with \*ngIf and \*ngSwitch
* Add classes with ngClass
* Add styles with ngStyle

**Lifecycle Hooks**

* Import and implement any Component lifecycle hooks that you need to add code to like:
  + OnInit
  + OnChanges
  + Etc…

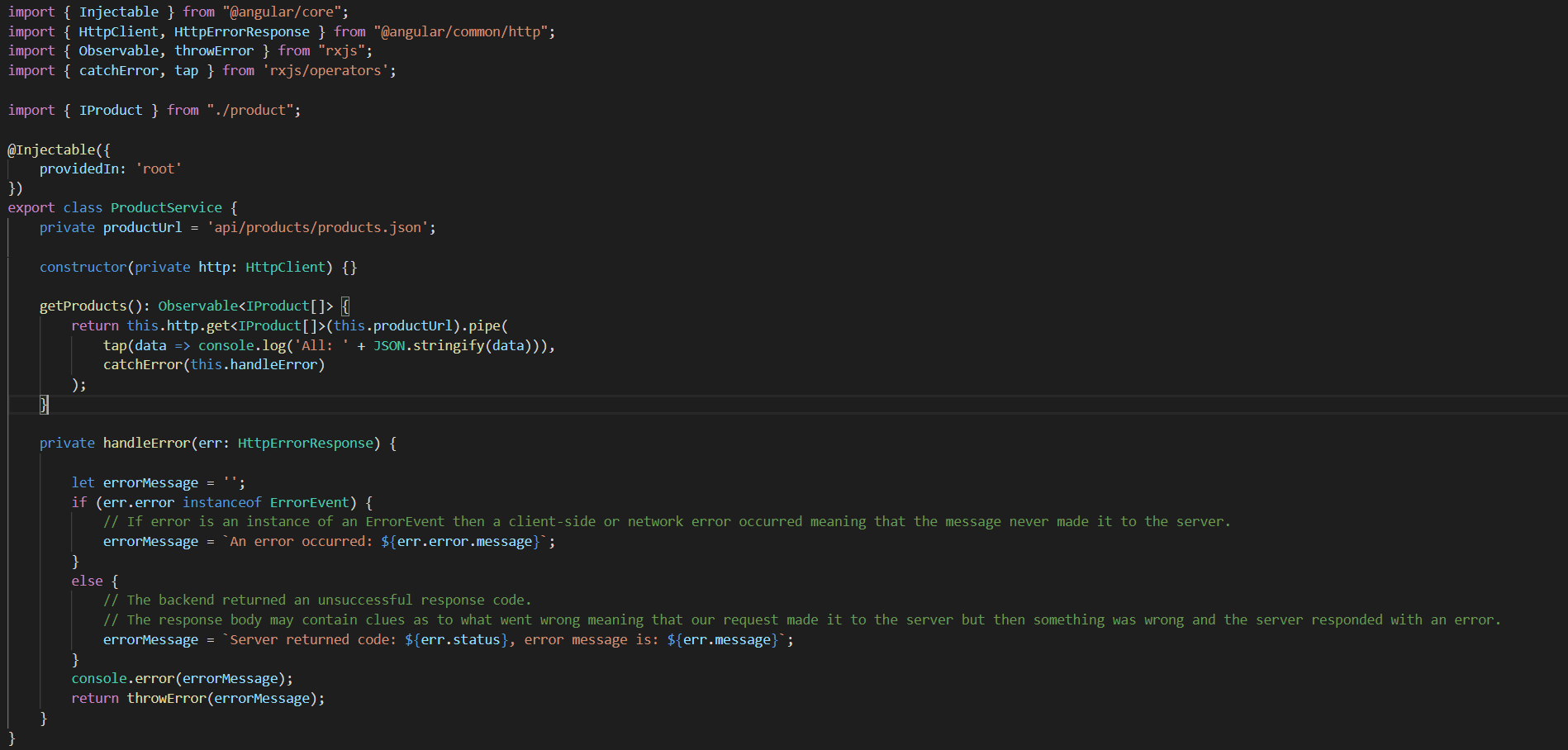
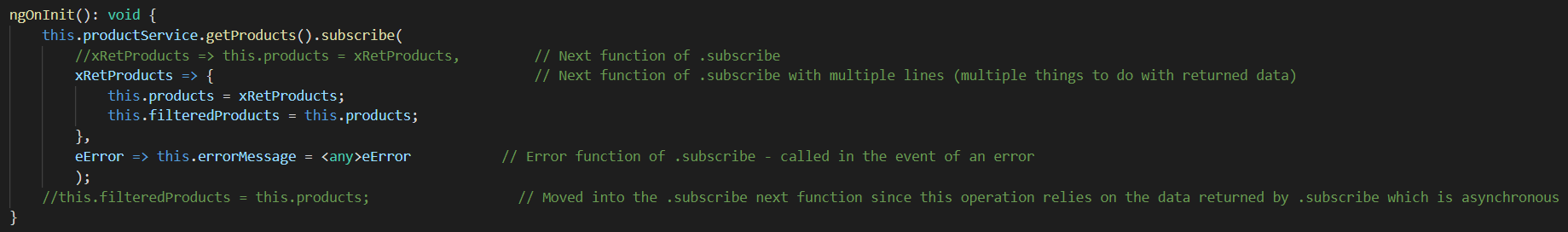
**Nested Components**

