**Angular 2**

**Just-in-time (JIT) compilation**

A bootstrapping method of compiling components and modules in the browser and launching the application dynamically. Just-in-time mode is a good choice during development.

**Difference between JIT and AOT:**

JIT - Compile TypeScript just in time for executing it.

* Compiled in the browser.
* Each file compiled separately.
* No need to build after changing your code and before reloading the browser page.
* Suitable for local development.

AOT - Compile TypeScript during build phase.

* Compiled by the machine itself, via the command line (Faster).
* All code compiled together, inlining HTML/CSS in the scripts.
* No need to deploy the compiler (Half of Angular size).
* More secure, original source not disclosed.
* Suitable for production builds.

**app/app.component.ts**

Defines the same AppComponent as the one in the QuickStart playground. It is the root component of what will become a tree of nested components as the application evolves.

**app/app.module.ts**

Defines AppModule, the root module that tells Angular how to assemble the application. Right now it declares only the AppComponent. Soon there will be more components to declare.

**main.ts**

Compiles the application with the JIT compiler and bootstraps the application's main module (AppModule) to run in the browser. The JIT compiler is a reasonable choice during the development of most projects.

**Observables**

Each Http service method returns an Observable of HTTP Response objects.

The HeroService converts that Observable into a Promise and returns the promise to the caller. This section shows you how, when, and why to return the Observable directly.

Background

An Observable is a stream of events that you can process with array-like operators.

Angular core has basic support for observables. Developers augment that support with operators and extensions from the RxJS library. You'll see how shortly.

Recall that the HeroService chained the toPromise operator to the Observable result of http.get(). That operator converted the Observable into a Promise and you passed that promise back to the caller.

Converting to a Promise is often a good choice. You typically ask http.get() to fetch a single chunk of data. When you receive the data, you're done. The calling component can easily consume a single result in the form of a Promise.

But requests aren't always done only once. You may start one request, cancel it, and make a different request before the server has responded to the first request.

A request-cancel-new-request sequence is difficult to implement with Promises, but easy with Observables.

<https://hassantariqblog.wordpress.com/2016/12/03/angular2-http-delete-using-observable-in-angular-2-application/>

When importing from the @angular/http module, SystemJS knows how to load services from the Angular HTTP library because the systemjs.config.js file maps to that module name. The HttpModule is necessary for making HTTP calls.

Observable

Think of an Observable as a stream of events published by some source. To listen for events in this stream, subscribe to the Observable. These subscriptions specify the actions to take when the web request produces a success event or a fail event (with the error in the payload).

The observable’s map callback moves to the success parameter and its catch callback to the fail parameter in this pattern.

The errorHandler forwards an error message as a failed promise instead of a failed observable.

Observable vs Promises

The less obvious but critical difference is that these two methods return very different results.

The promise-based then returns another promise. You can keep chaining more then and catch calls, getting a new promise each time.

The subscribe method returns a Subscription. A Subscription is not another Observable. It’s the end of the line for observables. You can’t call map on it or call subscribe again. The Subscription object has a different purpose, signified by its primary method, unsubscribe.

* Angular Change Detection : “<https://blog.thoughtram.io/angular/2016/02/22/angular-2-change-detection-explained.html>”
* Angular ngIf vs ngShow : ngIf will not create the template or it won't render the template (i.e it will completly remove the template) but ngShow will create the template and hide it.

ngIf is good pratice when we have to hide some template in the beginning base d on some logic but this is happening frequently then ngIf will be costly thats why in that case we will use ngShow.

* Angular css : any css defined in a particular component is limited to that component only,it won't affect the child nor parent component.

and if we want some css to be applied to the child component the we should use "deep".

ngSwitch will also hide the template by not creating them just like ngIf or commenting it.

* constructor(private eventService:EventService){}

here we are creating an instance of EventService. above code is same as

let eventService = new EventService();

* @Injectable : when we have to inject some other service or dependencies inside service then we need @Injectable.
* To use a global variable we can declare like:

ex. declare let toastr: any;

* Angular Routing
* canLoad: it will let us decide whether a not a user should navigate to another page or not.

**Injection Token**

* + Use an InjectionToken whenever the type you are injecting is not reified (does not have a runtime representation) such as when injecting an interface, callable type, array or parametrized type.
  + Generally, use Injection Token for third-party global.

Ex. import { ReflectiveInjector } from '@angular/core';

import { InjectionToken } from '@angular/core';

class MandrillService {};

class SendGridService {};

let EmailService = new InjectionToken<string>("EmailService");

let injector = ReflectiveInjector.resolveAndCreate([

{ provide: EmailService, useClass: SendGridService }

]);

let emailService = injector.get(EmailService);

console.log(emailService);

* **Angular use direct DOM rendering system.**
* **ViewChild: (**<https://codecraft.tv/courses/angular/components/viewchildren-and-contentchildren/>)

It adds a reference of child from view DOM to the component.

