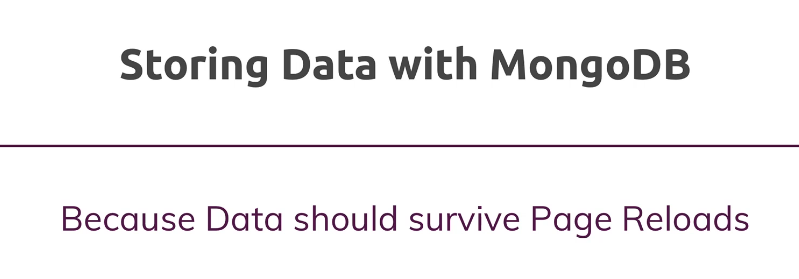
# Module Introduction

## Introduction



### We are handling data in angular app and storing the data locally, when we close the app the data is gone, we need to persist the data in a database. The Next step is to store the data from POST API into a mongoDB

### Observer the overall architecture and where we are going to work next and connecting to mongoDB

### 

## Topics covered

### What is MongoDB

### Comparing SQL & NoSQL

### Connecting Angular to Database

### Setting-up MongoDB, 2 options cloud or local

### Using MongoDB Atlas and IP Whitelist, or use local mongoDB

### Adding Mongoose

### Understanding Mongoose Schemas and Models

### Creating a POST instance

### Connecting our node express app to MongoDB

### Storing data in a database

### Fetching data from a database

### Transforming Response Data

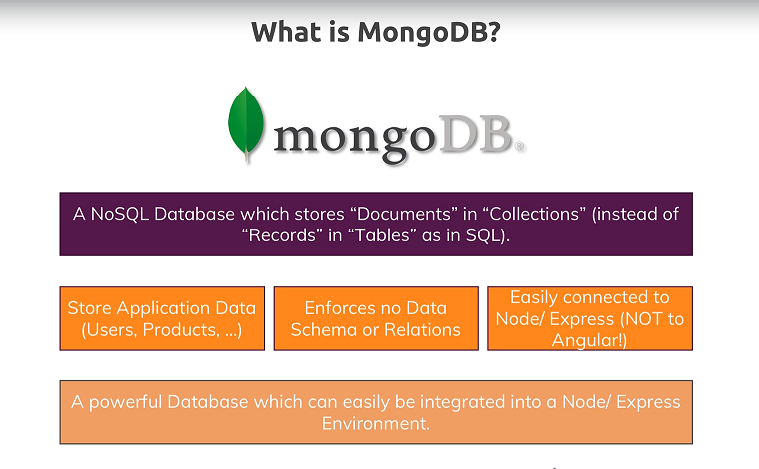
### Deleting Document

### Updating Front end after Deleting Posts

### Adding Posts with an ID

# What is MongoDB

## What is Mongo DB

* 1. 

## Why are we using mongodb and why are we using NOSQL db and not SQL

# Comparing SQL & NoSQL

## NOSQL VS SQL

### 

# Connecting Angular to Database?

## Can we connect our angular application directly to mongodb

### Technically we can connect our angular application to mongodb, but we should not do as it will pose security issues like connection string expose to clients. User authentication not possible at monodb level.

### 

# Setting-up MongoDB, 2 options cloud or local

## Wire up MongoDB

### There are 2 ways to wire up mongo dB

#### Download mongo from below link and install locally - <https://www.mongodb.com/download-center/community>

#### Using MongoDB cloud storage - MongoDB Atlas - Out of scope of this training but steps are mentioned in the next section.

## Setup local Mongo DB

### 

## Way to talk to local MongoDB

### In the previous step we setup the mongo dB, below are the different ways to connect and talk to mongo DB

### Mongo Shell -> Command line tool

### MongoDB Compass tool (GO to windows run , type “compass” and fin the tool) – This tool similar to MS-SQL client for MSSQL DB

#### 

#### <https://docs.mongodb.com/compass/current/>

### MongoDB Drivers

#### From any server side code we can use mongoDB drivers to connect to mongoDB. There are different drivers for each programming language.

