**Angular Advance Topics**

**1. Dynamic Forms**

In angular we can create dynamic forms using raw data about form controls and validations from json or backend. For that first of all we have to get form controls data and based on type of control we can create control dynamically and set validations dynamically.

<https://www.danywalls.com/creating-dynamic-forms-in-angular-a-step-by-step-guide>

<https://angular.io/guide/dynamic-form>

**2. Server-Side Rendering**

By default, Angular renders applications only in a browser. Angular Universal allows Angular to render an application on the server, generating static HTML content, which represents an application state. Once the HTML content is rendered in a browser, Angular bootstraps an application and reuses the information available in the server-generated HTML.

With server-side rendering an application generally renders in a browser faster, giving users a chance to view the application UI before it becomes fully interactive.

<https://angular.io/guide/universal>

Command to run SSR : npm run dev:ssr

Other commands: <https://angular.io/guide/universal#useful-scripts>

**Build and Deployment**

Run command - npm run build:ssr

Create new folder for deployment and paste dist folder into it.

Create new site in IIS and point this new created folder.

Install IIS Node & URL Rewrite on the server.

IIS Node - <https://github.com/tjanczuk/iisnode/releases>

URL Rewrite - <https://www.iis.net/downloads/microsoft/url-rewrite>

Copy main.js file from dist>>projectName>>server and paste into new created folder.

Create web.config file.

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<system.webServer>

<handlers>

<add name="iisnode" path="main.js" verb="\*" modules="iisnode" />

</handlers>

<rewrite>

<rules>

<rule name="DynamicContent">

<match url="/\*" />

<action type="Rewrite" url="main.js"/>

</rule>

<rule name="StaticContent" stopProcessing="true">

<match url="([\S]+[.](jpg|jpeg|gif|css|png|js|ts|cscc|less|ico|html|map|svg))" />

<action type="None" />

</rule>

</rules>

</rewrite>

<staticContent>

<clientCache cacheControlMode="UseMaxAge" />

<remove fileExtension=".svg" />

<remove fileExtension=".eot" />

<remove fileExtension=".ttf" />

<remove fileExtension=".woff" />

<remove fileExtension=".woff2" />

<remove fileExtension=".otf" />

<mimeMap fileExtension=".ttf" mimeType="application/octet-stream" />

<mimeMap fileExtension=".svg" mimeType="image/svg+xml" />

<mimeMap fileExtension=".eot" mimeType="application/vnd.ms-fontobject" />

<mimeMap fileExtension=".woff" mimeType="application/x-woff" />

<mimeMap fileExtension=".woff2" mimeType="application/x-woff" />

<mimeMap fileExtension=".otf" mimeType="application/otf" />

</staticContent>

</system.webServer>

</configuration>

**3. Test cases (Jasmine/Karma)**

[\\rxmedia\Video\_Tutorials\Video\_Tutorials\Angular\Angular\_Unit\_Test\_case\_tutorial](file:///\\rxmedia\Video_Tutorials\Video_Tutorials\Angular\Angular_Unit_Test_case_tutorial)

**command to run test**

ng-test

ng test --include path

**command for code coverage**

ng test –code-coverage

ng test –code-coverage --include path

**linting**

eslint (<https://www.telerik.com/blogs/angular-basics-using-eslint-boost-code-quality>)

ng lint

**4. ngRx State management library**

**Store**

RxJS powered global state management for Angular apps, inspired by Redux. Store is a controlled state container designed to help write performant, consistent applications on top of Angular.

<https://ngrx.io/guide/store>

**Action**

Actions express unique events that happen throughout your application. From user interaction with the page, external interaction through network requests, and direct interaction with device APIs, these and more events are described with actions**.**

<https://ngrx.io/guide/store/actions>

**Reducer**

Reducers in NgRx are responsible for handling transitions from one state to the next state in your application. Reducer functions handle these transitions by determining which actions to handle based on the action's type.

<https://ngrx.io/guide/store/reducers>

**Selector**

Selectors are pure functions used for obtaining slices of store state.

<https://ngrx.io/guide/store/selectors>

**Effect**

Effects are an RxJS powered side effect model for store. Effects use streams to provide new sources of actions to reduce state based on external interactions such as network requests, web socket messages and time-based events.

<https://ngrx.io/guide/effects>

**NgRx DevTools**

Store Devtools provides developer tools and instrumentation for store.