A @ViewChild decorator means, search inside this components template to find the exact DOM node, it’s view, for this child component.

The parameter we pass as the first argument to @ViewChild is the type of the component we want to search for, if it finds more than one it will just give us the first one it finds.

* **ViewChildren:**

But when there are multiple child components or DOM nodes (using ngFor), then we use Viewchildren. It returns a querylist.

Ex. @ViewChildren(JokeComponent) jokeViewChildren: QueryList<JokeComponent>;

* **ContentChildren:**

The concept of a content child is similar to that of a view child but the content children of the given component are the child elements that are projected into the component from the host component.

* **Differnce between observable and promise**

**Promis:** The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

* + Restricted to single response
  + It is asynchronous
  + Whereas, Promises cannot be canceled.

**Observable 0**

* + An Observable is like a **Stream** and represents 0 or multiple value now or in the future.
  + It is both asynchronous and synchronous
  + It is able to handle multiple values time to time.
  + We can make changes to the response using “map”.
  + Observables can be canceled.
* An observable is like a stream which allows passing zero or more events where the callback is called for each event. Whereas, A promise eventually calls the success or failed callback.
* Observable works with multiple values for a particular time. Whereas, Promises works with and even returns a single value at a time.
* Observables can be canceled. Whereas, Promises cannot be canceled.
* Observable supports map, filter, reduce and similar operators. Whereas, Promises have more readable codes with try/catch and async/await.
* In observable, one operator ‘retry’ can be used to retry whenever needed. Whereas, Promises cannot be retried. A promise should have access to the original function that returned the promise in order to have a retry capability.

Observable also has the advantage over Promise to be cancelable. If the result of an HTTP request to a server isn't needed anymore, the Subscription of an Observable allows to cancel the subscription, while a Promise will eventually call the success or failed callback even when you don't need the notification or the result it provides anymore.

What is Ecma Script?

ECMAScript (or ES)[1] is a trademarked[2] scripting-language specification standardized by Ecma International,  It was created to standardize [JavaScript](https://en.wikipedia.org/wiki/JavaScript), JavaScript has remained the best-known implementation of ECMAScript since the standard was first published

**Angular Constructor vs ngOnInit:**

Angular has constructor and ngOnInit method which are executed on component load. The main confusion here is when to use what?

The constructor is not actually part of Angular 2. It is feature of Typescript class, which is called when class is instantiated. Constructor is used to create new instance of a class. It can’t be used to decide when Angular has finished the initialization of component. In Angular, constructor can be used to initialize the fields and angular DI [Dependency Injection] looks for constructor parameters to find providers and resolve dependencies and passes them as constructor arguments.

Since constructor is called by JavaScript engine not by Angular, ngOnInit lifecycle hook was created in Angular. ngOnInit signals that Angular has finished initializing and setting up component. The ngOnInit is called after the constructor is executed. In constructor Angular initializes and resolves all class members so in ngOnInit you can initialize work and logic of the component.

**Component:**

A component controls a patch of screen called a view. we define logics and manipulate data inside class in component. The class interacts with the view through an API of properties and method. Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed.

The @Component decorator identifies the class immediately below it as a component class, and specifies its metadata. This decorator actually tells angular that it is a component.

some of the most useful @Component configuration options:

* **selector**: A selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <app-hero-list></app-hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
* **templateUrl**: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.
* **providers**: An array of providers for services that the component requires.

**Pipes:**

Angular pipes let us declare display-value transformations in your template HTML. A class with the @Pipe decorator defines a function that transforms input values to output values for display in a view.

**Directives**

When Angular template renders, it transforms the DOM according to the instructions given by directives. A directive is a class with a @Directive() decorator.

A component is technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the @Component() decorator, which extends the @Directive() decorator with template-oriented features.

In addition to components, there are two other kinds of directives: structural and attribute. Angular defines a number of directives of both kinds, and you can define your own using the @Directive() decorator. In templates, directives typically appear within an element tag as attributes, either by name or as the target of an assignment or a binding.

**What are structural directives?**

Structural directives alter layout by adding, removing, and replacing elements in the DOM. As with other directives, you apply a structural directive to a host element. Structural directives are easy to recognize. An asterisk (\*) precedes the directive attribute name.

Three of the common, built-in structural directives—NgIf, NgFor, and NgSwitch

**Attribute directives:**

Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

The ngModel directive, which implements two-way data binding, is an example of an attribute directive. ngModel modifies the behavior of an existing element (typically <input>) by setting its display value property and responding to change events.