* Outer Component is called the container or parent Component
* Inner Component is called the nested or child Component
* The container Component passes data **TO** the nested Component by adding the **@Input()** decorator to a property/variable in the nested Component
* The container Component receives data **FROM** the nested Component via raised Events passed up to it from the child Component using event properties (of type: EventEmitter) decorated with the **@Output()** decorator
  + The EventEmitter takes a generic type to define that type of payload that is passed with the Event
  + Use the new keyword to create an instance of the EventEmitter
* The @Input() and @Output() properties of a Component can be thought of as the public API for that Component
* Use the directive in the container Component’s template by using the nested Component’s selector
  + i.e.: *<pm-star></pm-star>*
* Use property binding to pass data **to** the nested Component. Any property in the nested Component decorated with the @Input() decorator can have data passed to it in this way
  + i.e.: *<pm-star [rating]=’product.starRating’></pm-star>*
* Use event binding to respond to events **from** the nested Component. Any event property in the nested Component decorated with the @Output() decorator raises events that a container Component can respond to
  + i.e.: *<pm-star (ratingClicked)=’onRatingClicked($event)’>*
  + $event contains the Event payload which is passed by the EventEmitter
* An alternative to communicating with a child component is to assign a **template reference variable** (#variableName) which is a variable that points to a child component. This allows the parent component to easily bind to any public properties or call any public methods in the child component without @Input and @Output.  
  Simply add a #variableName to the child component selector tag in the parent component and then reference variableName. in the parent component to have direct access to all public properties and methods in the child component:  
  *<event-thumbnail #thumbnail></event-thumbnail>  
  <h3>{{thumbnail.someProperty}}</h3>  
  <button class="btn btn-primary" (click)="thumbnail.logFoo()">Log foo</button>*

**Services**

* A Service is a class that is decorated with the @Injectable() decorator and is registered within an Angular Injector
* Services that are registered with the Root Injector are available to all Components in the application (recommended for most scenarios)
  + If a Service is registered with a particular Component then only that Component and its’ children/nested Components can use the Service
    - This is when you would use the providers: metadata in the @Component decorator. providers: is NOT used when the service is registered in the root injector
  + To register a Service with the Root Injector simply add *{ providedIn: ‘root’ }* as metadata within the @Injectable decorator in the Service class
    - Do NOT use the providers: array in the app.module.ts since Angular 6+ to register Services. Use the providedIn: attribute described above.
  + To use a Service within a Component:
    - Use the Typescript shorthand to declare and instantiate the service in as an argument in the Component constructor  
      i.e.: *constructor(private productService: ProductService)*
    - And add the Service to the imports at the top of your Component class  
      i.e.: *import { ProductService } from “./product.service”;*