<https://ngrx.io/guide/store-devtools>

[\\rxmedia\Video\_Tutorials\Video\_Tutorials\Angular\Angular\_NGRX](file:///\\rxmedia\Video_Tutorials\Video_Tutorials\Angular\Angular_NGRX)

**5. RxJS library advance**

**Higher order observables (Flattening operators):**

<https://rxjs.dev/guide/higher-order-observables>

Observables most commonly emit ordinary values like strings and numbers, but surprisingly often, it is necessary to handle Observables of Observables, so-called higher-order Observables

**concatMap**

Projects each source value to an Observable which is merged in the output Observable, in a serialized fashion waiting for each one to complete before merging the next.

<https://rxjs.dev/api/index/function/concatMap>

**mergeMap**

Projects each source value to an Observable which is merged in the output Observable.

<https://rxjs.dev/api/index/function/mergeMap>

**switchMap**

Projects each source value to an Observable which is merged in the output Observable, emitting values only from the most recently projected Observable

<https://rxjs.dev/api/index/function/switchMap>

**exhaustMap**

Projects each source value to an Observable which is merged in the output Observable only if the previous projected Observable has completed.

<https://rxjs.dev/api/index/function/exhaustMap>

**Custom Operator**

We can create custom operator by combining multiple built-in rxJs operators based on requirements.

<https://indepth.dev/posts/1421/rxjs-custom-operators>

**Error handling**

<https://rxjs.dev/api/index/function/catchError>

**catchError**

Catches errors on the observable to be handled by returning a new observable or throwing an error.

<https://rxjs.dev/api/index/function/catchError>

**retry**

Returns an Observable that mirrors the source Observable with the exception of an error.

<https://rxjs.dev/api/index/function/retry>

**Schedulers**

A Scheduler lets you define in what execution context will an Observable deliver notifications to its Observer.

A scheduler controls when a subscription starts and when notifications are delivered

<https://rxjs.dev/guide/scheduler>

**asyncScheduler**

async scheduler schedules tasks asynchronously, by putting them on the JavaScript event loop queue. It is best used to delay tasks in time or to schedule tasks repeating in intervals.

<https://rxjs.dev/api/index/const/asyncScheduler>

**queueScheduler**

Put every next task on a queue, instead of executing it immediately.

<https://rxjs.dev/api/index/const/queueScheduler>

**asapScheduler**

Perform task as fast as it can be performed asynchronously.

<https://rxjs.dev/api/index/const/asapScheduler>

**animationFrameScheduler**

Perform task when window.requestAnimationFrame would fire

<https://rxjs.dev/api/index/const/animationFrameScheduler>

**Subjects**

<https://rxjs.dev/guide/subject>

<https://www.tektutorialshub.com/angular/replaysubject-behaviorsubject-asyncsubject-in-angular/>

**Subject**

An RxJS Subject is a special type of Observable that allows values to be multicasted to many Observer

**Behaviour Subject**

A variant of Subject that requires an initial value and emits its current value whenever it is subscribed to.

**Replay Subject**

A variant of subject that "replays" old values to new subscribers by emitting them when they first subscribe.

**Async Subject**

A variant of Subject that only emits a value when it completes. It will emit its latest value to all its observers on completion.

**Multicasting**

Multicasting is the practice of broadcasting to a list of multiple subscribers in a single execution.

**share**

Share source among multiple subscribers

<https://rxjs.dev/api/operators/share>

<https://www.learnrxjs.io/learn-rxjs/operators/multicasting/share>

**shareReplay**

Share source and replay specified number of emissions on subscription.

<https://rxjs.dev/api/index/function/shareReplay>

**publish**

Share source and make hot by calling connect

<https://rxjs.dev/api/index/function/publish>

<https://www.learnrxjs.io/learn-rxjs/operators/multicasting/publish>

**connect**

Creates an observable by multicasting the source within a function that allows the developer to define the usage of the multicast prior to connection.

<https://rxjs.dev/api/index/function/connect>

**connectable**

An observable with a connect method that is used to create a subscription to an underlying source, connecting it with all consumers via a multicast.

<https://rxjs.dev/api/index/interface/Connectable>

**Hot and Cold Observables**

**Cold Observables**

When the data is produced by the Observable itself, we call it a cold Observable.

**Hot Observables**

When the data is produced outside the Observable, we call it a hot Observable.

<https://luukgruijs.medium.com/understanding-hot-vs-cold-observables-62d04cf92e03#:~:text=To%20understand%20the%20concept%20of,call%20it%20a%20hot%20Observable>.