**Modules:**

Angular has its own modularity system called NgModules. NgModules are containers for a cohesive block of code dedicated to an application domain.

The Angular module helps you to organize an application into associative blocks of functionality. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. The NgModule is used to simplify the ways you define and manage the dependencies in your applications and also you can consolidate different components and services into associative blocks of functionality.

Every Angular app has at least one NgModule class, the root module, which is conventionally named AppModule

An NgModule is defined by a class decorated with @NgModule(). The most important properties are as follows:

* declarations: The components, directives, and pipes that belong to this NgModule.
* exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.
* imports: Other modules whose exported classes are needed by component templates declared in this NgModule.
* providers: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
* bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

**Services:**

A service is typically a class with a narrow, well-defined purpose. A component can delegate certain tasks to services, such as fetching data from the server, validating user input.

To define a class as a service in Angular, use the @Injectable() decorator to provide the metadata that allows Angular to inject it into a component as a dependency.

services can also be provided in specific components. Services provided in component-level is only available within that component injector or in any of its child components.

**Dependency injection (DI):**

Dependency Injection (DI) is a way to create objects that depend upon other objects. A Dependency Injection system supplies the dependent objects (called the dependencies) when it creates an instance of an object.

**Injectors:**

An Angular injector is responsible for creating service instances and injecting them into classes.

**Lifecycle Hooks:**

Angular creates components or directives, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. In angular we can tap into particular key moments with the help of certain lifecycle sequence:

|  |  |
| --- | --- |
| ngOnChanges() | Respond when Angular (re)sets data-bound input properties. The method receives a [SimpleChanges](https://angular.io/api/core/SimpleChanges) object of current and previous property values.  Called before ngOnInit() and whenever one or more data-bound input properties change. |
| ngOnInit() | Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.  Called once, after the first ngOnChanges(). |
| ngDoCheck() | Detect and act upon changes that Angular can't or won't detect on its own.  Called during every change detection run, immediately after ngOnChanges() and ngOnInit(). |
| ngAfterContentInit() | Respond after Angular projects external content into the component's view / the view that a directive is in.  Called once after the first ngDoCheck(). |
| [ngAfterContentChecked()](https://angular.io/api/core/AfterContentChecked#ngAfterContentChecked) | Respond after Angular checks the content projected into the directive/component.  Called after the ngAfterContentInit() and every subsequent ngDoCheck(). |
| ngAfterViewInit() | Respond after Angular initializes the component's views and child views / the view that a directive is in.  Called once after the first [ngAfterContentChecked()](https://angular.io/api/core/AfterContentChecked#ngAfterContentChecked). |
| [ngAfterViewChecked()](https://angular.io/api/core/AfterViewChecked#ngAfterViewChecked) | Respond after Angular checks the component's views and child views / the view that a directive is in.  Called after the ngAfterViewInit and every subsequent [ngAfterContentChecked()](https://angular.io/api/core/AfterContentChecked#ngAfterContentChecked). |
| ngOnDestroy() | Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.  Called just before Angular destroys the directive/component. |