**HTTP**

* IMPORT
  + To use the **HttpClientModule** you need to import:  
    *import { HttpClientModule } from '@angular/common/http';*  
    the module in our **app.module.ts** and add **HttpClientModule** to the imports array.
* SERVICE
  + HTTP requests should be wrapped in a Service who’s constructor instantiates an instance of **HttpClient**
  + Each HTTP request should have its’ method/function defined in the Service, using the HttpClient Service and calling the desired HTTP method such as **get** with a Url argument to the web service serving the HTTP responses
    - The Url can optionally point directly to a .JSON file that contains your objects since JSON is what is returned by many web services
  + Use generics to specify the returned type which can even be your own type/object
  + Error handling can also be added to the Service to address any errors that may occur when making the HTTP requests  
    
* COMPONENT/CLASS
  + Instantiate an instance of your Service (containing HttpClient) within the constructor of a Component or other class that needs the data from the Service  
    
  + Call the **subscribe** method within the Component to begin the emitting of data from the Observable
  + Provide a function to the **subscribe** method to react to the emitted items.
    - Normally a property within the Component is set to the returned JSON object to be displayed or used by the Component
    - Optionally, to execute multiple lines of code in the **subscribe** method, you can use curly brackets {} like this:  
      *data => { this.products = data; this.filteredProducts = this.products; }*
  + An error function can also be defined as a second argument to the **subscribe** function to react to any error that may occur when making a particular HTTP request from our Service  
    

