Angular

Routing

https://www.techiediaries.com/routing-angular-router/

Directives:

https://www.freecodecamp.org/news/angular-directives-learn-how-to-use-or-create-custom-directives-in-angular-c9b133c24442/

forms:

https://edupala.com/how-to-implement-angular-template-driven-form/ - important

https://edupala.com/angular-reactive-form-validation/

https://www.itsolutionstuff.com/post/angular-13-template-driven-forms-with-validation-exampleexample.html

Services and HTTP

https://www.itsolutionstuff.com/post/angular-13-service-tutorial-with-exampleexample.html

Reactive forms

https://codecraft.tv/courses/angular/forms/model-driven/

custom validation

https://codecraft.tv/courses/angular/advanced-topics/basic-custom-validators/

Custom Directives

https://www.freecodecamp.org/news/angular-directives-learn-how-to-use-or-create-custom-directives-in-angular-c9b133c24442/#:~:text=Creating%20a%20custom%20directive%20is,Component%20decorator%20with%20%40Directive%20decorator.

Pipes

https://www.positronx.io/how-to-create-custom-pipe-in-angular-application/

@Input and @Output

https://dzone.com/articles/component-communication-in-angular-day-1

npm config set legacy-peer-deps true

ng add @ng-bootstrap/ng-bootstrap

Angular forms

Angular provides two approaches to building forms: template-driven forms and reactive forms. In this answer, I'll focus on template-driven forms.

Template-driven forms are built using Angular's built-in ngForm directive. To create a template-driven form in Angular, you typically follow these steps:

Import the FormsModule module into your module.

typescript

Copy code

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

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

import { CommonModule } from '@angular/common';

@NgModule({

imports: [

CommonModule,

FormsModule

],

declarations: [ ... ],

exports: [ ... ]

})

export class MyModule { }

Add a form element to your template and bind it to an instance of ngForm.

html

Copy code

<form #myForm="ngForm">

<!-- form controls go here -->

</form>

Add form controls to your template using the ngModel directive.

html

Copy code

<form #myForm="ngForm">

<label for="name">Name</label>

<input type="text" id="name" name="name" [(ngModel)]="myModel.name">

<label for="email">Email</label>

<input type="email" id="email" name="email" [(ngModel)]="myModel.email">

</form>

Handle form submission by adding a submit event handler to your form.

html

Copy code

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

<!-- form controls go here -->

<button type="submit">Submit</button>

</form>

typescript

Copy code

export class MyComponent {

myModel: any = {};

onSubmit() {

// handle form submission here

}

}

Reactive forms are built using the ReactiveFormsModule module, which provides a set of classes and directives that allow you to build and manage forms programmatically in your component code. To create a reactive form in Angular, you typically follow these steps:

Import the ReactiveFormsModule module into your module.

typescript

Copy code

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

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

import { CommonModule } from '@angular/common';

@NgModule({

imports: [

CommonModule,

ReactiveFormsModule

],

declarations: [ ... ],

exports: [ ... ]

})

export class MyModule { }

Create a FormGroup object in your component code.

typescript

Copy code

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

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

@Component({

selector: 'my-component',

templateUrl: 'my-component.html'

})

export class MyComponent {

myForm: FormGroup;

constructor() {

this.myForm = new FormGroup({

name: new FormControl(),

email: new FormControl()

});

}

}

Bind form controls to your template using the formControlName directive.

html

Copy code

<form [formGroup]="myForm">

<label for="name">Name</label>

<input type="text" id="name" formControlName="name">

<label for="email">Email</label>

<input type="email" id="email" formControlName="email">

</form>

Handle form submission by adding a submit event handler to your form.

html

Copy code

<form [formGroup]="myForm" (ngSubmit)="onSubmit()">

<!-- form controls go here -->

<button type="submit">Submit</button>

</form>

typescript

Copy code

export class MyComponent {

myForm: FormGroup;

constructor() {

this.myForm = new FormGroup({

name: new FormControl(),

email: new FormControl()

});

}

onSubmit() {

// handle form submission here

}

}

In this example, we have a simple form with two form controls: a name input and an email input. We create a FormGroup object in our component code, and we bind the values of the form controls to this group using the formControlName directive. We handle form submission using the (ngSubmit) event.

Note that with reactive forms, you have more fine-grained control over the state of the form and its controls. You can access and modify this state information using the FormGroup and FormControl classes provided by the ReactiveFormsModule module.

Angular Library

https://javascript.plainenglish.io/create-angular-library-2022-3965beee6dc6

https://indepth.dev/posts/1238/complete-beginner-guide-to-publishing-an-angular-library-to-npm

ng new my-library --create-application=false

ng generate library my-library --prefix=myLib

**ng g module new-lib**

**ng g c new-lib --project my-library**

**ng g application testing**

ng build my-library --configuration production

**npm login**

**npm publish**

**npm install my-library-sudha**

**How to use in a project.**

**In app.module**

import { MyLibraryModule } from 'my-library';

in component.html

<my-library-component></my-library-component>

ng new angular-pwa

ng add @angular/pwa

npm install --global http-server

ng build --configuration production

http-server -p 8080 -c-1 dist/project name

Interceptors

ng generate interceptor my-interceptor

Remember to import the interceptor in your module's providers array to make it available for use in your application.

Animation:

ngular provides a powerful animation module that allows you to create various animations within your application. These animations can be used to enhance user experience, create visually appealing transitions, and bring life to the user interface.

To use Angular animations, you need to import the **BrowserAnimationsModule** in your root module (**AppModule**) and inject it using the **imports** array.

Here's an example:

1. First, import the required animation-related modules:

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

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

Add the **BrowserAnimationsModule** in the **imports** array of your root module (**AppModule**):

@NgModule({

declarations: [

// Components, directives, etc.

],

imports: [

// Other modules

BrowserAnimationsModule // Import the BrowserAnimationsModule here

],

providers: [],

bootstrap: [AppComponent] // Your root component

})

export class AppModule { }

Now that the **BrowserAnimationsModule** is set up, you can create animations in your components or other parts of your application.

Angular provides various ways to create animations:

* **CSS Animations:** These animations are defined using CSS transitions or keyframes and can be triggered within Angular components using Angular's animation hooks.
* **JavaScript Animations:** You can use the Angular animation API to define animations programmatically in TypeScript/JavaScript.

Here's a simple example of creating a fade-in animation using Angular's animation API in a component:

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

import { trigger, transition, style, animate } from '@angular/animations';

@Component({

selector: 'app-my-component',

template: `

<div [@fadeIn]>This element will fade in</div>

`,

animations: [

trigger('fadeIn', [

transition(':enter', [

style({ opacity: 0 }),

animate('500ms', style({ opacity: 1 }))

])

])

]

})

export class MyComponent implements OnInit {

constructor() { }

}

indexedDB:

IndexedDB is a low-level API for client-side storage of significant amounts of structured data in web browsers. It's an alternative to using cookies, local storage, or session storage. IndexedDB is particularly useful when you need to store larger amounts of data, as it allows you to store complex objects and query that data.

In Angular, you can use IndexedDB to store and manage data within your application. To interact with IndexedDB in an Angular application, you'll typically follow these steps:

**Using IndexedDB in Angular:**

1. Install a wrapper or use the IndexedDB API directly.

You can choose to interact with IndexedDB by using the raw IndexedDB API or use third-party libraries that simplify working with IndexedDB in an Angular application. Libraries like "ngx-indexed-db" or "idb" provide Angular services and wrappers to interact with IndexedDB.

2. Install the necessary package(s).

If you opt for a library, install it via npm or yarn. For example, for **ngx-indexed-db**:

**npm install ngx-indexed-db**

Ngx-indexed-db is **a service that wraps the IndexedDB database in an Angular service**. It exposes a simple observables API to enable the usage of IndexedDB.

#### 3. Create a service to interact with IndexedDB.

Create an Angular service that encapsulates the IndexedDB functionality. This service will handle opening the database, creating object stores, adding, retrieving, updating, and deleting data.

Example using **ngx-indexed-db**:

**import { Injectable } from '@angular/core';**

**import { NgxIndexedDBService } from 'ngx-indexed-db';**

**@Injectable({**

**providedIn: 'root'**

**})**

**export class IndexedDBService {**

**constructor(private dbService: NgxIndexedDBService) {}**

**openDB() {**

**this.dbService.createDatabase(1, db => {**

**db.createObjectStore('myStore', { autoIncrement: true, keyPath: 'id' });**

**});**

**}**

**// Other methods like add, get, update, delete data from the database.**

**}**

**object store in indexeddb**

In IndexedDB, an object store is a fundamental concept and a core component for storing data. It's essentially where your data is stored within a database. An object store acts as a collection of JavaScript objects, similar to a table in a traditional relational database.

### Characteristics of Object Stores:

1. **Data Storage:** Object stores are used to store data. Each object store contains multiple items, and each item is a JavaScript object.
2. **Schema:** Object stores are defined with a schema. When creating an object store, you define the structure of the data that will be stored within it. This includes specifying the properties (fields) each object will have, as well as the key that uniquely identifies each object.
3. **Key-Value Pairs:** Every object stored within an object store is associated with a unique key, which allows for efficient retrieval and manipulation of the data.
4. **Indexed Properties:** IndexedDB also allows you to create indexes on object store properties to enable faster querying and sorting of the stored data.

**In IndexedDB, an index** is a way to improve the efficiency of querying and sorting data stored in an object store. It allows you to perform faster searches and retrievals on specific properties within the objects stored in an object store.

### Key Points about Indexes in IndexedDB:

1. **Improved Data Retrieval:** An index in IndexedDB is similar to an index in a book. It's a separate structure that enables faster lookups based on a specific property of the stored objects.
2. **Property-Based:** Indexes are created on properties within the object store. This means you can create an index on one or more properties of the objects stored in the object store.
3. **Efficient Queries:** Once an index is created, you can perform queries on that particular property without having to loop through all the objects in the object store, resulting in faster and more efficient data retrieval.
4. **Options:** When creating an index, you can specify whether the values within the indexed property are unique or not.

**In IndexedDB, a cursor** is an object used to iterate through multiple records in an object store or an index. It allows sequential access to multiple records, enabling you to perform various operations on these records such as reading, updating, or deleting them one by one.

### Key Points about Cursors in IndexedDB:

1. **Iterating Through Data:** Cursors are used to traverse through a range of data stored in an object store or an index.
2. **Sequential Access:** Cursors provide a way to sequentially access multiple records, enabling you to process data one item at a time.
3. **Support for Queries:** Cursors can be used for various types of queries, allowing more complex and specific retrieval of data.
4. **Handling Large Datasets:** Cursors are particularly useful for handling large datasets without having to load all data into memory at once.

**In IndexedDB, transactions** are a crucial aspect of handling data interactions within a database. They are used to group and manage database operations like reading or writing data, ensuring the integrity and consistency of data manipulation.

### Key Points about Transactions in IndexedDB:

1. **Atomic Operations:** Transactions ensure that a group of operations (reads or writes) either all succeed or all fail. This maintains the integrity of the database.
2. **Isolation:** Transactions provide isolation between different processes accessing the database concurrently. This prevents conflicts and ensures consistency.
3. **Data Modifications:** All data modifications (add, put, delete) within an IndexedDB occur within the context of a transaction.
4. **Commit or Rollback:** Transactions can be committed (saved to the database) or rolled back (canceled) based on the success or failure of the operations within them.