Season-02 Namaste Node.js

16/09/2024

Waterfall Model (SDLC)

Requirements → (PM + Designer)
↳ Design → Senior Engineer/EM
↳ Development → SDE1, SDE2
↳ Testing → SDET
↳ Deployment → DevOps
↳ Maintainance.

Monolith vs. Microservices

→ Dev Speed
Dev speed is slow in monolith because a lot of people working on same repo.

→ CodeRepo
Monolith has huge, sigle codeRepo.
Difficult to maintain.

→ Scalability
Very difficult to scale monolith architecture.

→ Deployment
single deploment in monolith.
can be a pro or cons.

→ Tech Stack
Restricted in monolithic

→ Infra Cost
more in micro service

→ Complexity
As you grow complexity increases in monolithic.

→ Fault Isolation

→ Testing
Microservices has separate test cases.
Testing in Monlithic is bit easier.

→ Ownership
Central ownership in monolith

→ Maintainance
→ Re wamps
→ Debugging
→ Dev Experience.

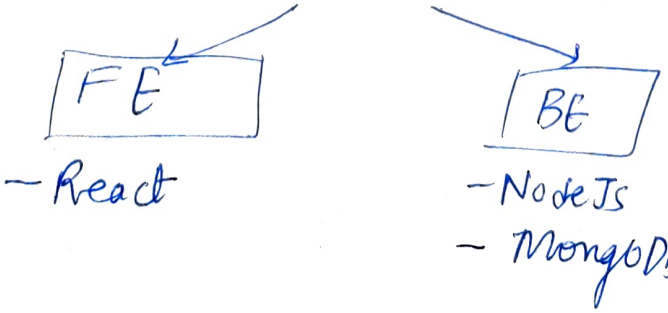
Episode-02 | Features, HLD, LLD & Planning

Product requirement (done by product managers)

① Create an account
② Login
③ Update your profile
④ feed page - explore
⑤ Send connection Request
⑥ See our Matches
⑦ See the request we have sent/recieved
⑧ update your profile

Tech planning

2 Microservices



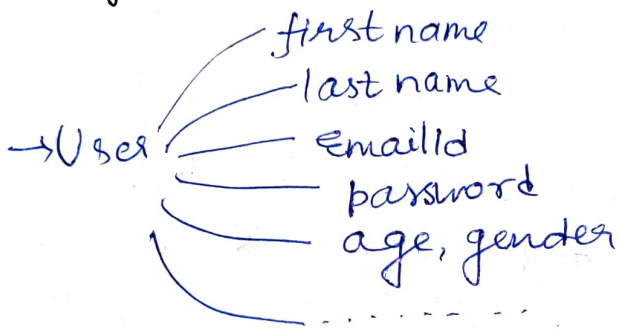
"If you spend a lot of efforts in planning, writing code becomes very easy".

"If your planning is good, you don't have to refactor it again and again".

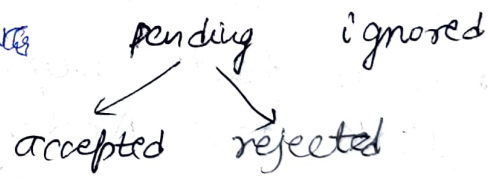
LLD (Low level Design)

DB Design
API design / Document the APIs.

DB Design



→ ConnectionRequest
- fromUserId
- toUserId
- status = PENDING



API Design {REST API}

GET, POST, {PUT, PATCH}, DELETE
↳ update

HTTP methods

PUT replaces an entire resource, while PATCH updates only part of a resource.

CRUD operations

POST	/signup	PATCH	/profile
POST	/login	DELETE	/profile
GET	/profile	POST	/sendRequest → ignore / interested
POST	/profile	POST	/review → accept / reject

GET /requests

GET /connections

Episode-03 | Creating our Express Server

30/09/2024

Version

4.19.2
↑ ↑ ↑
major minor patch

^ 4.xx
~ 4.5.x

How to open terminal in VS code? ctrl + n

npm init → it will initialiaze your project.
this will create a package.json file for you.
Package.json is like a index for your project.
Create a src folder and inside this a file named app.js
⇒ App.js is the starting point of your application.
⇒ Here we will initialiaze our app.

Now, we will use express.js to create our server.
Server can listen to allside request and respond
Express is fast unopiniated, minimalist web framework for Node.js.

npm i express.

⇒ Now, two files will appears → i) Node modules
ii) package-lock.json

Dependency → Dependency is any package that our project is dependend on.

How to handle request?

```
app.use((req, res) => {
    res.send("Hello from server");
});
```

This function is called request handler.

If we want that our server respond to only a particular route then,

```
app.use("/hello", (req, res) => {
    res.send("Hello, hello, hello");
});
```

To install nodemon at global level →

```
sudo npm i -g nodemon
```

(Because you are trying to install at global level which requires admin permit)

In package.json, edit the script →

```
"scripts": {
    "start": "node src/app.js",
    "dev": "nodemon src/app.js"
}
```

Code

```
const express = require("express");
const app = express();
app.use("/", (req, res) => {
    res.send("Namaste Akshay!");
});
app.use("/hello", (req, res) => {
    res.send("Hello hello, hello!");
});
app.use("/test", (req, res) => {
    res.send("Hello from the server");
});
app.listen(7777, () => {
    Console.log("server is successfully running");
});
```

30/09/2024

Episode-04 || Routing and Request Handlers.

How to push your code on git?

git init

What if you delete node modules?

Don't panic, just do

npm install.

As long as you have package.json and package lock.json, you can recreate your node modules.

That's why we never push node modules to github.

To tell not to track some folder,
on the root of project →

.gitignore
node_modules

How to commit changes?

git add .
git commit -m "created a express server"

Create a new repository on github, then on terminal copy
git remote add origin git@github.com:harshguptadev/devTi
git branch -M main
git push -u origin main.

Q should we push package-lock.json onto github?

Wild card in route handlers

```
app.use("/", (req, res) => {
        res.send("Hello");
});
```

⇒ Now any route starting from "/" will be overridden by this about request handler if we write this route first.

⇒ If you route is starting from "/", then anything starting with "/" will be matched to this.

like "/hello", "/hello/1" will give same results if a route is formed for "/hello".

⇒ Just by changing the order of code, a lot of things changes.

⇒ The code will start matching the routes from the top.

"Order of routes matter a lot".

HTTP Methods

Get, put, post, patch, delete

How to handle different method, differently in our apis?

Instead of app.use, app.get

this will match all the HTTP method API call

this will handle only get.

Advanced things about routing

/abc

/ab?c ⇒ this means 'b' is optional over here

/ab+c ⇒ /abc, /abbc, /abbbbc

/ab*cd ⇒ /abcd, /abharshicd, /abguptacd

/a(bc)?c ⇒ /abcc, /ac

/a(bc)+c ⇒ /abcbcc, /abcc

You can also write regex over here

/a/ → means it should contain a

/.*fly$/

How to get query params in our route handler

```
console.log(req.query);
http://localhost:7777/user?userid=101
```

How to make routes dynamic?

~~http://localhost:7777/user/:id~~

```
app.get("/user/:userId", (req, res) => {
    console.log(req.params);
});

app.get("/user/:userId/:name/:password",
    (req, res) => {
        console.log(req.params);
        res.send
    });
```

Episode-05 || Middlewares and Error Handlers

One Route can have multiple route handlers.

Code-1

```
app.use("/user", (req, res) => {
        console.log("Handling the route user!!");
        res.send("Response!!");
    },
    (req, res) => {
        console.log("Handling the route user 2!!");
        res.send("2nd Response!!");
    }
);
```

O/p → Response.

Code-2

```
app.use(...
    ...
    //res.send("Response!!");
},
(req, res) => {
    }
});
```

O/p → infinite loop.

```
(req, res, next) => {
```

now it will go to next route handler.

```
}
```

You can also create array of route handlers.

You can define by this way also →

```
app.get("/user", (req, res, next) => {
    console.log("Handling the route user!!");
    next();
});
app.get("/user", (req, res, next) => {
    console.log("Handling the route 2");
    res.send("2nd Route Handler");
});
```

Generally we create middleware by →

app.all(↳ explore this

```
app.use("/admin", (req, res, next) => {
    const token = "xyz";
    const isAdminAuth = token === "xyz";
    if (!isAdminAuth) {
        res.status(401).send("Unauthorized request");
    } else {
        next();
    }
});
```

Another way is to create a middleware folder.

```
const { adminAuth, userAuth } = require("./middlewares");
app.use("/admin", adminAuth);
app.get("/user", userAuth, (req, res) => {
});
```

Error Handling

use try catch.

But still if some error is unhandled, you can handle them by →

at the last

```
app.use("/", (err, req, res, next) => {
    if(err){
        //log your error
        res.status(500).send("Something went wrong");
    }
});
```

01/10/2024

Episode-06 || Database, Schema & Models || || Mongoose ||

How to connect our app to database?

create a config folder inside src folder.

create database.js.

npm i mongoose.

database.js

```
Const mongoose = require('mongoose');
mongoose.connect(
        string,
);

Const connectDB = async () => {
    await mongoose.connect(
        "string          /devTinder"
    );
};
connectDB().then(() => { //log })
    .catch(err => {"log"});
```

app.js

```
require("./config/database");
```

⇒ First connect to the database, then listen to the server.

So,

database.js

```
const mongoose = require("mongoose");
const connectDB = async() => {
    await mongoose.connect(
        "_____/devTinder"
    );
};
module.exports = connectDB;
```

app.js

```
const express = require("express");
const connectDB = require("./config/database");
const app = express();

connectDB().then(() => {
  console.log("DB connection established...");
  app.listen(7777, () => {
    console.log("Server listening...");
  });
}).catch((err) => {
    console.error("Database cannot be
                   connected");
});
```

Creating the Schema

In src, create a models folder.

```
// first we will create a schema
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
    firstName: {
        type: String
    },
    lastName: {
        type: String
    },
    emailId: {
        type: String
    },
    password: {
        type: String
    },
    password: {
        type: String,
    },
    age: {
        type: Number
    },
    gender: {
        type: String
    }
});
```

//Now we will create mongoose model

```
const User = mongoose.model("User", usersSchema);

module.export = User;
```

Now, we will add our first data to our database.

app.js

```
const User = require("./models/user");
app.post("/signup", async (req, res) => {
    const userObj = {
        firstName: "Akshay",
        lastName: "Saini",
        emailId: "akshay@saini.com",
        password: "akshay@123"
    }
    // creating a new instance of the User model
    const user = new User(userObj);
    try {
        await user.save(); // this function will return
                           // you a promise.
        res.send("User Added sucessfully");
    } catch(err) {
        res.status(400).send("Error saving the user" + err.message);
    }
});
```

devTinder is database
User is a collection
and it contains documents.

Always wrap the db operations in try catch block..

01/10/2024

Episode-07 | Diving into the APIs

```
app.use(express.json()); // to parse the json data
                         // from request object.

Now, you can perform.
console.log(req.body);

const user = new User(req.body);
```

Episode-08 | Data Sanitization & Schema Validation

```
gender: {
    type: String,
    validate(value){
        if(!["male","female","others"].includes(value)){
            throw new Error("Gender data is not valid")
        }
    },
},
```

// This validate function will only work if you create some new object.

// You have to enable it to run on updates also.

In the patch api, in the option section

```
runValidators: true.

await User.findByIdAndUpdate({_id:userId}, data,
    { returnDocument: "after",
      runValidators: true,
    });
```

API level validation

```
app.patch("/user/:userId", async (req, res) => {
    const userId = req.params?.userId;
    const data = req.body;
  try {
   const ALLOWED_UPDATES = ["photoUrl", "about", "gender",
                            "age", "skills"];
   const isUpdateAllowed = Object.keys(data).every((k) =>
                            ALLOWED_UPDATES.includes(k));
   if (!isUpdateAllowed) {
       throw new Error("Update not allowed");
   }
```

⇒ So this was the code, how you handle that user can update only selected items later on.

⇒ Never trust the user data. Always apply all the validations to your data.

```
   if (data?.skills.length > 10) {
     throw new Error("skills cannot be more than 10");
   }
```

Always have backend validation.
Attackers can steal your API and can break your DB.

⇒ This is called Data Sanitization.

How to validate emailId?

You can take help of a external library called npm validator.

npm i validator

Two level of validation

1) Schema or DB level validation
2) API level validation.

Schema

```
const validator = require('validator')

emailId: {
    type: String,
    lowercase: true,
    required: true,
    unique: true,
    trim: true,
    validate(value){
        if(!validator.isEmail(value)){
            throw new Error("Invalid email address:" + value);
        }
    }
},
```

Similarly you can add a lot of validators

"Never Trust req.body"

- isValidUrl
- isStrongPassword.
- isJWT
- isLowerCase
- isMobileNo.

Episode 09 | ~~End~~ Encrypting Password

01/10/2024

An attacker can send any malicious data into the API, so never trust req.body.

Signup api is the entry point of our app, it should be most secure.

```
// validation of data
        ⇓
// Encrypt the password
        ⇓
// create the new instance of the model
        ⇓
// Save the model to the database
```

Now, we will install bcrypt library for encrypting our password.

```
app.post("/signup", async(req, res) => {
    try {
        validateSignupData(req);
        const { firstName, lastName, email, password} =
                                        req.body;

    // Encrypt the password
    const passwordHash = await bcrypt.hash(
                                    password, 10);
    console.log(passwordHash);
    // Creating a new instance of the User model
    const user = new User({
        firstName, lastName, email,
        password: passwordHash
    });
    await user.save();
    res.send("User Added successfully");
```

Login API

```
app.post("/login", async(req, res) => {
    try {
    const {emailId, password} = req.body;
    const user = await User.findOne({emailId: emailId});
    if(!user) {
      throw new Error("EmailID is not present in DB");
    }
    const isPasswordValid = await bcrypt.compare(
                              password, user.password);

    if(isPasswordValid){
      res.send("login Successfully!!");
    } else {
      throw new Error("Password is not correct");
    }
  }
  catch(err) {
    res.status(400).send("Error:" + err.message);
  }
});
```

Your error message should be → "Invalid Credentials"

You should not give attacker any extra information.

Episode-10 | Authentication, JWT & Cookies

01/10/2024

To read the cookie, we need one more npm library that is cookie-parser.

npm i cookie-parser.

JWT token structure:
header.payload.signature

npm i jsonwebtoken

jwt.sign()
jwt.verify()

```
//Create a JWT Token
const token= await jwt.sign({_id:_id}, "Dev@123");

// Add the token to cookie and send the response
// back to the user
 res.cookie("token", token);
```

```
app.get("/profiles", async(req,res)=>{
      const cookies = req.cookies;
      const {token} = cookies;
      const decodedMessage = await jwt.verify(token,"Dev@123");

      const {_id} = decodedMessage;
       if(!_id) throw error;
    const user = await User.findById(_id);
    if(!user){
          throw new Error("User does not exist");
    }
```

Creating the auth middleware.

```
const userAuth = async (req, res, next) => {
    // Read the token from the req cookies
    // validate the token
    // find the user
try {
    const {token} = req.cookies;
    if(!token) throw new Error("Token invalid");
const decodedObj = await jwt.verify(token, "Dev@123");

    const {id} = decodedObj;
    const user = await User.findById(_id);
    const user = await User.findById(_id);
    if(!user) {
        throw new Error("User not found");
    } req.user = user; // just attach the user to req.
    next();
}
catch(err) {
    res.status(400).send("Error" + err.message);
}
};
```

How you can expire your JWT token?

```
await jwt.sign({_id: user._id}, "Dev@123",
    { expiresIn: "1d",
}),
```

1d → one day

1h = 1 hr

You can even expire your cookies

```
res.cookie("token", token, {
    expires: new Date(Date.now() + 8 * 3600000),
});
```

8 hr

Mongoose Schema Methods

You can offload the method of signing JWT token to schema methods.

* Make sure you are not using arrow func. Because inside arrow function 'this' doesn't work.

```
const token = await user.getJWT();
```

models > user.js

```
userSchema.methods.getJWT = async function() {
    const user = this;
    const token = await jwt.sign({_id: user._id},
        "Dev@gurm", {expiresIn: "7d"});

    return token;
}

module.exports = mongoose.model("User", userSchema)
```

You can also offload the bcrypt method

```
userSchema.methods.validatePassword =
    async function(passwordInputByUser) {
    const user = this;
    const passwordHash = user.password;
```

```
const isPasswordValid = await bcrypt.compare(
    passwordInputByUser,
    passwordHash
);
return isPasswordValid;
}
```

In app.js

```
const isPasswordValid =
    await user.validPassword(password);
```

Episode-11 | Diving into the APIs and express Routers

01/10/2024

API contract

DevTinder APIs

authRouter
- POST /signup
- POST /login
- POST /logout

profileRouter
- GET /profile/view
- PATCH /profile/edit
- PATCH /profile/password

ConnectionRequestRouter
- POST /request/send/interested/:userId
- POST /request/send/ignored/:userId
- POST /request/review/accepted/:requestId
- POST /request/review/rejected/:requestId

~~userRouter~~
- GET user/connections
- GET user/request-s/received
- GET user/feed - Gets you the profiles of other users on platform

Status: ignore, interested, accepted, rejected

~~##~~ we will use expressRouter to handle these many APIs.

Just do the logical separation

Episode-12 | Logical DB query & Compound Index 03/10/2024

We will do indexing to optimize our database.

```
// Compound Index
userSchema.index({ firstName: 1, lastName: 1});
```

Episode-13 | ref, Populate & Thought Process of writing APIs 04/10/2024

Thought Process POST vs GET

You are the guardian of your database.

We will create a relation between our user schema and connectionRequest Schema.

```
const connectionRequestSchema = new mongoose.Schema(
  {
    fromUserId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: true,
    },
```

Main → acco Auth

```
});
```

routes/user.js

```
userRouter.get("/user/requests/received", userAuth,
async(req, res) => {
  try {
    const loggedInUser = req.user;
    const connectionRequests = await ConnectionRequest
      find({
        toUserId: loggedInUser._id,
        status: "interested",
      }).populate("fromUserId", ["firstName", "lastName"]
```

"firstName LastName" or

⇒ Make sure you are not overfetching the data.