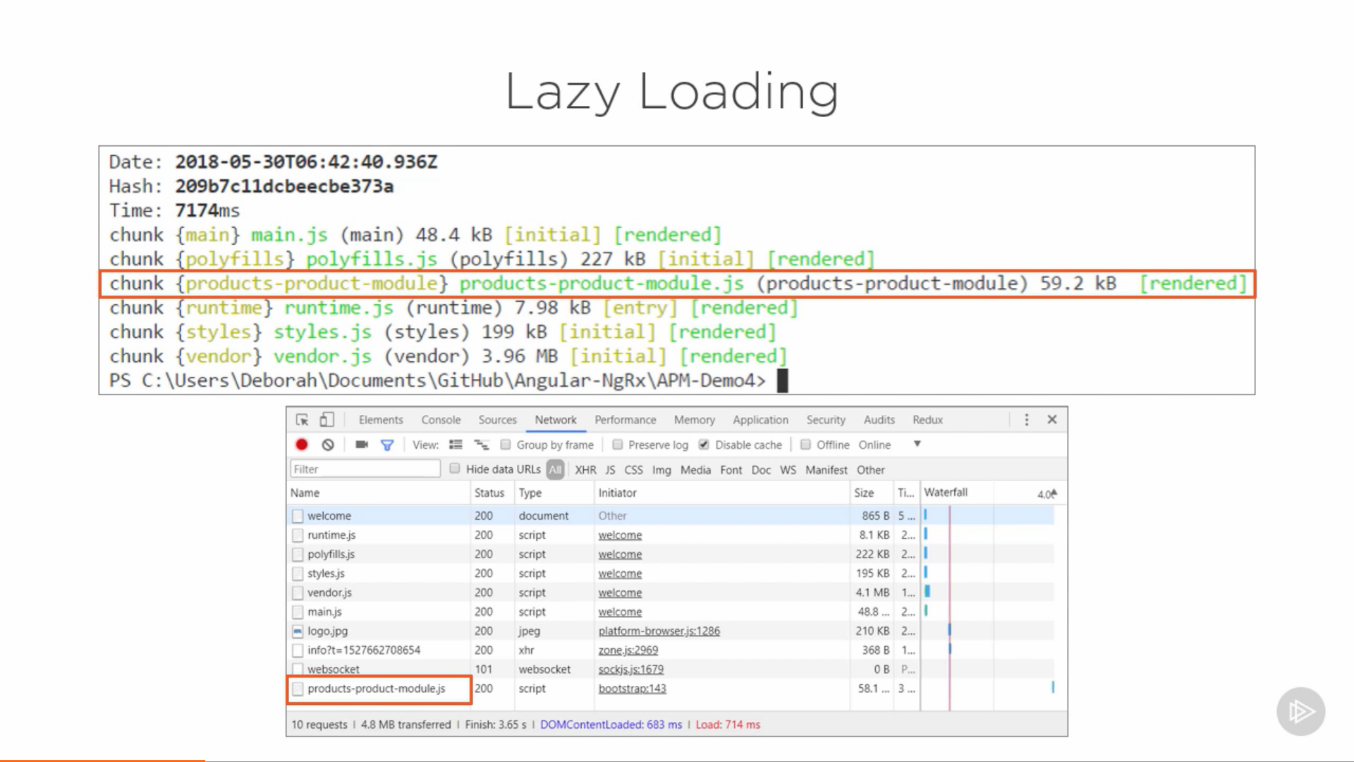
Template Driven Forms Features

* Easy to use
* Suitable for simple scenarios and fails for complex scenarios
* Similar to AngularJS
* Two way data binding(using [(NgModel)] syntax)
* Minimal component code
* Automatic track of the form and its data(handled by Angular)
* Unit testing is another challenge
* Reactive Forms Features

More flexible, but needs a lot of practice

* Handles any complex scenarios
* No data binding is done (immutable data model preferred by most developers)
* More component code and less HTML markup
* Reactive transformations can be made possible such as
  + Handling a event based on a debounce time
  + Handling events when the components are distinct until changed
  + Adding elements dynamically
* Easier unit testing

**Lazy Loading: (**[**https://angularfirebase.com/lessons/how-to-lazy-load-components-in-angular-4-in-three-steps/**](https://angularfirebase.com/lessons/how-to-lazy-load-components-in-angular-4-in-three-steps/)**)**

Lazy loading is a technique in Angular that allows you to load angular module asynchronously when a specific route is activated.  
  
when we build our app it builds the lazy loaded module as a separate js file and it will be downloaded only when specific url is hit otherwise it won’t.  


The benefits are:

* Keep the initial payload small (which improves app startup performance)
* Smaller payloads lead to lesser download time
* Lower resource costs especially on mobile networks
* If a user doesn't visit a section of our app, they won't ever download those resources.

**Angular Compilation:**

Before the browser can render the application, the components and templates must be converted to executable JavaScript by an Angular compiler.

Angular offers two ways to compile your application:

1. Just-in-Time (JIT), which compiles your app in the browser at runtime. It is default.

ng build

ng serve

1. Ahead-of-Time (AOT), which compiles your app at build time.

ng build --aot

ng serve –aot

**Why compile with AOT?**

* Faster rendering: With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.
* Fewer asynchronous requests: The compiler inlines external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.
* Smaller Angular framework download size: There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.
* Detect template errors earlier: The AOT compiler detects and reports template binding errors during the build step before users can see them.
* Better security: AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

