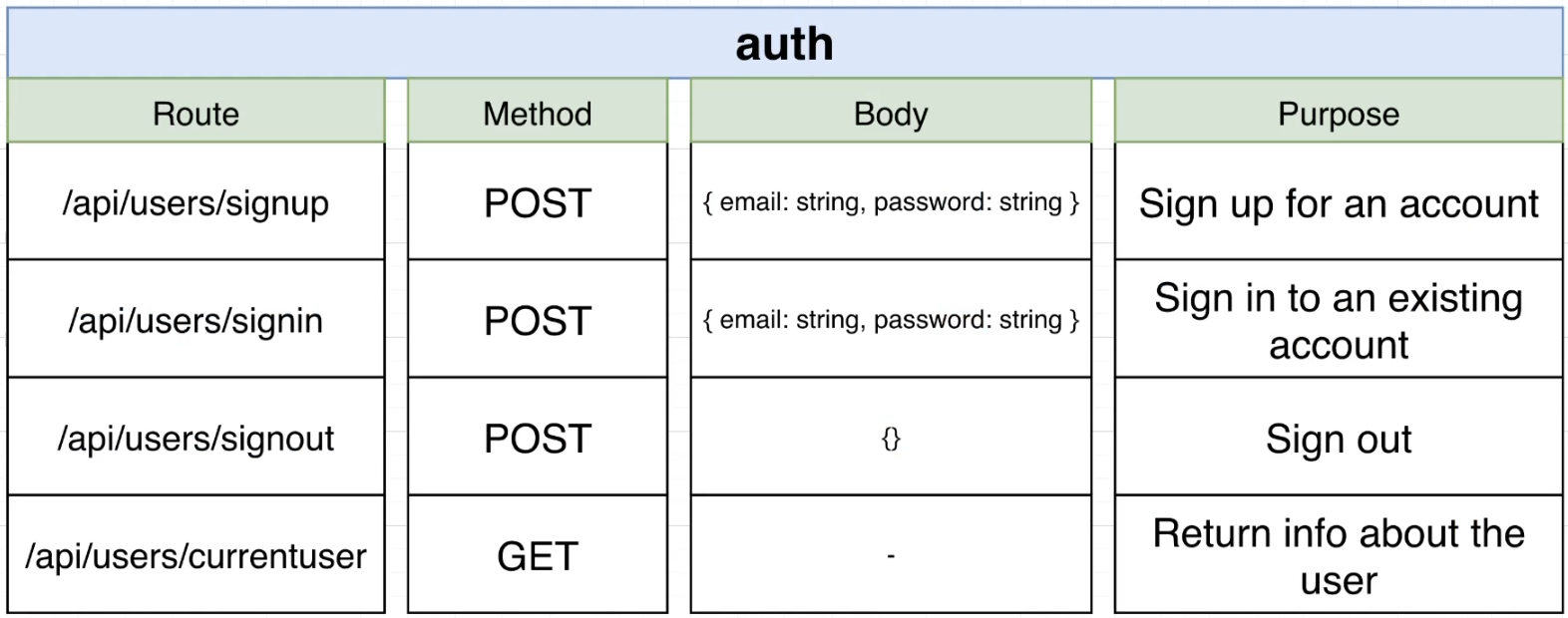
**AUTH SERVICE**

**PHASE 1 and 2**

**Auth Service**

Auth Service is an service which is reponsible for authentication process like signIn, signUp, singOut. (We will not use JWT for this project)

**Quick Diagram for DataFlow Overview**



**Initialization**

Let’s Start with auth service

--> mkdir **ticketing**

-->cd **ticketing**

--> mkdir **auth**

--> cd **auth**

--> npm **init -y**

--> npm install **express @types/express**  **typescript**  **ts-node-dev**

Now Create typescript(tsc) config file

--> tsc **--init**

**Steps to follow for proper coverage of authentication Service**

**Phase 1 (Routes, Validation, ErrorHandling)**

1. Create Routes
2. For Signin Authentication use Library (**express-validator**)
3. Now Create **Error Handler** (**middleware**) and an Method to create **custom Errors**
4. Errors must follow the same structure all along the project for this project
5. -> [**{** message:**string**, feild?:**string }**]
6. use Library (**express-async-errors**) for calling async function from index.js

**Phase 2** (**Routes Defination**, **Authentication**, **Authorization**, **Isolated Env Set**)

1. Define all Routes
2. Define authentication and authorization were required (jwt)
3. Define Secrete key in isolated pod environment and config deployment

**Phase 3** (**Testing**)

1. Installingdependencies **jest**, **supertest**, **ts-jest**, **mongodb-memory-server** all in **development env**
2. Installing some type defined dependencies **@types/jest , @types/supertest.**
3. Setting up jest config in package.json
4. Setting up test script in package.json
5. Setting up **setup.ts** file in test folder and write some code for **starting**, **cleaning** and **ending** the service for jest test at every test loop.
6. Creating **\_\_test\_\_** folder under same directory, where testing is required and create testing files with **test\_name.test.ts**
7. **npm run test**

**Phase 1**

**Libraries Used**

1. **express**
2. **@types/express** **(Type Defination File)**

These Libraries help to create an server which initates with app.listen, and will help us to provide proper defination to routes.

1. **express-validator**
2. **@types/express-validator (Type Defination File)**

These Libraries help for easy validation of feilds such as email, password etc.

It uses body(‘feild\_name’) method for detecting the feild to be validated.

It uses validationResult method which will fetch out the detected error information from the request maniputated by body method.

1. **typescript**

Typescript Library will help to define types of each methods and will help to validate the arguments and many other things, so that integrity can be maintained.

Use -> **tsc** config

1. **ts-node** or **nodemon**

ts-node will help to convert typescript to javascript and start the index.ts file other wise two command must be used

ts-node = tsc + node

1. **express-async-errors**

Express-async-errors will help to start server listening in async manner just

**import** ‘express-async-errors’

1. **mongoose**

mongoose Library is used to connect and use mongodb via url request.

Remember thier is NO Type Defination file for mongoose for typescript compatibility

so we need to manualy define methods while using mongoose so that typescript can understand

1. **crypto**

crypto is an libarary which will be used for hashing passwords we are going to create salt via ‘randomByte’ and hash with ‘scrypt’ (callback function)

1. **util**

util is an library which will be used for converting callback scrypt to promise using ‘promisify’

**Detailed Code**

1. **Define server**

const express = require(‘express’);

const app = express();

app.listen(3000,()=>{ console.log(“Server Runnning on 3000”) })

1. **Define app parsers**

app.use(express.json());

1. **Create routes**

**( You can follow MVC folder Tree (Model,View,Controllers) )**

api/user/signin

api/user/signup

api/user/signout

api/user/currentuser

1. Create Error Handler **Middleware**
2. Create and middleware to handle all unknown url request (‘**\***’)
3. Create an **pod** for mongodb using **mongo-depl.yaml** file
4. Configure mongodb for using typescript manually
5. For Password hashing we will use **scrypt,randomBytes from ‘crypto’**

and **promisify from ‘util’** (to convert scrypt to promise) because scrypt is an callback fnx

**Phase 1 Code**

**index.ts**

import express from "express";

import "express-async-errors";

const app = express();

const userRouter = require("./routes/user/user\_Route");

app.use(express.json());

app.use("/api/users", userRouter);

const start = async () => {

app.listen(3000, () => {

console.log("Listening to http://localhost:3000");

});

};

start();

This is an simple implementation of starting server with required router (userRouter)

Now we are going to implement Error Handling which will handle throw error (Custom Error)

**CustomError.ts**

export abstract class CustomError extends Error {

abstract statusCode: number;

constructor(errorMessage: string) {

super(errorMessage);

Object.setPrototypeOf(this, CustomError.prototype);

}

abstract serializeErrors(): { message: string; feild?: string }[];

}

**Remember -** If you are extending any inbuild class in your custom class always set the prototype of the new object (**which will be created in future when we call contructor**) as its parent Class prototype so that chaining continous.

It can done by Object.setPrototypeOf( **Class\_Prototype\_need\_tochange**, **of\_whose\_prototype\_we\_want**)

**(Middleware) ErrorHandler.ts**

import { CustomError } from "../errors/custom-error";

import { Request, Response, NextFunction } from "express";

export const ErrorHandler = (

err: Error,

req: Request,

res: Response,

next: NextFunction

) => {

if (err instanceof CustomError) {

res.status(err.statusCode);

return res.send({ errors: err.serializeErrors() });

}

res.status(400);

return res.send([{ message: "Something Went Wrong" }]);

};

Big Step (Create **mongodb POD** container)

1. Create and Deployement with (**mongo** **DockerHub Image**) and its Service with Default Port **27017**

apiVersion: apps/v1

kind: Deployement

metadata:

name: auth-mongo-depl

spec:

replicas: 1

selector:

matchLabels:

app: auth-mongo

template:

metadata:

labels:

app: auth-mongo

spec:

containers:

- name: auth-mongo

image: mongo

---

apiVersion: v1

kind: Service

metadata:

name: auth-mongo-srv

spec:

type: ClusterIP

selector:

app: auth-mongo

ports:

- name: db

port: 27017

targetPort: 27017

protocol: TCP

1. Scaffold will detect the changes and will deploy automatically
2. Now to access mongodb Server goto ( **mongodb:// pod\_service\_name:27017 / databaseName** )
3. Eg -> **mongodb://auth-mongo-srv:27017/auth**

Now You can create Different Custom Errors according to your needs

1. bad-request**.ts**
2. database-connection-error**.ts**
3. not-found-error**.ts**
4. request-validation-error**.ts**

**Now Create Method for password Hashing Password.ts**

import { scrypt, randomBytes } from "crypto";

import { promisify } from "util";

const scryptAsync = promisify(scrypt);

export class Password {

static async toHash(password: string) {

const salt = randomBytes(8).toString("hex");

const buf = (await scryptAsync(password, salt, 120)) as Buffer;

return `${buf.toString("hex")}.${salt}`;

}

static async compare(storedPassword: string, suppliedPassword: string) {

const [hashedPassword, salt] = storedPassword.split(".");

const buf = (await scryptAsync(suppliedPassword, salt, 120)) as Buffer;

return buf.toString("hex") === hashedPassword;

}

}

**There are going to be Two Methods which will be used for hashing and comparing**

1. **toHash** (**password: string**)**:** **string**

this method is going to use salt Generate by randomBytes (length).toString(‘hex’)

this salt futher will be used to generate **buffer** **by scrypt** and then converted to string(‘hex’) and **added with salt with “ . ”**

1. **compare** (**storedPassword: string, suppliedPassword: string**)**:** **string**

this method will **seperate salt from hashedPassword** and then compare hashedPassword with supplied Password using the salt **split(‘.’)**

it is going to **return** **boolean** (Weather storedPassword equal to supplied Password )

**Now you can also need to create model for mongodb**

In this we are going to create UserModel with {email,password} with Class String

import mongoose from "mongoose";

import { Password } from "../services/Password";

// An interface that describes the properties

// that are requried to create a new User

interface UserAttrs {

email: string;

password: string;

}

// An interface that describes the properties

// that a User Model has

interface UserModel extends mongoose.Model<UserDoc> {

build(attrs: UserAttrs): UserDoc;

}

// An interface that describes the properties

// that a User Document has

interface UserDoc extends mongoose.Document {

email: string;

password: string;

}

const userSchema = new mongoose.Schema(

{

email: {

type: String,

required: true,

},

password: {

type: String,

required: true,

},

},

{

toJSON: {

transform(doc, ret) {

ret.id = ret.\_id;

delete ret.\_id;

delete ret.password;

delete ret.\_\_v;

},

},

}

);

userSchema.pre("save", async function (done) {

if (this.isModified("password")) {

const hashed = await Password.toHash(this.get("password"));

this.set("password", hashed);

}

done();

});

userSchema.statics.build = (attrs: UserAttrs) => {

return new User(attrs);

};

const User = mongoose.model<UserDoc, UserModel>("User", userSchema);

export { User };

if set transform, mongoose will call this function to allow you to transform the returned object,

**doc** ,

**ret** -> this will represent actual data

so try to manupuate retured data in ret

To connect to mongodb in **index.ts (Phase 1 Final index.ts)**

try to connect to mongodb://mongo\_cluster\_srv:27017/databaseName

**Eg include all Error Handling (catch) , all invalid route handler, all routes , connection to mongodb**

import express from "express";

import mongoose from "mongoose";

import cookieSession from "cookie-session";

import "express-async-errors";

const app = express();

const userRouter = require("./routes/user/user\_Route");

import { ErrorHandler } from "./middlewares/error-handler";

import { NotFoundError } from "./errors/not-found-error";

app.use(express.json());

// cookieSession Work if trust proxy set to "true"

app.set("trust proxy", true);

app.use(

cookieSession({

secure: true,

signed: false,

})

);

app.use("/api/users", userRouter);

app.use("\*", () => {

throw new NotFoundError();

});

app.use(ErrorHandler);

const databaseName = "auth";

const defaultPort = 27017;

const clusterIPService = "auth-mongo-srv";

const start = async () => {

if (!process.env.JWT\_KEY) {

throw new Error("JWT\_KEY not defined");

}

try {

await mongoose

.connect(`mongodb://${clusterIPService}:${defaultPort}/${databaseName}`)

.then(() => {

console.log("Connected to Mongodb Container Pod Via ClusterIP");

app.listen(3000, () => {

console.log("Listening to http://localhost:3000");

});

});

} catch (error) {

console.error("Database connection Error \n" + error);

}

};

start();

**Phase 2**

**Libraries Used**

1. cookie-session
2. @types/cookie-session

cookie-session is library which willl help to set cookie in Request (req.session) with key value pair, and put it into “Set-Cookie” header of Response if in case need to fetch from response.

it will only work if **app.set( “trust proxy” , true )** on index.ts

1. jasonwebtoken
2. @types/jsonwebtoken

jsonwebtoken is an library which is used to **sign** and encrypt json data with an **secret key** and can be used to **authenticate** or **authorize** user by **verify** with that same **secret key**.

**Middlewares Created**

1. current-user.ts
2. unauthentic-request-handler.ts
3. validate-request-handler.ts

**Secret Object**

Let focus to Create ENV secret (OBJECT) for isolated Pod

Always try to create secret using **imperitive command** rather than declarative command Create secret because we don’t want any one to see secret writen in deployment yaml file.

Imperitive Command

**kubectl** create secret **generic** **jwt-secret** **--from-literal**=JWT\_KEY=SECRET\_KEY

**Defination**

**create secret ->** it is an command to create an secret

**generic ->** it is type of secret we want to createthere can be docker secret or many more but here we will create generic type

**jwt-secret ->** it is the **name** for the **secret (Object)** like the name of pod **(Object)**

**-from--literal ->** it is way we will define secret with **key value pair**

**JWT\_KEY ->** it is an **KEY**

**SECRET\_KEY ->** it is an **Secret value**

**Now lets modify Deployement yaml file of auth-depl (attach env with isolated image pod)**

apiVersion: apps/v1

kind: Deployment

metadata:

name: ticketing-auth-depl

spec:

replicas: 1

selector:

matchLabels:

app: auth

template:

metadata:

labels:

app: auth

spec:

containers:

- name: auth

image: **davimehra/ticketing\_auth**

env:

- name: JWT\_KEY

valueFrom:

secretKeyRef:

name: jwt-secret

key: JWT\_KEY

---

apiVersion: v1

kind: Service

metadata:

name: ticketing-auth-srv

spec:

selector:

app: auth

type: ClusterIP

ports:

- name: auth

protocol: TCP

port: 3000

targetPort: 3000

This name will be used in project for fetching value

**process.env.JWT\_KEY**

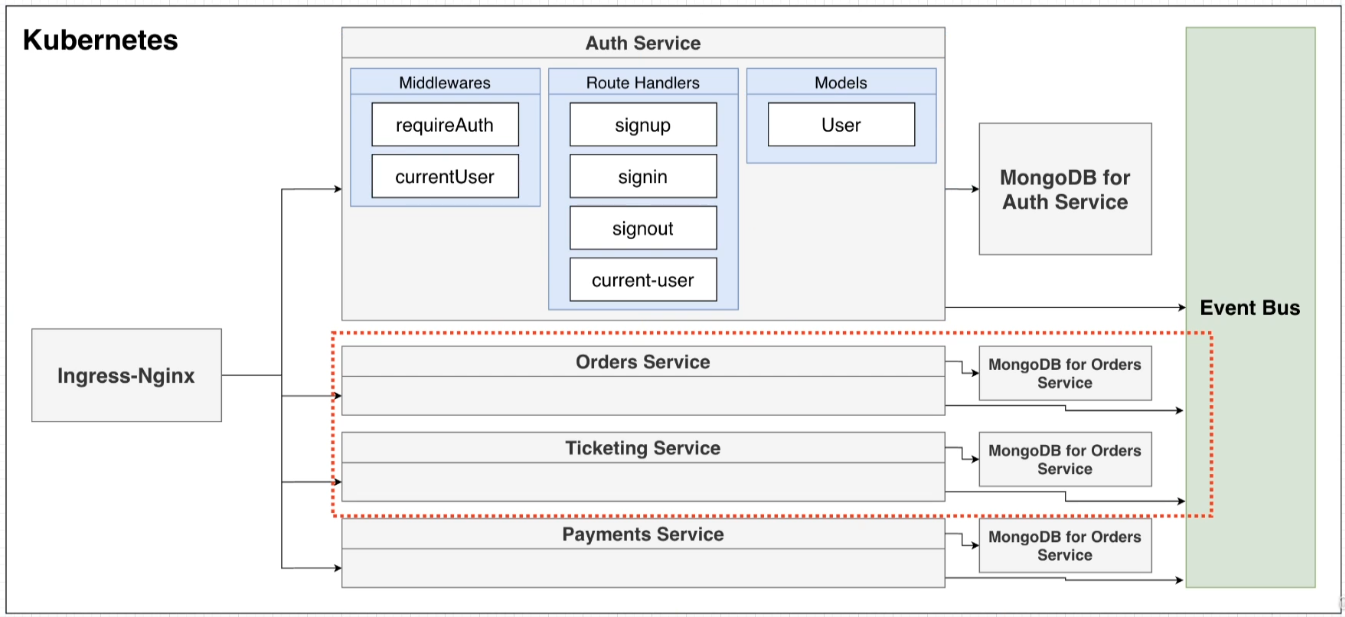
Can give any name here but remember name for future use in pod

Must be same name of secret OBJECT declared while creating secret

Must be the key name while creating key value pair while creating secret object

Now **imagePod** **davimehra/ticketing\_auth** can access environment variable **JWT\_KEY** from any where inside the pod

Middleware usage Diagram



Here requireAuth and currentUser are middlewares

1. **currentUser middleware**

It is used to verify the **jwt accessToken** present in req.cookie and will decode the information present in it with secret token in pod environment. The decoded information is put into **req.currentUser** and will be used in **requireAuth** for authentication.

it will proceded to next middleware with next(). Even if it is unsucessfully to fetch information from accessToken and put into req.currentUser we will move to next middleware of **requireAuth**

which will handle remaining process.

1. **requireAuth middleware**

It is used to verify the presence of currentUser in request .

if verified sucessfully then will next() to preceding middleware other wise will throw new **UnauthenticRequestError()**

Code for currentUser.ts (**middleware**)

import jwt from "jsonwebtoken";

import { Response, Request, NextFunction } from "express";

interface userPayload {

email: string;

id: string;

}

declare global {

namespace Express {

interface Request {

currentUser?: userPayload;

}

}

}

const currentUser = (req: Request, res: Response, next: NextFunction) => {

if (!req.session?.jwt) {

// will be handled in Unauthentic Request handler

return next();

}

const accessToken = req.session.jwt;

try {

const payload = jwt.verify(

accessToken,

process.env.JWT\_KEY!

) as userPayload;

req.currentUser = payload;

} catch (error) {}

next();

};

export { currentUser };

Code for requireAuth.ts (**middleware**)

import { Request, Response, NextFunction } from "express";

import { UnauthenticRequestError } from "../errors/unauthentic-request-error";

const requireAuth = (req: Request, res: Response, next: NextFunction) => {

if (!req.currentUser) {

throw new UnauthenticRequestError();

}

next();

};

export { requireAuth };

**Signup Controller Code**

Requirements -> jwt, cookie-session, Errors Handling, returning cookie and json data as response.

**Remember** -> here we will pass **two** middleware either pass it in Controller here or pass in Were you define route.

**middlewares** -> array of body for validateInputs, validateRequest

Code.

import { Request, Response, NextFunction } from "express";

import { BadRequestError } from "../../../errors/bad-request";

import { User } from "../../../models/UserModel";

import jwt from "jsonwebtoken";

const signup\_Controller = async (req: Request, res: Response) => {

const { email, password } = req.body;

const userExists = await User.findOne({ email });

if (userExists) {

throw new BadRequestError("Email Already Exists");

}

const user = User.build({ email, password });

await user.save();

try {

const accessToken = jwt.sign(

{ email: user.email, id: user.id },

process.env.JWT\_KEY!

);

req.session = { jwt: accessToken };

} catch (error) {

console.error(error);

}

res.status(201);

res.json(user);

};

module.exports = { signup\_Controller };

**SignIn Controller Code**

**Remember** -> here we will pass **two** middleware either pass it in Controller here or pass in Were you define route.

**middlewares** -> array of body for validateInputs, validateRequest

Code.

import { User } from "../../../models/UserModel";

import { Request, Response } from "express";

import { Password } from "../../../services/Password";

import { BadRequestError } from "../../../errors/bad-request";

import jwt from "jsonwebtoken";

const signin\_Controller = async (req: Request, res: Response) => {

const { email, password } = req.body;

const existingUser = await User.findOne({ email }).exec();

if (!existingUser) {

throw new BadRequestError("Invalid Credentials");

}

const passMatch = await Password.compare(existingUser.password, password);

if (!passMatch) {

throw new BadRequestError("Invalid Credentials");

}

try {

const accessToken = jwt.sign(

{ email: existingUser.email, id: existingUser.id },

process.env.JWT\_KEY!

);

req.session = { jwt: accessToken };

} catch (error) {

console.error(error);

}

res.status(200);

res.json(existingUser);

};

module.exports = { signin\_Controller };

**SignOut Controller Code**

Here to make cookie session null in request (req) we need to assign null to make cookie empty in req.session

import { Request, Response } from "express";

const signout\_Controller = async (req: Request, res: Response) => {

req.session = null;

res.status(200);

res.json({});

};

export { signout\_Controller };

**CurrentUser Controller Code**

**Remember** -> here we will pass **two** middleware either pass it in Controller here or pass in Were you define route.

**middlewares** -> currentUser , authRequire

import jwt from "jsonwebtoken";

import { Response, Request, NextFunction } from "express";

interface userPayload {

email: string;

id: string;

}

declare global {

namespace Express {

interface Request {

currentUser?: userPayload;

}

}

}

const currentUser = (req: Request, res: Response, next: NextFunction) => {

if (!req.session?.jwt) {

// will be handled in Unauthentic Request handler

return next();

}

const accessToken = req.session.jwt;

try {

const payload = jwt.verify(

accessToken,

process.env.JWT\_KEY!

) as userPayload;

req.currentUser = payload;

} catch (error) {}

next();

};

export { currentUser };

Finally **Routes.ts**

**Including all middleware implementation,Controllers implementataions**

import { Router } from "express";

import { body } from "express-validator";

import { currentUser } from "../../middlewares/current-user";

import { requireAuth } from "../../middlewares/requireAuth";

import { validateRequest } from "../../middlewares/validate-request-handler";

const { signup\_Controller } = require("./controllers/signup\_Controller");

const { signin\_Controller } = require("./controllers/signin\_Controller");

const { signout\_Controller } = require("./controllers/signout\_Controller");

const {

currentuser\_Controller,

} = require("./controllers/currentuser\_Controller");

const router = Router();

router.post(

"/signup",

[

body("email").isEmail().withMessage("Invalid Email Address"),

body("password")

.isLength({ min: 8, max: 16 })

.withMessage("Password Length between 8-16"),

],

validateRequest,

signup\_Controller

);

router.post(

"/signin",

[

body("email").isEmail().withMessage("Invalid Email Address"),

body("password").trim().notEmpty().withMessage("Password Must not Empty"),

],

validateRequest,

signin\_Controller

);

router.get("/signout", signout\_Controller);

router.get("/currentuser", currentUser, requireAuth, currentuser\_Controller);

module.exports = router;

**Phase 1 and 2 Index.ts (FINAL)**

import express from "express";

import mongoose from "mongoose";

import cookieSession from "cookie-session";

import "express-async-errors";

const app = express();

const userRouter = require("./routes/user/user\_Route");

import { ErrorHandler } from "./middlewares/error-handler";

import { NotFoundError } from "./errors/not-found-error";

app.use(express.json());

// cookieSession Work if trust proxy set to "true"

app.set("trust proxy", true);

app.use(

cookieSession({

secure: true,

signed: false,

})

);

app.use("/api/users", userRouter);

app.use("\*", () => {

throw new NotFoundError();

});

app.use(ErrorHandler);

const databaseName = "auth";

const defaultPort = 27017;

const clusterIPService = "auth-mongo-srv";

const start = async () => {

if (!process.env.JWT\_KEY) {

throw new Error("JWT\_KEY not defined");

}

try {

await mongoose

.connect(`mongodb://${clusterIPService}:${defaultPort}/${databaseName}`)

.then(() => {

console.log("Connected to Mongodb Container Pod Via ClusterIP");

app.listen(3000, () => {

console.log("Listening to http://localhost:3000");

});

});

} catch (error) {

console.error("Database connection Error \n" + error);

}

};

start();

**Phase 3**

**Testing Service**

For testing services we need special libraries like (**devDependencies**)

1. **jest**

jest is an library which will help to start testing by searching for well defined test files located in **\_\_test\_\_** folder with extension of **.test.ts** or **.test.js**

1. **supertest**

supertest help to provide portable request sending feature like postman

1. **ts-jest**

ts-jest will help jest to create environment suitable for typescript.

1. **mongodb-memory-server**

mongodb-memory-server is an library which will create an instance of mongodb in temporary memory so that test can performed without creating or utilizing any local space.

**Important** (**libssl1.1**) is required by **mongodb-memory-server** which is available in ubuntu 20.04 or less

1. **@types/jest** , **@types/supertest**

These are type defination library for typescript support of these libraries

**STEP 1**

Create **setup.ts** file in **test** folder

Create **beforeAll**, **afterEach**, **afterAll** functions to setup test environment.

import { MongoMemoryServer } from "mongodb-memory-server";

import mongoose from "mongoose";

import Request from "supertest";

import { app } from "../app";

let mongo: any;

beforeAll( async () => {

process.env.JWT\_KEY = "asdfsdv";

mongo = await MongoMemoryServer.create();

const mongoUri = mongo.getUri();

await mongoose.connect(mongoUri);

}

);

beforeEach(async () => {

const collections = await mongoose.connection.db.collections();

for (let collection of collections) {

await collection.deleteMany({});

}

});

afterAll(async () => {

mongo.stop();

await mongoose.connection.close();

});

**beforAll**, **beforeEach**, **afterAll** and other these are the **lifecycle function** of jest

throw these functions jest at its checkpoint of its lifecycle will find these specific functions.

**beforeAll** -> it will work once before All test in future , **remember not before every test.**

**beforeEach** -> it will work before each test.

**afterEach** -> it will work after each test.

**afterAll** -> it will work after All the test will finish . At the end **remember not after every test.**

Here

**process.env.EnvironmentVar** can be initiated for the tests but it will not be available outside the test environment. Even You can **declare global function** and can use in every test.

**Step 2**

Write some config for jest to unnderstand typescript and also to run tests in **package.json**

**scripts**:{

**“test”** **:** **”**jest --watchAll --no-cache**”**

},

“**jest**”:{

"**preset**": "ts-jest",

"**testEnvironment**": "node",

"**setupFilesAfterEnv**": **[ "./src/test/setup.ts"]**

}

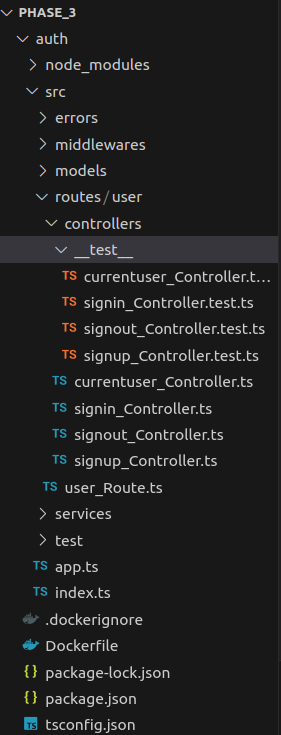
**Step 3**

**Create** **Test Suits** (testFiles).

**Create** **Test Cases** for each test suit.

**Remember** Test files must be placed at same place in folder \_\_test\_\_ , where the files are located and for whom we are testing for.

This folder **\_\_test\_\_** will be seen by jest as test suite files and will **run every test cases** in that suit



Here controller containes all Controllers files

Here **\_\_test\_\_** is an folder for jest to detect the test files

These are the files of all test.ts which will be treated as TEST SUITS.

and will be run every case for jest

**STEP 4**

Test File Example for **currentuser.test.ts**

Here **supertest** is an library which will provide **Request Functionality** for performing api request within environment

import Request from "supertest";

import { app } from "../../../../app";

it("return 200 on successfull currentuser fetch", async () => {

const cookie = await global.signin();

const response = await Request(app)

.get("/api/users/currentuser")

.set("Cookie", cookie)

.expect(200);

expect(response.body.currentUser.email).toBeDefined();

});

it("return 403 on successfull currentuser fetch", async () => {

await Request(app).get("/api/users/currentuser").expect(403);

});

**it (function)** -> this function will provide testing features like status message, a callback function and the expect method for defining final expectation.

**STEP 5**

Remember we are going to run these tests in developement environment.

so always while creating an docker image use --> **npm install --only=prod**

this command while creating docker image will only install production libraries and will not include **dev (Development)** library this will save some memory.

**Now Run test with cmd**

--> **npm run test**