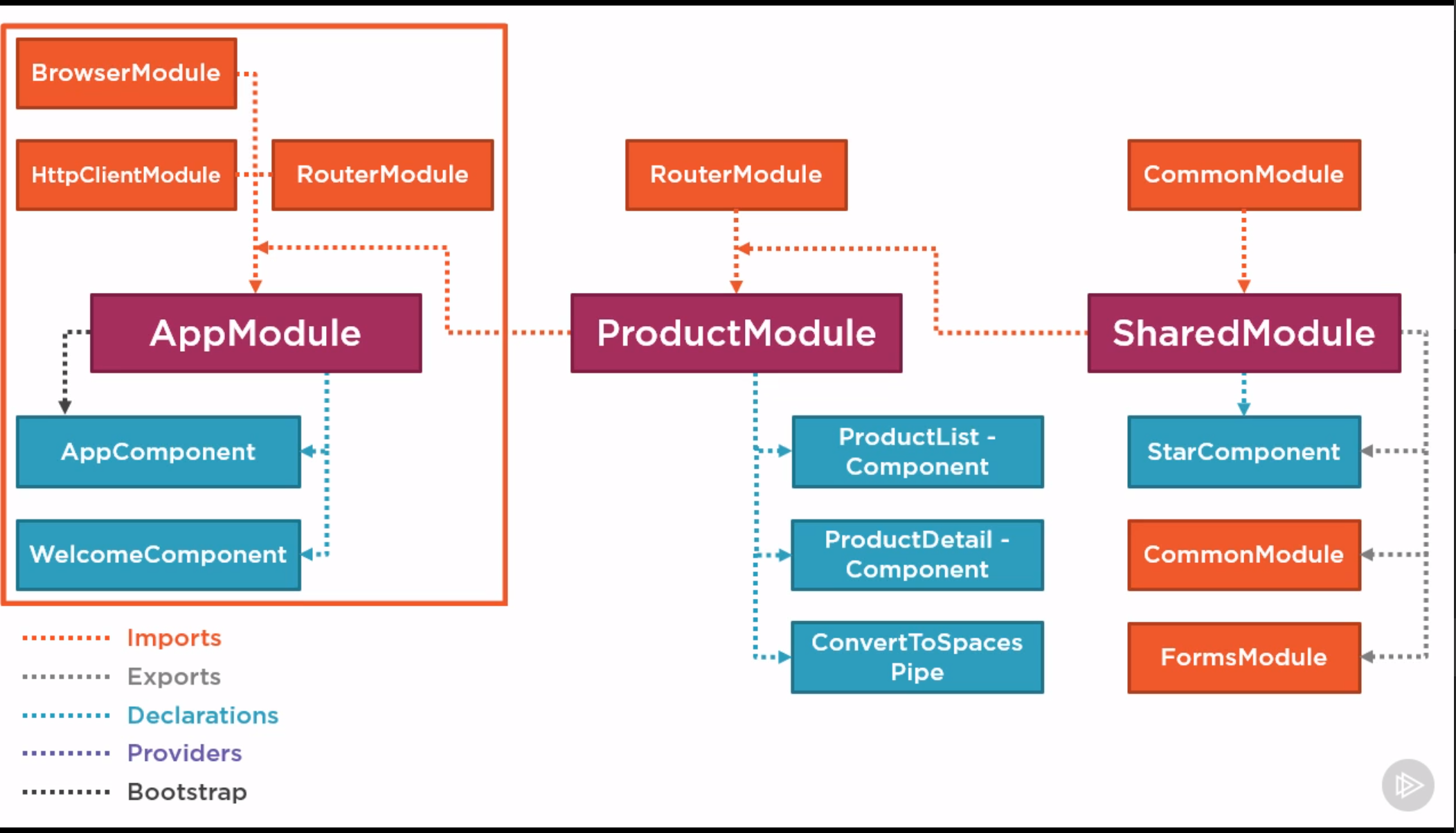
**ROUTING**

* To use Routing, you need to import the **RouterModule**t:  
  *import { RouterModule } from '@angular/router';*  
  in our **app.module.ts** and add **RouterModule** to the imports array.
* Configure a route for each Component that will need to be displayed
* Components that will be set up with a Route, DO NOT NEED the **selector** in the @Component metadata
* Routes can be set up/configured in the **app.module.ts** like this:  
  
* Routes are activated by tied actions
* Place the view for your Router using the *<router-outlet></ router-outlet>* tag
* CONFIGURING ROUTES
  + Define the base element in the index.html with *<base href=”/”>*
  + Import the RouterModule into an Angular Module
  + Add each route to the Router Module by passing an array to the **RouterModule.forRoot** method
    - Router order matters. The router will pick the first match.
  + Routes require a **path:** (the Url segment for the route)
  + Most routes also include a **component:** argument that points to the Component itself
* TYING ROUTES TO ACTIONS
  + Add the **RouterLink** directive as an attribute to any clickable element
    - The RouterLink should be added as an attribute and should be enclosed in square brackets[]
    - Bind (set equal to) to a link parameters array (string arguments in an array separated by commas)
    - The first element in the array is the path and all other elements in the array are route parameters  
      i.e.: *<li><a class='nav-link' [routerLink]="['/products']">Product List</a></li>*
* PLACING THE VIEW
  + Add the **RouterOutlet** directive where you want to display the routed Components view
    - Specified in the host Component’s template  
      i.e.: *<router-outlet></router-outlet>*

**ROUTING – ADDITIONAL**

* Passing parameters to a Route:
  + CONFIGURE ROUTE:  
    *{ path: 'products/:id', component: ProductDetailComponent },  
    // More URL arguments can be used by adding more /:*
  + TIE ROUTE TO AN ACTION:  
    *<a [routerLink]="['/products', product.productId]">{{product.productName}}</a>*
  + EXTRACT PARAMETER
    - Use the **ActivatedRoute** Service in the Component that needs to access the Route parameter  
      *constructor(private route: ActivatedRoute) { }*
    - To get a single initial parameter you can use the single line *snapshot.paramMap.get(‘’)* function like this:  
      *let id = +this.route.snapshot.paramMap.get('id');*
      * The parameter name must match the Route configuration parameter name
    - The **let** (like var but better) defines a block scoped variable, the **+** sign converts a string to a number
    - Getting multiple parameters or variable/changing parameters requires the use of an Observable
* Activating a Route with Code
  + Use the **Router** Service to programmatically activate a route  
    *constructor(private route: ActivatedRoute, private router: Router) { }*
  + Use the *.navigate* function to activate a route by name like this:  
    *this.router.navigate(['/products']);*
  + A function that activates a route can be tied to a user interaction using event binding
* Protecting Routes with Guards
  + A Guard is a Service that can be used to control routing to a Route based on conditions
  + There are different types of Guards with CanActivate being the default
    - CanActivate, CanDeactivate, Resolve, and CanLoad
  + A Guard can be used to check a Route parameter to make sure that it is valid
  + The logic for a Guard is coded in the body of the Guard such as canActivate(next, state){…}
  + The Guard is then added directly to the Route configuration within an array. Multiple Guards may be set up for a single Route if needed:  
    *{ path: 'products/:id', canActivate: [ProductDetailGuard], component: ProductDetailComponent }*

**Additional Modules are used to organize an Angular program into sections that can be imported into other Modules:**



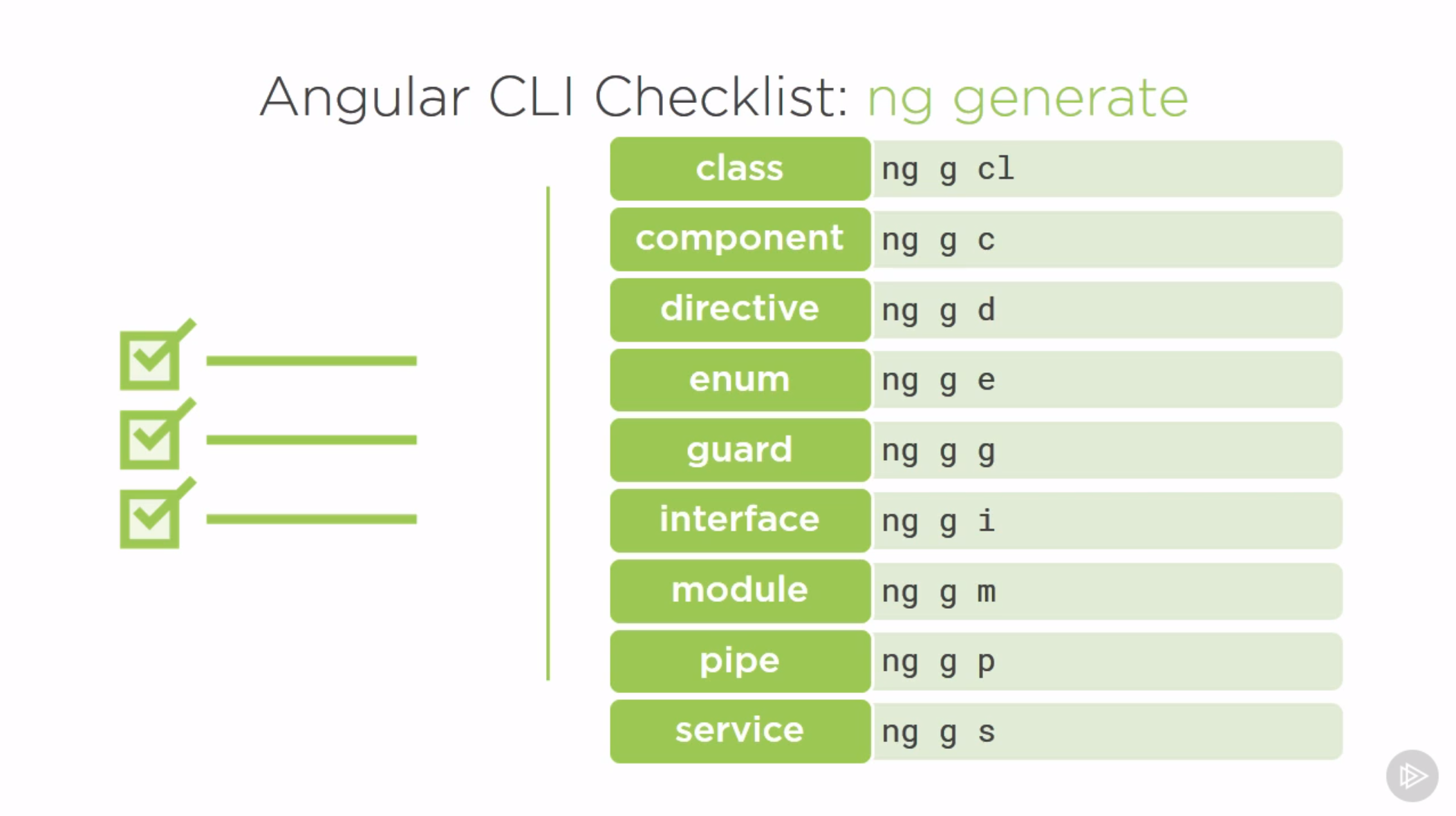
**Angular CLI Commands:**

**ng help**

**ng new (see below for popular options)**

**ng serve –o** Serve a dev build in memory

**ng g**

**ng g --help**

To create a new app with Routing, prefix of fcs (instead of app), and use Sass CSS Style Sheets **ng new my-new-app --routing --prefix fcs --style scss**

**--dry-run** or **-d** Run a dry run to see what will be created

**--skip-install** Create an app without all of the npm install dependency files

**ng new my-app --routing** Create a new app with routing pre-configured

**ng g m admin --routing -m app.module** Create a new admin module with child routing that is imported into the app.module

**ng g c admin/users** Create an users component inside of the admin module

**ng build** Dev build of Angular project

**ng build --prod** Production build of Angular project that includes tree shaking and uglification to reduce size. Also uses AOT(Ahead of Time compiling) by default which also reduces size.

**ng add @angular/material** Add Angular Material to your Angular project

**ng g @angular/material:material-nav --name nav** Generate and add a material-nav component named nav using the Angular Material schematic named material-nav

**ng g @angular/material:material-dashboard --name dashboard** Generate and add a material-dashboard component named dashboard

**ng g @angular/material:material-table --name customer-list** Generate and add a material-table component named customer-list

**angular.json:**

* Custom or 3rd Party Styles and Scripts (like jQuery) can be added to the build section of the angular.json file to be used throughout a program.

**GIT**

* To set your project up with Git you can create a new project in GitLab/GitHub with the same name as your Angular project
  + Setup involved installing Git for Windows
  + Having Daniel setup up my Environment Variable for HOME to Users/bschroeder instead of G:/
  + Creating an empty project within the GitLab interface (https://gitrepo)
* Then, run these Git commands in a terminal window (after installing and setting up Git) to push your project to a remote Git repository:
  + *cd existing\_folder*
  + *git init*
  + *git remote add origin* [*git@gitrepo.court.fresno:bschroeder/test-app.git*](mailto:git@gitrepo.court.fresno:bschroeder/test-app.git)
  + **OR** *git remote add origin* [*https://github.com/BrianSchroederJr/ng-fundamentals.git*](https://github.com/BrianSchroederJr/ng-fundamentals.git)NOTE: Make sure that the repository is empty and brand new before running these commands or else they will fail since the repository has work that you do not have locally!
  + *git add .*
  + *git commit -m "Initial commit"*
  + *git push -u origin master*