**What are Event emitters?**

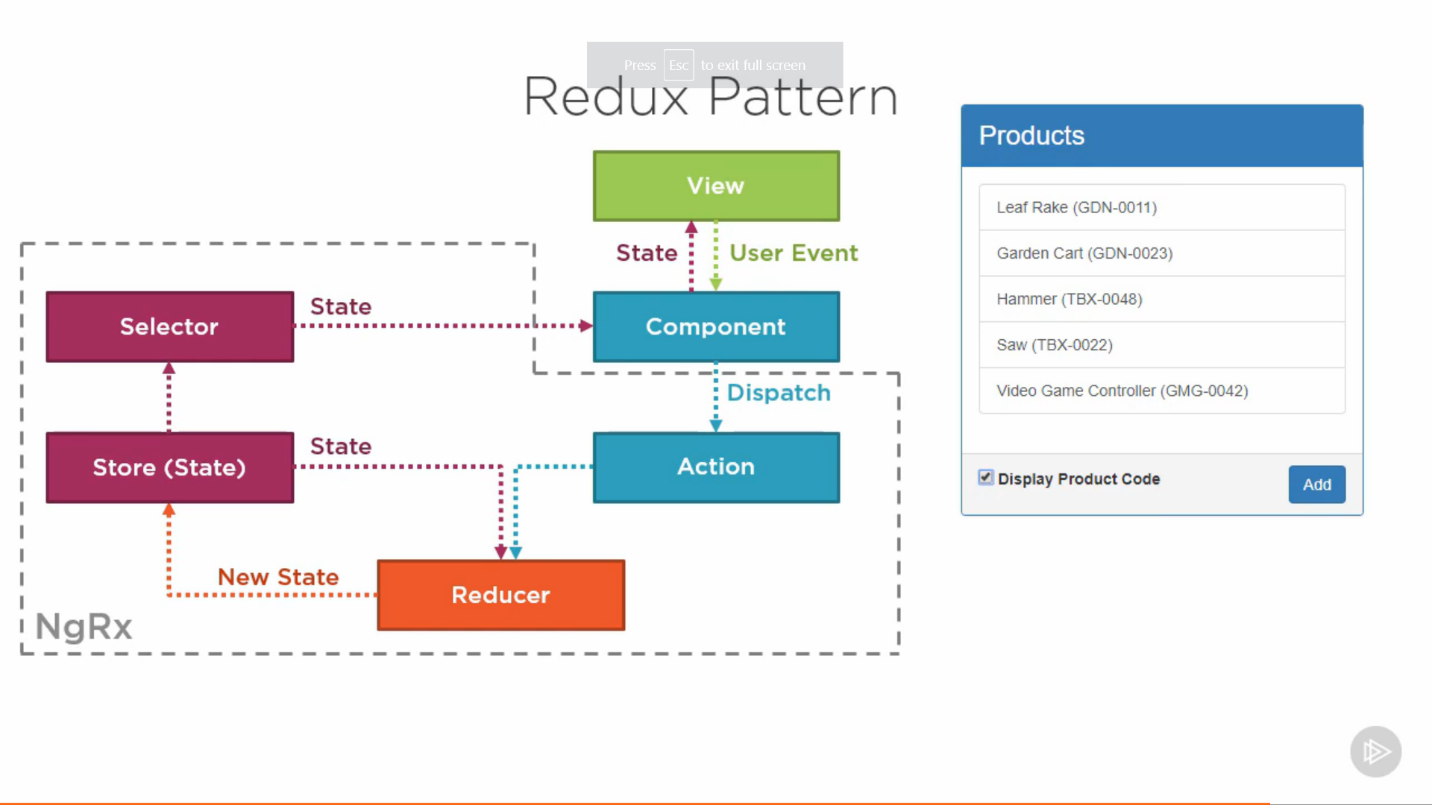
An Event emitter is a class defined in core module that can be used by components and directives to emit custom events.

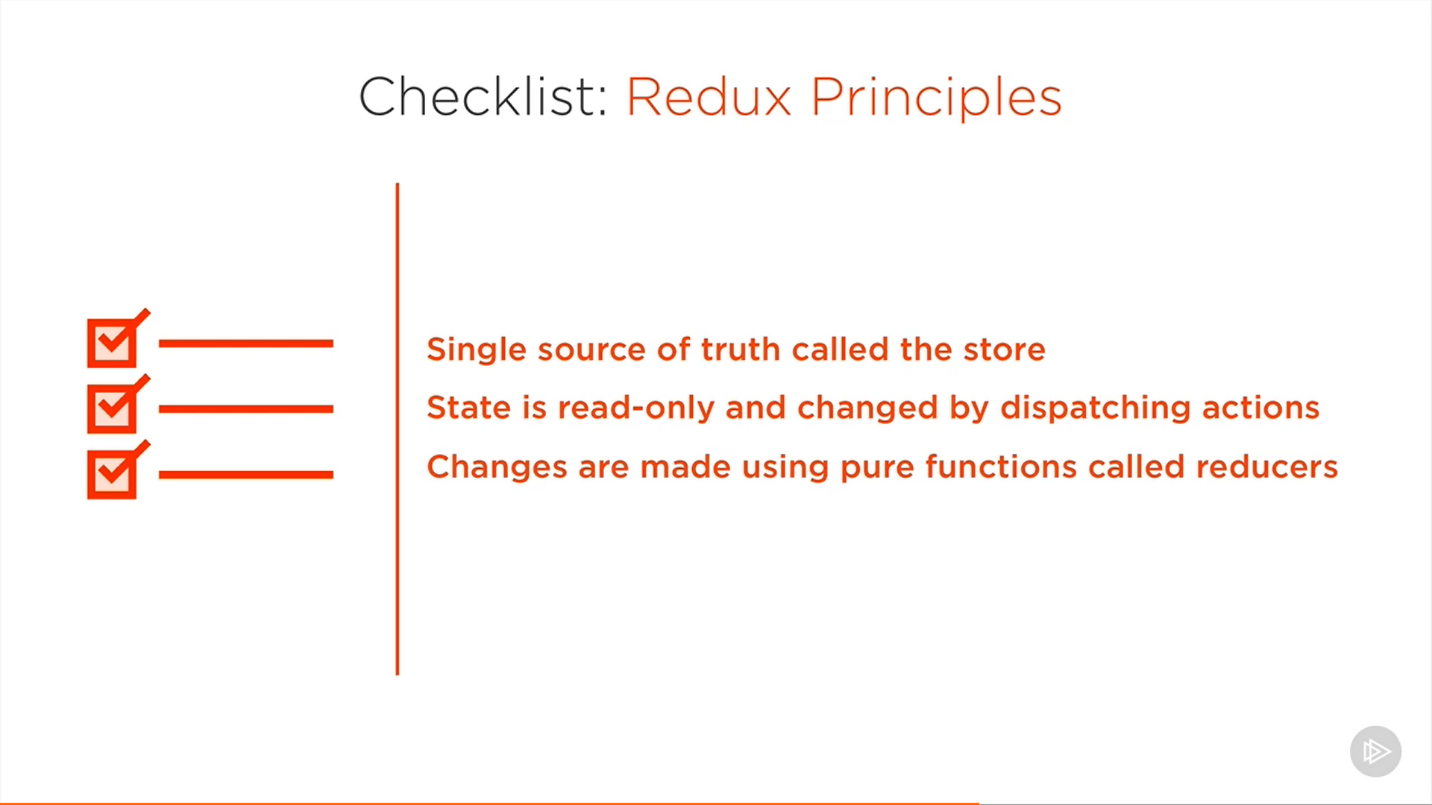
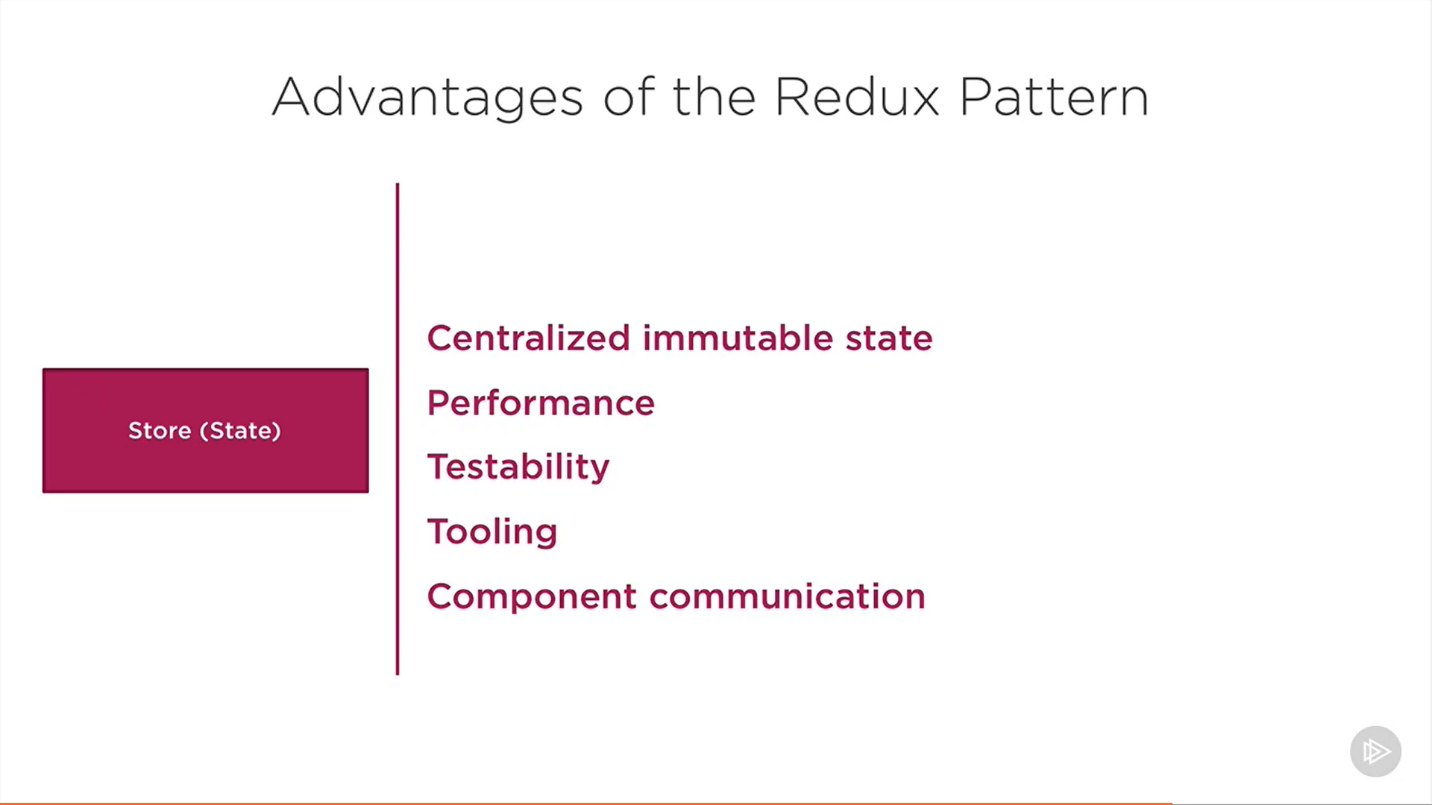
**Host Decorator:** <https://medium.com/frontend-coach/self-or-optional-host-the-visual-guide-to-angular-di-decorators-73fbbb5c8658>

**It is way to restrict the dependency search for given service.**

**angular.json:** <https://nitayneeman.com/posts/understanding-the-angular-cli-workspace-file/>

**How ngModel in angular works or implements 2-way binding?**

* **How observable subscribe to multiple events.**
* <https://stackoverflow.com/questions/36986548/when-to-use-asobservable-in-rxjs>
* **NGRX:**It is state management tool which has one-way data flow  
    
    
  Topics:  
  Reducers, Dispatchers, Store, ngRx Effects for side effects
* **Pure Functions:**   
  function which gives consistent results.  
  Ex. function(a, b){  
  return a + b;  
  }  
    
  Impure Functions:   
  function whose result is not consistent results.  
  Ex.   
    
  var c = 1;  
  function(a, b){  
  return a + b + c;  
  }

**Here result is not consistent because result depends on external variable.**

<https://github.com/sudheerj/angular-interview-questions>

**object grouping by property:**

<https://stackoverflow.com/questions/21776389/javascript-object-grouping>  
<https://medium.com/@edisondevadoss/javascript-group-an-array-of-objects-by-key-afc85c35d07e>  
**public groupReportListByCategory(collection, property) {**

**let i = 0;**

**let val;**

**let index;**

**const values = [];**

**const result = [];**

**for (; i < collection.length; i++) {**

**val = collection[i][property];**

**index = values.indexOf(val);**

**if (index > -1) {**

**result[index].reports.push(collection[i]);**

**} else {**

**values.push(val);**

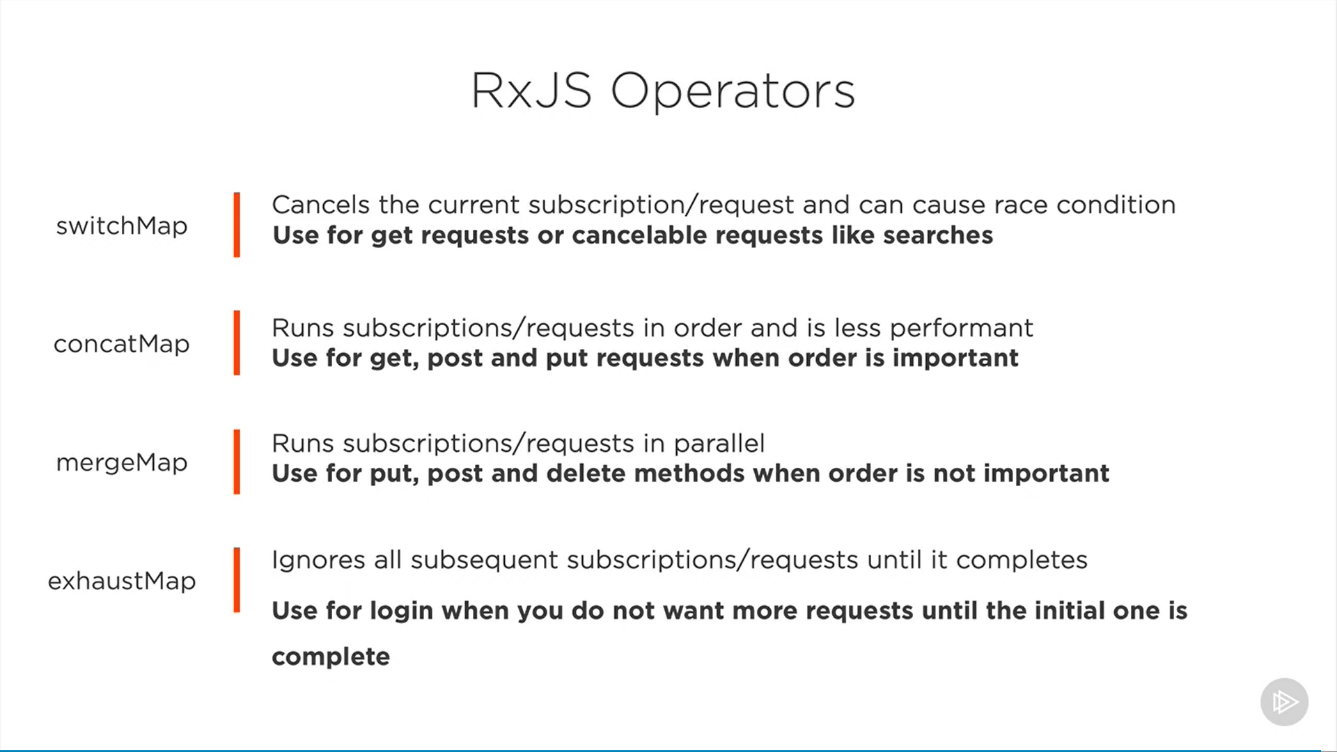
**result.push({ category: val, reports: [collection[i]] });**

**}**

**}**

**return result;**

**}**

* In general, an observable can return multiple values over time. An observable from HttpClient always emits a single value and then completes, never to emit again.
* **Pipe operator rxjs:** You can use pipes to link operators together. A Pipeable Operator is a function that takes an Observable as its input and returns another Observable. It is a pure operation: the previous Observable stays unmodified.  
  obs.pipe( op1(), op2(), op3(), op3(), )
* **Tap operator**: Intercepts each emission on the source and runs a function, but returns an output which is identical to the source as long as errors don't occur.
* A higher order observable is just a fancy name for an observable that emits observable.  
  <https://blog.angular-university.io/rxjs-higher-order-mapping/>  
  
* Async pipe vs subscription:  
  async pipe should be used when we are not using the response in component.
* Hide the HTML until get a value:   
  <div \*ngIf=”product$ | async as product”>  
  <p>{{product.name}}</p>  
  </div>
* <https://stackoverflow.com/questions/54771154/why-handle-errors-with-catcherror-and-not-in-the-subscribe-error-callback-in-ang>
* Use ErrorHandler: <https://angular.io/api/core/ErrorHandler#description>  
  and for http errors use catch errors in interceptors  
    
  **inside errorHandler use logic:**  
   handleError(error) {

if (error.url) {

// Http errors are handled by the error interceptor.

return;

}

if (!error.message) {

error.message = 'unknown-error';

}

console.error(error);  
  
// logic here how to handle  
}

**Inside interceptor:**  
intercept(request: HttpRequest<any>,

next: HttpHandler): Observable<HttpEvent<any>> {

if (request.url.indexOf(this.translateUrl) > 0) {

return next.handle(request);

}

return next.handle(request)

.catch((response: any) => {

/\* istanbul ignore else \*/

if (response instanceof HttpErrorResponse) {

// 416 / 419s are not errors we need to display to the user

// TODO: US20270 - 401 / 403s should be handled by the auth.interceptor

if (response.status == 400

|| response.status == 401

|| response.status == 403

|| response.status == 416

|| response.status == 419) {

return Observable.throw(response);

}

let errorMessage = "errors." + (response.error.errorCode || 'unknown');

return Observable.throw(new Error(errorMessage));

}

/\* istanbul ignore next \*/

return Observable.throw(response);

});

}

* Two folder approach:
  + Feature wise (recommended) : have other files folder related to feature inside main feature folder like having product-state folder inside product folder.
    - Easy to find related files
    - Follows angular style guide
    - Less cluttered
  + By function: have a separate function folder like having a common state folder and that contains the sub-feature states folder.
* Container Presentational pattern:
* ChangeDetectionStratergy :
  + Default: triggers chain detection cycle for every event trigger, requests, promises, xhr change etc.
  + onPush: triggers chain detection cycle for every event trigger but not for xhr(api events or promise based events) requests.   
    Benefits of onPush:
    - better performance
    - Skip change detection unless @input receives a new value or object reference.
    - It is easier to use when we follow presentational or container components.
* Benefits of index file:
  + Separation of concern
  + Cleaner code