Notes

Continue from page 38

**Validate Job**

validationMiddleware.js

* Import Job
* import mongoose from "mongoose";
* import Job from "../models/JobModel.js"
* We update the validateIdParama function
* export const validateIdParam = withValidationErrors([
* param("id").custom(async (value) => {
* const isValidId = mongoose.Types.ObjectId.isValid(value);
* if (!isValidId) throw new BadRequestError("invalid MongoDB id");
* const job = await Job.findById(value);
* if (!job) throw new NotFoundError(`no job with id : ${value}`);
* }),
* ]);
* Import not found error
* import { BadRequestError, NotFoundError } from "../errors/customErrors.js";

jobControllers.js

* remove the import
* import { NotFoundError } from "../errors/customErrors.js";
* update the following controllers by removing:
* if (!isValidId) throw new BadRequestError("invalid MongoDB id");
* getJob
* export const getJob = async (req, res) => {
* const { id } = req.params;
* const job = await Job.findById(id);
* res.status(StatusCodes.OK).json({ job });
* };
* updateJob
* export const getJob = async (req, res) => {
* const { id } = req.params;
* const job = await Job.findById(id);
* res.status(StatusCodes.OK).json({ job });
* };
* deleteJob
* export const updateJob = async (req, res) => {
* const { id } = req.params;
* const updatedJob = await Job.findByIdAndUpdate(id, req.body, {
* new: true,
* });
* res.status(StatusCodes.OK).json({ job: updatedJob });
* };
* Thunder Client
* Get All Jobs
* Send
* Copy Id of one of the outputs
* Get Single Job
* Paste the id
* {{URL}}/jobs/id(paste)
* Change the last digit of the id
* We get error 400
* We want error 404
* validationMiddleware.js
* update the withValidationErrors function
* const withValidationErrors = (validateValues) => {
* return [
* validateValues,
* (req, res, next) => {
* const errors = validationResult(req);
* if (!errors.isEmpty()) {
* const errorMessages = errors.array().map((error) => error.msg);
* if (errorMessages[0].startsWith("no job")) {
* throw new NotFoundError(errorMessages);
* }
* throw new BadRequestError(errorMessages);
* }
* next();
* },
* ];
* };
* Thunder Client
* Get Single Job
* Change the last digit of the id
* Send
* Error now is 404
* Refactoring the controller
* jobControllers.js
* getJob
* export const getJob = async (req, res) => {
* const job = await Job.findById(req.params.id);
* res.status(StatusCodes.OK).json({ job });
* };
* updateJob
* export const updateJob = async (req, res) => {
* const updatedJob = await Job.findByIdAndUpdate(req.body.id, req.body, {
* new: true,
* });
* res.status(StatusCodes.OK).json({ job: updatedJob });
* };
* deleteJob
* export const deleteJob = async (req, res) => {
* const removedJob = await Job.findByIdAndDelete(req.params.id);
* res.status(StatusCodes.OK).json({ job: removedJob });
* };
* We can substitute the custom error with the javaScript error and it will still work

**User Model**

* Clean the DB (Database) Website To prevent bugs
* Clusters
* View Monitoring
* Collections
* Jobify
* Click waste bin next to jobs
* Models folder – create UserModel.js file
* UserModel.js
* import mongoose from "mongoose";
* const UserSchema = new mongoose.Schema({
* name: String,
* email: String,
* password: String,
* lastName: {
* type: String,
* default: "lastName",
* },
* location: {
* type: String,
* default: "my city",
* },
* role: {
* type: String,
* enum: ["user", "admin"],
* default: "user",
* },
* });
* export default mongoose.model("User", UserSchema);

**User Controller and Router**

* controllers folder – create authController.js file
* authController.js
* export const register = async (req, res) => {
* res.send("register");
* };
* export const login = async (req, res) => {
* res.send("login");
* };
* Routes folder – create authRouter.js file
* authRouter.js
* import { Router } from "express";
* const router = Router();
* import { login, register  } from "../controllers/authController.js";
* router.post("/register", register);
* router.post("/login", login);
* export default router;
* server.js
* got the routers imports (added one is the bottom one) (top one if for direction in the code)
* //routers
* import jobRouter from "./routes/jobRouter.js";
* import authRouter from "./routes/authRouter.js"
* add it to the middleware (bottom one)
* app.use("/api/v1/jobs", jobRouter);
* app.use("/api/v1/auth", authRouter);
* Thunder Client
* Jobify – create a new folder – Auth Route
* Auth Route
* New request – Register User
* POST {{URL}}/auth/register
* Send
* Response - register
* Duplicate Register User
* Rename to Login User
* POST {{URL}}/auth/login
* Send
* Response - login

**Create User**

* authController.js
* Import StatusCode and User
* import { StatusCodes } from "http-status-codes";
* import User from "../models/UserModel.js";
* update the register function
* export const register = async (req, res) => {
* const user = await User.create(req.body);
* res.status(StatusCodes.CREATED).json({ user });
* };
* Thunder Client
* Register User
* Body – JSON – JSON Content
* Paste the following
* {
* "name": "john",
* "email": "john@gmail.com",
* "password": "secret123",
* "lastName": "smith",
* "location": "my city"
* }
* Send
* New user will be registered

Validate Registered User

* Middleware folder
* validationMiddleware.js
* at the bottom add validationRegisterInput
* the guideline to create the validation will be based on the UserModel
* import the User (the bottom one)
* import Job from "../models/JobModel.js";
* import User from "../models/UserModel.js"
* validationRegisterInput
* export const validateRegisterInput = withValidationErrors([
* body("name").notEmpty().withMessage("name is required"),
* body("email")
* .notEmpty()
* .withMessage("email is required")
* .isEmail()
* .withMessage("invalid email format")
* .custom(async (email) => {
* const user = await User.findOne({ email });
* if (user) {
* throw new BadRequestError("email already exists");
* }
* }),
* body("password")
* .notEmpty()
* .withMessage("password is required")
* .isLength({ min: 8 })
* .withMessage("password must be at least 8 characters long"),
* body("location").notEmpty().withMessage("location is required"),
* body("lastName").notEmpty().withMessage("last name is required"),
* ]);
* Now work on the routes
* authRouter.js
* import the validateRegisterInput (the bottom one)
* import { login, register } from "../controllers/authController.js";
* import { validateRegisterInput } from "../middleware/validationMiddleware.js";
* insert the validation on the register router
* router.post("/register", validateRegisterInput, register);
* Thunder Client
* Register User
* JSON Content
* Delete some of the key:value pair to activate certain errors
* If the email already exist error is getting too much clear up the Data base on MongoDB

**Admin User**

* We can change the user to Admin from MongoDB
* But we will set it up on the code. If the user is the first uploaded data. That user will be the admin
* authController.js
* update the register function
* export const register = async (req, res) => {
* const isFirstAccount = (await User.countDocuments()) === 0;
* req.body.role = isFirstAccount ? "admin" : "user";
* const user = await User.create(req.body);
* res.status(StatusCodes.CREATED).json({ user });
* };
* Thunder Client
* Clear Data on MongoDB
* Register User
* Add first user
* Add second user and change “role”: “admin”
* Role will be user

**Hash Password**

* Preventing saving the password without hiding it in the database
* [bcryptjs](https://www.npmjs.com/package/bcryptjs)
* Install the package
* npm i bcryptjs@2.4.3
* authController.js
* import bcrypt package (the bottom one)
* import User from "../models/UserModel.js";
* import bcrypt from "bcryptjs";
* update the register function
* export const register = async (req, res) => {
* const isFirstAccount = (await User.countDocuments()) === 0;
* req.body.role = isFirstAccount ? "admin" : "user";
* const salt = await bcrypt.genSalt(10);
* const hashedPassword = await bcrypt.hash(req.body.password, salt);
* req.body.password = hashedPassword;
* const user = await User.create(req.body);
* res.status(StatusCodes.CREATED).json({ msg: "user created" });
* };
* