# Module Introduction

## 

## Introduction

### After learning Angular, it is time to learn “EN” or MEAN course. Node is a javascript runtime to execute javascript code on the server. We can execute javascript code on server with additional features like handling requests, accessing DB, accessing file system Etc. with missing feature like accessing DOM compared to browser javascript.

### Express is a framework on top of node to make the development easier

### 

### In this course and future courses, we will build the server with below capabilities

#### Handling Requests

#### Business Logic

#### Persist data to a storage (Mongo DB)

#### Authenticate and Authorization.

## Topics

### Nodejs – Basics with “Hello World” Program

### Connecting Node & Angular – Theory

### What is a RESTful API

### Adding Node backend

### Adding Express Framework

### Improving Server.js code

### Fetching Initial Posts

### Using Angular HTTP Client

### Understanding CORS

### Adding POST Backend Point

### Adding Angular

# Nodejs – Basics with “Hello World” program

## Install node runtime from nodejs.org - <https://nodejs.org/en/download/>

### 

## Create a node project

### Create a node project folder by name “post-api”

### Inside the folder execute a command “npm init” to start a new node project

### 

### After the above step if we inspect the project structure we have single file called package.json created to start with

### 

### Open the project folder in visual studio code and observe the package.json

### 

### As node can execute a JavaScript code, lets write a simple hello world JavaScript program and execute inside node runtime. Add a file called server.js and write a simple code as shown below. NOTE: Please note this is not a server yet

### 

### Run command -> node server.js

### Observe the output

### 

## Let’s convert this to a server by importing http package

### Copy the below code to server.js file and execute command -> node server.js

// create a server variable

const http = require("http");

const server = http.createServer((req, res) => {

  res.end("This is my first response");

});

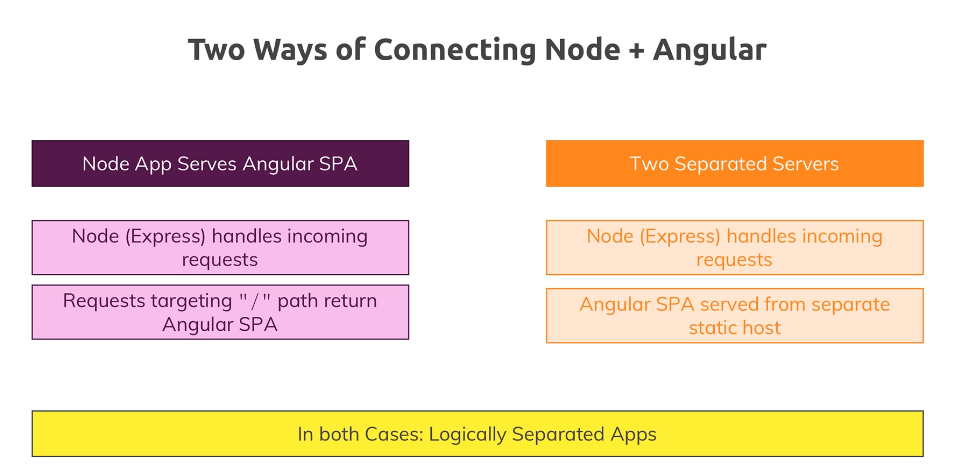
// listen at a port

server.listen(5001);

### Observe the response in browser

### 

# Connecting Node & Angular – Theory Explanation with an example



# What is a RESTful API and Evolution of REST API standards

## REST Stands for

### 

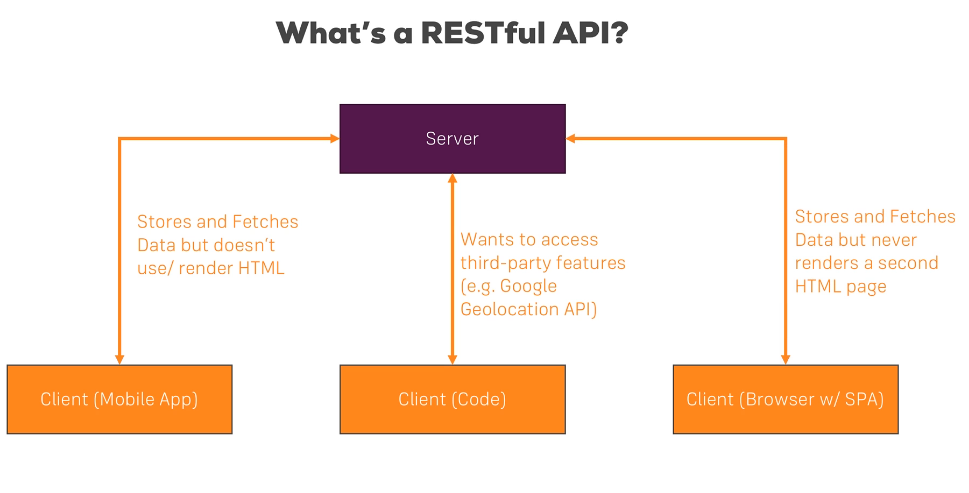
## Evolution of REST APIs standards

### The below architecture works for traditional applications.

#### 

### For Angular application implementing SPA(single page application) we use need a different client server communication pattern

#### There will be different types of clients with different needs – See the below use cases



### So, there was a need to keep the server stateless (not store the state of who called)

### When we scale the server stateless server will be beneficial

### As there are different client requesting the same data there standard way of **requesting the server across different client needs and standard way of sending the response back to different clients**

#### For example mobile client should not go with getAllPosts and SPA app going with getMyPosts

### This led to defining client server communication standards called REST

#### <https://restfulapi.net/>

### Standards defined REST URL pattern

#### <https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api>

## RESTful APIs with standards

### Standard URI,

### Standard HTTP Verb

### Standard Data format - JSON Data

### 

### In this module we will learn how to implement these REST APIs

# Adding Node backend

## THEORY: Adding Node server code in angular project structure

### In the earlier step we created a separate project for node. For convenience of demo we will create the server code also as part of the angular project, but we will run node server and angular server separately

## PRACTILCE: Adding Node server code in angular project structure

### As done earlier we will create a server and follow the below steps refer the screen shot

### Add folder called “backend” at root level where we are going to place all server code

### Add a file called server.js

### 

### Paste below code in server.js

// create a server variable

const http = require("http");

const server = http.createServer((req, res) => {

  res.end("This is my first response");

});

const port = process.env.PORT || "3000";

console.log("Running on port " + port);

// listen at a port

server.listen(port);

## RUN APP: Run the node server and verify

### 

### 

# Adding Express Framework

## THEORY: Adding Express

### As server needs to listen to http request and understand GET, POST, PATCH, DELTE request verbs and parse request data, we need to write a lot of code in nodejs to achieve this

### This is made easy by using another package called express

### Let’s listen to the client request using express package

## PRACTICLE: Adding Express and Listen to HTTP Requests with Express

### Let’s listen to the client request using express package

### Install Express by running command “npm install express --save”

### 

### Open the code is editor and notice package.json , express dependency is installed

### Simple ‘Hello World’ using express

### Make below changes to server.js and observer the highlighted areas (code is given below )

### 

### Include the below Code in server.js

var express = require("express");

var app = new express();

const port = process.env.PORT || "3000";

app.get("/", function (req, res) {

  res.send("Hello from Express");

});

app.listen(port, function () {

  console.log("Running on port " + port);

});

## RUN APP: After Adding Express Run the server and check the output

### 

### 

## THEORY: Adding middleware concept using express

### express is nothing but a chain of middleware’s

## PRACTICLE: Adding middleware concept using express

### Create a new folder called backend

### Create app.js file and include below code ( code provided in the next step )

### 

### Code – include in app.js

const express = require("express"); // this is a package which we installed

const app = express(); //express is a big chain of middlewares f1->f2->f3->f4

app.use((req, res, next) => {

  console.log("first middle where");

  next();

});

app.use((req, res, next) => {

  console.log("second middle where");

  next();

});

app.use((req, res, next) => {

  res.send("Hello from express");

});

module.exports = app;

### Modify the server,js file to include the above exported app object with below code

const http = require("http");

const app = require("./backend/app");

const port = process.env.PORT || "3000";

app.set("port", port);

const server = http.createServer(app);

console.log("Running on port " + port);

server.listen(port);

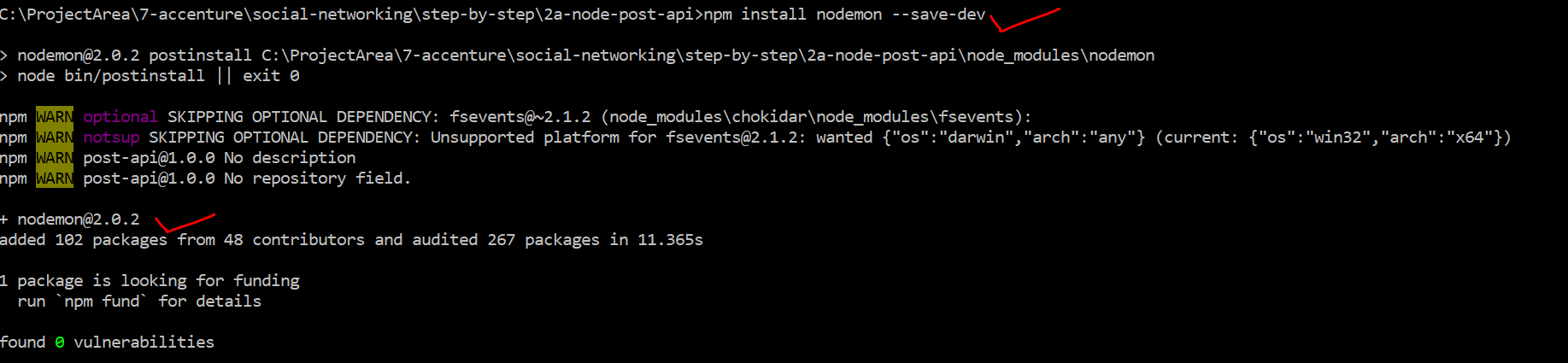
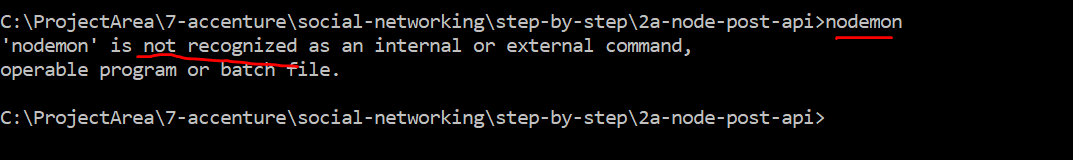
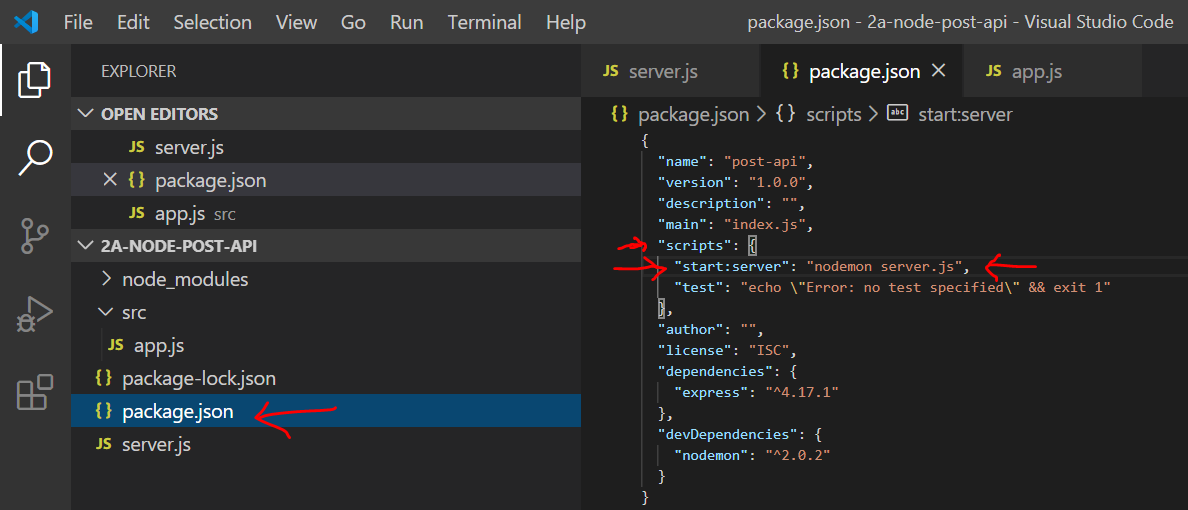
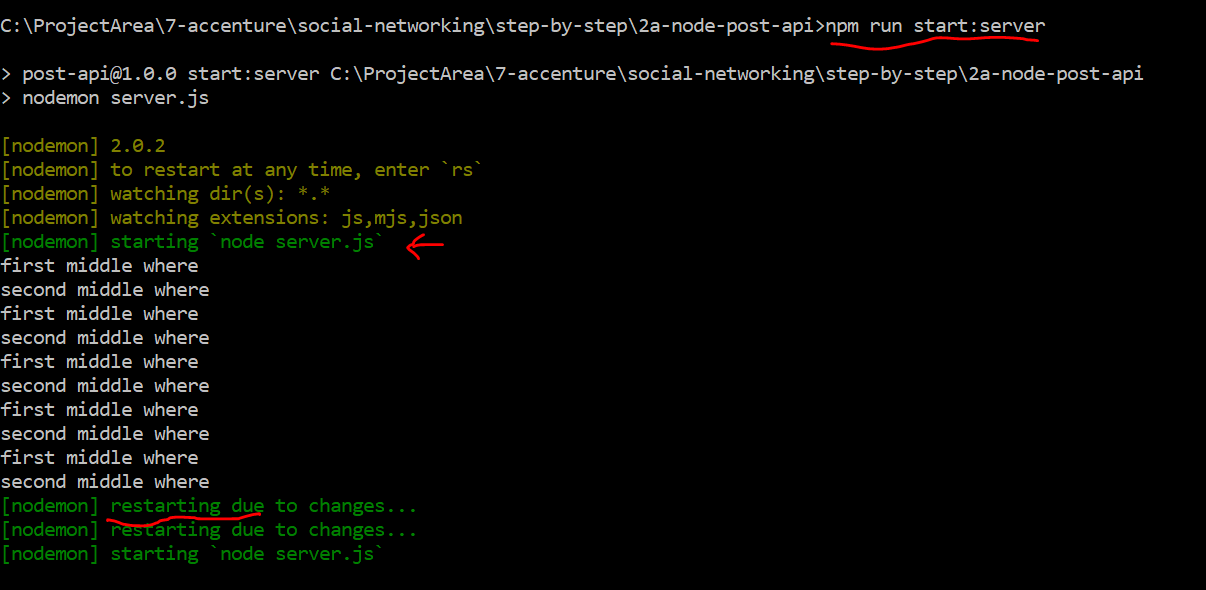
## RUN APP: After adding middleware Execute the project and see the output in browser and console

### Run the server

### 

### Request the server from browser

### 

1. Every time we are doing code changes, we are killing the server and restarting. If we have to make our server recompile and restart automatically whenever new code changes happen, we can do this with help of nodemon
   1. Run command -> npm install nodemon --save-dev
      1. 
   2. NOTE : nodemon is installed locally for this project only and it won’t run the command
      1. 
   3. In-order to run nodemon create a custom command as shown below in package.json under scripts section
      1. 
      2. Run the server and do some code changes and observe that server restarts automatically
         1. 

# Improving Server.js code

## Now we have basic node and express app setup, lets improve the server.js with below 2 improvements

## THEORY: Improve serve.js code to handle below items

### Handle Port number

### Handle Error

### Handle Request Listener

## PRACTILCE: Improve serve.js code by pasting the below code in server.js to achieve the above 3 improvements

const app = require("./backend/app");

const debug = require("debug")("node-angular");

const http = require("http");

const normalizePort = (val) => {

  var port = parseInt(val, 10);

  if (isNaN(port)) {

    // named pipe

    return val;

  }

  if (port >= 0) {

    // port number

    return port;

  }

  return false;

};

const onError = (error) => {

  if (error.syscall !== "listen") {

    throw error;

  }

  const bind = typeof addr === "string" ? "pipe " + addr : "port " + port;

  switch (error.code) {

    case "EACCES":

      console.error(bind + " requires elevated privileges");

      process.exit(1);

      break;

    case "EADDRINUSE":

      console.error(bind + " is already in use");

      process.exit(1);

      break;

    default:

      throw error;

  }

};

const onListening = () => {

  const addr = server.address();

  const bind = typeof addr === "string" ? "pipe " + addr : "port " + port;

  debug("Listening on " + bind);

};

const port = normalizePort(process.env.PORT || "3000");

app.set("port", port);

const server = http.createServer(app);

server.on("error", onError);

server.on("listening", onListening);

server.listen(port);

## RUN APP: After above improvement run the app and verify

### NOTE: After the code change every time we need to stop the server and rerun to pick up the code changes

### 

## THEORY: Use Nodemon to avoid server stop and start for every code change to reflect

### As we need to restart the server everytime we make code change to reflect the changes , this becomes tedious.

### We can avoid this by using a nodemon

### Every time we are doing code changes, we are killing the server and restarting. If we have to make our server recompile and restart automatically whenever new code changes happen, we can do this with help of nodemon

## PRACTICLE: Install Nodemon and setup

### Run command -> npm install nodemon --save-dev

### 

### NOTE: nodemon is installed locally for this project only and it won’t run the command

### 

### In-order to run nodemon, we need to create a custom command as shown below in package.json under scripts section

### 

## RUN APP: Run the server and do some code changes and observe that server restarts automatically

### 

# Fetching Initial Posts

## THEORY: GET API for post data

### In our angular app we need post object data from server. In order to serve the GET request from angular application lets create a GET API to respond with a hard-coded list of posts

## PRACTICLE: GET API for post data

### Include the below code in app.js and explain the code

const express = require("express");

const app = express();

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

  const posts = [

    {

      id: "fadf12421l",

      title: "First server-side post",

      content: "This is coming from the server",

    },

    {

      id: "ksajflaj132",

      title: "Second server-side post",

      content: "This is coming from the server!",

    },

  ];

  res.status(200).json({

    message: "Posts fetched succesfully!",

    posts: posts,

  });

});

module.exports = app;

## RUN APP: Verify GET API

### Verify GET API using URL <http://localhost:3000/api/posts>

### 

# Using Angular HTTP Client

## THEORY: Integrate Angular application with GET API

### Let’s connect Angular application to get post data from GET API

### Identify in angular app where to integrate the GET API

### How to Integrate GET API? using angular’ s inbuilt HTTP Client library

## PRACTILE: Integrate Angular application with GET API

### Import HttpClientModule at app.module.ts as shown below

import { HttpClientModule } from "@angular/common/http";

#### 

#### Once imported at module level, we can use httpClient anywhere in the module at components and services within the module

### Instead of using httpClient at post component level, it is a good practice to use at post service level as it can be used by other components too.

### In order to use it at posts.service.ts we need to import and inject the httClient using constructor as shown in below screen shot

import { HttpClient } from "@angular/common/http";

constructor(private http: HttpClient) {}

#### 

### After injecting the httpClient , we can use httpClient to fetch the data from GET API using below code.

#### Paste the below code by replacing existing getPosts() method in posts.service.ts (For demo do this step by step)

  getPosts() {

    this.http

      .get<{ message: string; posts: Post[] }>(

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

      )

      .subscribe(postData => {

        this.posts = postData.posts;

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

      });

  }

#### Explain the URI

#### Explain how subscribe works

#### Explain how positive and negative response from subscription are handled

#### Explain the response structure expected

#### API returns json but typescript has a javascript object to receive the response. The conversion from json to object is performed automatically by http get api function

#### We need to inform the parts of our app which are waiting for the post data (in our case post list component)

## RUN APP: Error Encountered, fix and Rerun

### If we try to run the app we see below error in command line ( compilation issue)

### 

### Go to below file and remove the this.posts as we are not expecting any return data ( explain this)

### 

### Run the app and check the UI output

### 

### 

### We don’t see any Posts on UI and if we see the console output we see an error

### 

## Explanation of the error which will be resolved in the next section

# Understanding CORS

## THEORY: Understanding CORS

### 

### Remember we are running client and service on different domains

### If they are running on same domain, we will not encounter this issue

### If the client request is coming from a different domain, we will encounter CORS error

### This is the default behavior and not a security issue

### As we want to allow our API to respond to clients from different domain, we need disable this default behavior by setting the headers at server side for the incoming requests

### Which headers are these? Let’s see that with an example

## PRACTILCE: Disable CORS using a middleware

### We need add a middleware to inspect every incoming request and set the request headers to allow inside for processing

### Paste the below code before app.use(“/api/posts”) in app.js

app.use((req, res, next) => {

  res.setHeader("Access-Control-Allow-Origin", "\*");

  res.setHeader(

    "Access-Control-Allow-Headers",

    "Origin, X-Requested-With, Content-Type, Accept"

  );

  res.setHeader(

    "Access-Control-Allow-Methods",

    "GET, POST, PATCH, DELETE, OPTIONS"

  );

  next();

});

#### Explain additional headers

#### Explain methods allowed (NOTE: PUT method is missing)

#### Explain next()

## RUN APP: Refresh the UI

After disabling CORS ( default behavior) using middleware the new posts from server should be served to angular application

### 

### We were successfully able to get data from server , in the next section lets try to post data to server

# Adding POST Backend Point

## THEORY: In this section we will implement API to POST the data from angular ui . The data which will be posted are “user posts”

### Notice the below screen shot (highlighted portions )

### 

### As we have only on middleware app.use(), all types of client requests(GET, POST…) are landing in app.use

### As we are implementing POST API, we need use specific middleware app.post()

### Similarly, we have middleware for GET, PUT, DELETE

### As we are not connected to DB yet, we will just implement receiving the request and perform console log

## PRACTICLE : Implement post API with app.post() middleware and body-parser package

### POST API should accept data from POST http request body

### The POST http request body data need to be extracted from the request and parsed made available in request variable at the server side. This is performed by a middleware package called “body-parser”. Let’s install “body-parser” with below command

### npm install –save “body-parser

### 

### This is a node express package, which is used as middleware.

### Let’s start building the POST API with below code

### Screen shot of where to add the code (code is given below)

### 

### Import the package in app.js

const bodyParser = require("body-parser");

### Let’s attach this middleware into our request pipeline with below code before CORS middleware

// Lets attach the body-parser middleware

// bodyParser.json() -> this will tell only to process json type data from the request body

app.use(bodyParser.json());

//another example showing body-parser can process other types of body other than json

app.use(bodyParser.urlencoded({ extended: false }));

### Add below post API code after CORS middleware

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

  const post = req.body;

  // we still need to send the response as we dont want client be waiting and timeout

  res.status(201).json({

    message: "Post added successfully",

  });

});

### Change app.use() to app.get to be specific ( there are issue if we are not specific as app.use might behave like a catch all bucket)

### Start the server using command

### 

## RUN APP: Testing the POST API

### As we can not test the POST API in browser without UI app , there are 2 way to execute POST API

### One is connecting the POST API to our angular app which we will do in the next section

### Second one is to use POSTMAN tool and test the POST API before integrating it with angular app ( this is optional )

### Install POSTMAN tool from below site- <https://www.postman.com/downloads/>

### Postman tool will be used perform all types http requests

### 

### For example: At this point we have GET API and that can be executed in POSTMAN as below

### 

### Initiate the POST request in POSTMAN and observe the response

### 

# Adding Angular

## THEORY: Integrate POST API with angular

## PRACTICLE: Integrate POST API with angular

### Replace the addPost method of posts.service.ts file in angular application with below code

  addPost(title: string, content: string) {

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

    this.http

      .post<{ message: string }>("http://localhost:3000/api/posts", post)

      .subscribe(responseData => {

        console.log(responseData.message);

        this.posts.push(post);

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

      });

  }

### Explain the url

### Explain the subscription

### Explain the response data

### Explain posted data logging at server side and response log at client side

### Explain local post data being updated only after successful server-side response, in case of error the local post data will not be updated, as optimistic update is happening inside success block of subscribe which is asynchronous by default.

### There will be an error in code highlighted below

### 

### Add id field to post model in post.model.ts file to resolve the above error

### 

## RUN APP: Start both servers is they are not

### 

### 

### POST the data from UI

### Before POSTing

### 

### After POST succeeds

### 

### We can see the data posted on the server

### 