**Backpressure and Flowcontrol**

When it comes to streaming data, streams can be overly chatty in which the consumer cannot keep up with the producer. To that end, we need mechanisms to control the source so that the consumer does not get overwhelmed. These mechanisms can come in either the form of lossy or loss-less operations, each of which depends on the requirements. For example, if you miss a few mouse movements, it may not be a problem, however, if you miss a few bank transactions, that could definitely be a problem.

<https://xgrommx.github.io/rx-book/content/getting_started_with_rxjs/creating_and_querying_observable_sequences/backpressure.html>

**debounceTime**

Emits a notification from the source Observable only after a particular time span has passed without another source emission

<https://rxjs.dev/api/index/function/debounceTime>

**sampleTime**

Emits the most recently emitted value from the source Observable within periodic time intervals

<https://rxjs.dev/api/index/function/sampleTime>

**throttleTime**

Emits a value from the source Observable, then ignores subsequent source values for duration milliseconds, then repeats this process.

<https://rxjs.dev/api/index/function/throttleTime>

**pausable** **approach**

pause emission for some time duration and resume after some time and vice versa.

**Buffer**

Buffers the source Observable values until closingNotifier emits.

<https://rxjs.dev/api/index/function/buffer>

**bufferCount**

Buffers the source Observable values until the size hits the maximum bufferSize given.

<https://rxjs.dev/api/index/function/bufferCount>

**bufferTime**

Buffers the source Observable values for a specific time period.

<https://rxjs.dev/api/index/function/bufferTime>

**windowTime**

Branch out the source Observable values as a nested Observable periodically in time.

<https://rxjs.dev/api/index/function/windowTime>

**Testing with Marble Diagrams**

<https://rxjs.dev/guide/testing/marble-testing>

**Combining and transforming observables**

**Combining observables**

These are Observable creation operators that also have join functionality -- emitting values of multiple source Observables.

<https://rxjs.dev/guide/operators#join-creation-operators>

**combineLatest**

Combines multiple Observables to create an Observable whose values are calculated from the latest values of each of its input Observables.

<https://rxjs.dev/api/index/function/combineLatest>

**combineLatestAll**

Flattens an Observable-of-Observables by applying combineLatest when the Observable-of-Observables completes.

<https://rxjs.dev/api/index/function/combineLatestAll>

**combineLatestWith**

Create an observable that combines the latest values from all passed observables and the source into arrays and emits them

<https://rxjs.dev/api/index/function/combineLatestWith>

**forkJoin**

Accepts an array of observableInput or a dictionary object of observableInput and returns an Observable that emits either an array of values in the exact same order as the passed array, or a dictionary of values in the same shape as the passed dictionary.

<https://rxjs.dev/api/index/function/forkJoin>

**zip**

Combines multiple Observables to create an Observable whose values are calculated from the values, in order, of each of its input Observables.

<https://rxjs.dev/api/index/function/zip>

**race**

Returns an observable that mirrors the first source observable to emit an item.

<https://rxjs.dev/api/index/function/race>

**concat**

Creates an output Observable which sequentially emits all values from the first given Observable and then moves on to the next.

<https://rxjs.dev/api/index/function/concat>

**merge**

Creates an output Observable which concurrently emits all values from every given input Observable.

<https://rxjs.dev/api/index/function/merge>

**Transforming observables**

Using these operators we can get data and perform required operation on data.

<https://rxjs.dev/guide/operators#transformation-operators>

**expand**

Recursively projects each source value to an Observable which is merged in the output Observable.

<https://rxjs.dev/api/index/function/expand>

**groupby**

Group into observables based on provided value

<https://www.learnrxjs.io/learn-rxjs/operators/transformation/groupby>

**partition**

Splits the source Observable into two, one with values that satisfy a predicate, and another with values that don't satisfy the predicate.

<https://rxjs.dev/api/index/function/partition>

**pairwise**

Groups pairs of consecutive emissions together and emits them as an array of two values.

<https://rxjs.dev/api/index/function/pairwise>

**scan**

Useful for encapsulating and managing state. Applies an accumulator (or "reducer function") to each value from the source after an initial state is established -- either via a seed value (second argument), or from the first value from the source.

<https://rxjs.dev/api/index/function/scan>

**pluck**

Select property to emit.

<https://www.learnrxjs.io/learn-rxjs/operators/transformation/pluck>

**Customizing observable creation**

Observables can be created with new Observable. Most commonly, observables are created using creation functions, like of, from, interval, etc.

<https://rxjs.dev/guide/observable>