#### Drivers can be downloaded from -> <https://docs.mongodb.com/drivers/>

#### 

### As we are using nodejs we can use below driver

#### <https://docs.mongodb.com/drivers/node/>

#### 

#### This uses mongoclient using which we can talk to mongoDB directly

#### <http://mongodb.github.io/node-mongodb-native/3.5/tutorials/collections/>

#### MongoDB Manual for CRUD operation

#### <https://docs.mongodb.com/manual/crud/>

## Talk to MongoDB using Mongoose

### For the scope of this training we are using a 3rd party package “mongoose” package which makes **accessing** **mongodb** **easier** and **convenient**

#### <https://mongoosejs.com/docs/>

# Using MongoDB Atlas and IP Whitelist, or use local mongoDB

### Go to below url

### <https://www.mongodb.com/download-center>

### 

### Sign-up for a free account

### If email already in use then use that to sign in

### 

### Or login with google

### 

### Once you login you will see below screen and should have a new project and option to create a new cluster( as highlighted below)

### 

### Click on build a cluster and choose free option

### 

### Choose AWS and other default options as highlighted below and click on create cluster

### 

### After clusters are created we should be seeing below screen

### Observe the highlighted areas and instruction messgages that creation is still in progress

### 

### After creation is successful, we should see below

### 

### It is a cluster because it is group of 3 DBs to have failover mechanism

### Next step is to create a user to access the DB

### 

### After clicking Add DB user, select read/write or admin access, enter username and click on autogenerate password. Store the password safely

### 

### After adding user you should see below screen

### 

### Next step is to add your local machine IP address to IP whitelist of mongoDB server

### Why we need to do this?

### **IP whitelisting** is a security feature often used for limiting and controlling access only to trusted users. **IP whitelisting** allows you to create lists of trusted **IP** addresses or **IP** ranges from which your users can access your domains.

### Add IP added of your local PC as shown below

### 

### 

### 

### 

### WARNING

### 

### In the next section we will install a package which helps us connect to the database

# Adding Mongoose

## Mongoose Introduction

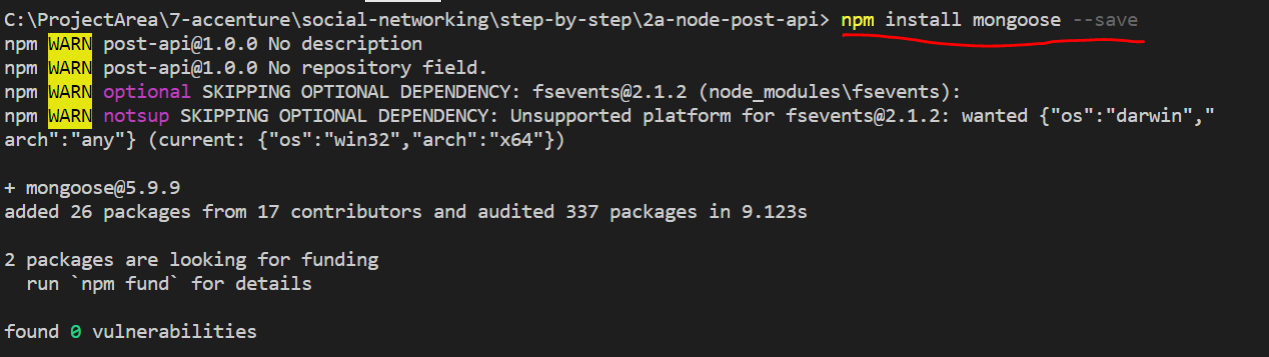
### Mongoose is a package which helps us connect to the database

### MongoDB doesn’t recommend having a schema as it is meant to store unstructured data. But in our case if we want to enforce schema, we need to do this at nodejs level. This is achieved by mongoose.

### Mongoose is built on top of mongodb driver

### Do not use mongoose if you are using unstructured data, off course mongoose will still work with unstructured data but not recommended

## Install mongoose using “npm install mongoose –save”

* 1. 

### This package will allow us to create schema, add and update data into mongo DB

# Understanding Mongoose Schemas and Models

## Create Mongoose Schema

### As we are enforcing a schema for our example by using mongoose, let’s create a schema as below in node project

### Create a folder called Model and create a file called posts.js

### 

### Mongoose schema takes a javascript object

### We can have different types of schema; schema is just a blueprint

### We create a model out of schema and export it for external use

### Code

const mongoose = require("mongoose");

const postSchema = mongoose.Schema({

  title: { type: String, required: true },

  content: { type: String, required: true },

});

module.exports = mongoose.model("Post", postSchema);

### Explain the code – Import statement, schema, required field property, export statement

### More about mongoose schemas below - <https://mongoosejs.com/docs/guide.html>

### 

# Creating a POST instance

## Let’s update POST API to use post schema and model created in the previous step

### NOTE : we still have not connected to database

### Import the “Post” Model in app.js

const Post = require("./models/post");

### Change the POST API code as below

### The code is creating a new Post model and accepting a javascript object in the constructor with values from the http post request body

app.post("/api/posts", (req, res, next) => {

  const post = new Post({

    title: req.body.title,

    content: req.body.content,

  });

  console.log(post);

  res.status(201).json({

    message: "Post added successfully",

  });

});

### Start the server if not started already

### 

### Verify the changes by initiating a new POST request from UI

### 

### Check the logs on the node server logs

### 

### Notice the auto generated id

# Connecting our node express app to MongoDB

## THEORY: Let’s save the above POST data object into mongoDB

### In order to save the data to DB we need to connect to mongoDB using below connection sting

### If you are using local mongoDB use below connection string

#### mongodb://localhost:27017/?readPreference=primary&appname=MongoDB%20Compass%20Community&ssl=false

### If you are using cloud then find the connection string using below steps

### Go to mongo cloud as mentioned in earlier steps and click on connect as shown below

### 

In the next window choose the option highlighted

### 

### In the next window copy the connection string

### 

### Example

### mongodb+srv://ashok:<password>@cluster0-ledyc.mongodb.net/test?retryWrites=true&w=majority

### replace the password with password created when creating a DB user.( refer earlier steps)

### use the connectionstring(either cloud or local) in the next step to proceed further.

## PRACTICLE: Import mongoose in app.js and connect to DB using below code using connection string

const mongoose = require("mongoose");

mongoose

  .connect(

    "mongodb://localhost:27017/?readPreference=primary&appname=MongoDB%20Compass%20Community&ssl=false"

  )

  .then(() => {

    console.log("Connected to database!");

  })

  .catch(() => {

    console.log("Connection failed!");

  });

## After including above code and as soon as server is restarted notice the below highlighted message from command line saying “connection to Database”

### 

# Storing data in a database

## Now the final step to save the data into DB is to include the below highlighted code in POST API

app.post("/api/posts", (req, res, next) => {

  const post = new Post({

    title: req.body.title,

    content: req.body.content,

  });

  post.save();

  console.log(post);

  res.status(201).json({

    message: "Post added successfully",

  });

});

## After the code change the execute the POST API thru POSTMAN tool and observer the data being stored in DB from mongodb compass tool

### 

### If we carefully observe the above the collection “posts” is getting created under “admin” db directly

### Lets change the connection string as below to store it under “MyPosts” DB

### "mongodb://localhost:27017/MyPosts?readPreference=primary&appname=MongoDB%20Compass%20Community&ssl=false"

### After the above change run the POST API again using POSTMAN tool and observe the new DB being created and “posts” collection under it

### 

### Observe that collection name is posts. This is because we created a mongoose model called “post” and the plural form of that is the name of the collection

### Another way to find out whether data was stored in DB or not is thru shell using commands as shown below

### 

# Fetching data from a database

## As we are able to store the data in DB , lets fetch it using GET API. Replace get api with below code.

app.get("/api/posts", (req, res, next) => {

  Post.find().then((documents) => {

    res.status(200).json({

      message: "Posts fetched successfully!",

      posts: documents,

    });

  });

});

## We can find more variations to GET APIs under mongoose documentation

### <https://mongoosejs.com/docs/api/model.html>

### 

## Verify from UI whether newly stored data is retrieved and displayed on front end or not

### 

# Transforming Response Data

### In this section we will transform the data retrieved from DB before we use them in UI or map them to UI model

### We look at the data at the node server console after post a data from UI , we see that data has “\_id” field which is an auto generated field from database

### 

### If we look at our post model in angular project , it has “id” field without underscore. This could pose problem when start using id field when we trying to the filed to either update or delete the post record

### 

### We could rename id to \_id, but instead of that we map “\_id” data returned from server to “id” field.

### To do this we can take advantage of great operators provided by observables

### The next topic is to explain where and how we can include operators

### If we observe the existing code the http.get observable stream fetches the data and provides to the data to subscription block. Before the data is handed over to subscription block we can include operators.

### We can include multiple operators and this can

### 

### As shown below pipe is a method provided by observable to include multiple operators

### 

### Now inside pipe block lets include the operator called map which can transform the data

### What map actually does is it receives an arugmet( http response ) array from observable stream and can convert every element in the array to a new element and store them all in a new array.

### 

### Final code to include in posts.service.ts of angular application (observe the highlighted changes compared to previous code )

### Import the below line at the beginning of posts.service.ts

import { map } from "rxjs/operators";

### replace the getPosts API in posts.service.ts with below code

  getPosts() {

    this.http

      .get<{ message: string; posts: any }>("http://localhost:3000/api/posts")

      .pipe(

        map((postData) => {

          return postData.posts.map((post) => {

            return {

              title: post.title,

              content: post.content,

              id: post.\_id,

            };

          });

        })

      )

      .subscribe((transformedPosts) => {

        this.posts = transformedPosts;

        this.postsUpdated.next([...this.posts]);

      });

  }

### After the above code change, we will have the correct ID which will be used in the next section to delete the Posts

# Deleting Document

## Let’s create DELETE API with help of below code. Include the below code in app.js of node project

### As the delete request requires identification of the document to be deleted we need to pass the identifier, The way to pass identifier is to include :id in the url of DELETE API

### Find the API in mongoose documentation which helps in achieving the deletion, the api used for deletion is deleteOne()

### deleteOne() takes a javascript as a parameter which identifies the record to be deleted, in our case it is { \_id: req.params.id }

### use the below code in node project app.js file to complete the API

app.delete("/api/posts/:id", (req, res, next) => {

  Post.deleteOne({ \_id: req.params.id }).then((result) => {

    console.log(result);

    res.status(200).json({ message: "Post deleted!" });

  });

});

## Execute the DELETE API in POSTMAN and observe the changes in GET API ( We can skip this if one want to directly integrate with Angular application and test it with UI)

### In order to delete a single post we need its ID

### 

### Pick up the ID from GET API response from above step

### Execute DELETE API as shown below from POSTMAN tool. The URL for DELETE API should include the id as shown below. This is as per REST API standards

### 

## Integrate the DELETE API with angular application

### In the angular application delete event is done on click of the delete button, hence lets add a handler to the click event of the delete button as shown below

### Make the below code changes (code not provided, type it out) in post-list.component.html

### Observe the highlighted changes.

### The delete handler takes an argument which identifies the record.

### This is was ID field which was transformed in the previous section.

### 

### After doing html changes add the handler function definition in the component (code not provided, type it out)

### 

### This handler needs to call the service to call actual delete API, Lets add a delete function to service layer posts.service.ts

### (code not provided for a reason, type it out as per screen shot)

### 

### Go back to post list component and call the service layer’s delete API as shown below

### 

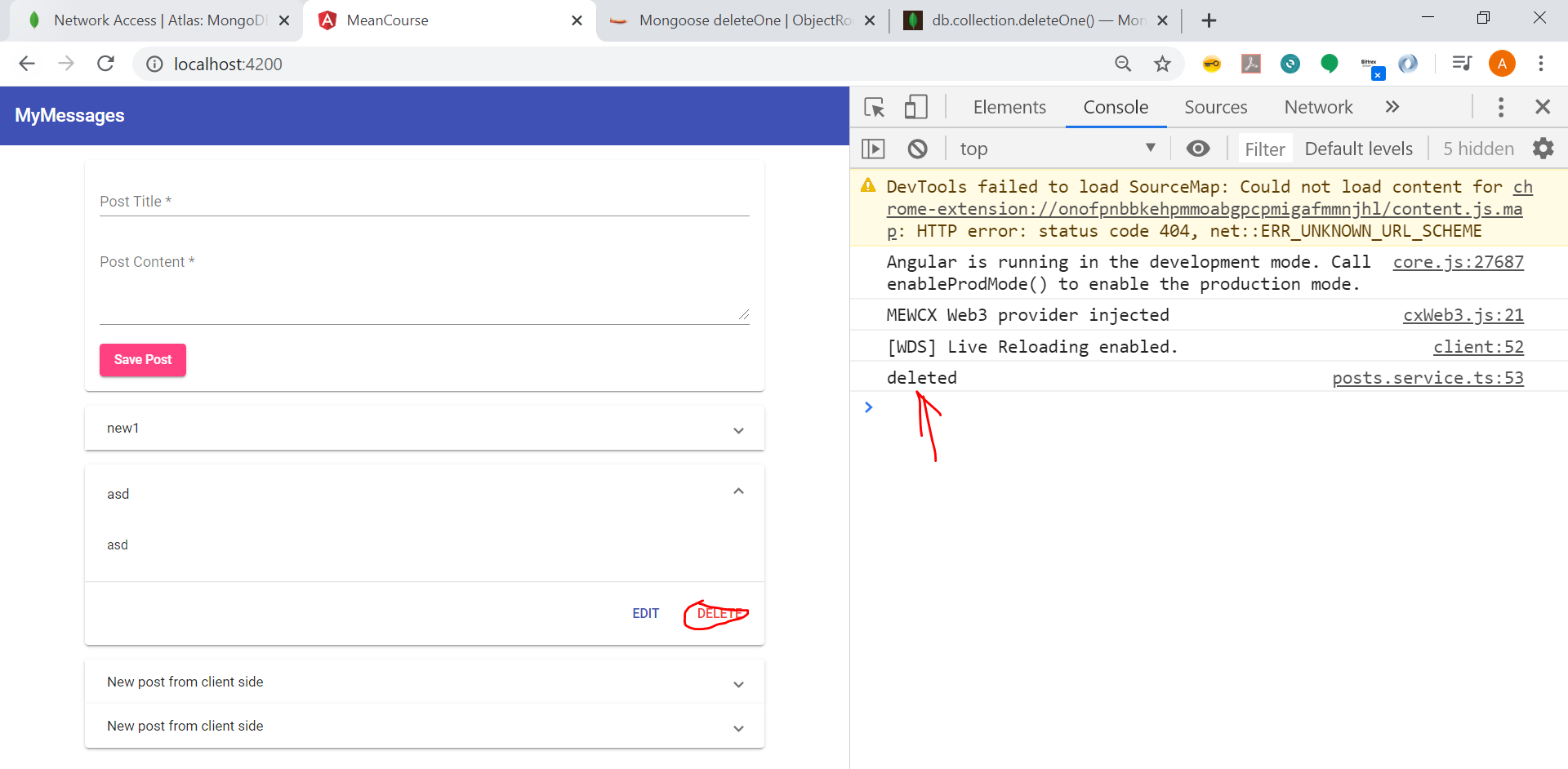
### Run the app and see you will receive a CORS issue.

### 

### This is not a CORS issue rather , URL is not formed properly after , we need a “/” after posts

<http://localhost:3000/api/posts5ea6024937ae535f4c8952da>

### Lets make the code change in service layer ( identify where to make) and execute again



Sever command line logs showing successful deletion



If you observe the deletion succeeds but the screen doesn’t refresh to reflect the deletion.

The next step is to do that

# Updating Front end after Deleting Posts

The front end doesn’t refresh after deleteting the data , we can fix this with below code.

Replace the deletePost function in posts.service.ts file with below code , explain the highlighted code

  deletePost(postId: string) {

    this.http

      .delete("http://localhost:3000/api/posts/" + postId)

      .subscribe(() => {

        const updatedPosts = this.posts.filter((post) => post.id !== postId);

        this.posts = updatedPosts;

        this.postsUpdated.next([...this.posts]);

      });

  }

# Adding Posts with an ID

### Delete all the posts on the UI and start over

### After adding first record without refreshing the page try to delete the post, we will not see anything deleted but a complex error on the server log.

### 

### 

### The reason for that is very first time when we create a post record we create with null as ID, observe the below code

### 

### I can refresh the page and reattempt deletion, it will succeed

### After successfully posting we do not refresh the local post record on the UI with the ID created in DB.

### This happens only for the first record not for others

### There are 2 ways to solve the issue

#### Get all fresh posts from DB everytime we create a new post

#### Or just update the local post with id which was received from DB.

### We will fix the issue by receiving the ID from DB as soon as the POST records are created

### Make code changes on the server to send the ID created . modify app.js file as below

### If we observe the existing code after post.save() we are not inspecting what response DB is sending. Do a console log and check

### After posting, the DB automatically includes the created record in the response, all we need to do is send the id from save() function response to UI

app.post("/api/posts", (req, res, next) => {

  const post = new Post({

    title: req.body.title,

    content: req.body.content

  });

  post.save().then(createdPost => {

    res.status(201).json({

      message: "Post added successfully",

      postId: createdPost.\_id

    });

  });

});

### Make the below code change to addPost function in posts.service.ts

  addPost(title: string, content: string) {

    const post: Post = { id: null, title: title, content: content };

    this.http

      .post<{ message: string; postId: string }>(

        "http://localhost:3000/api/posts",

        post

      )

      .subscribe((responseData) => {

        const id = responseData.postId;

        post.id = id;

        this.posts.push(post);

        this.postsUpdated.next([...this.posts]);

      });

  }

### Run the APP and verify the fix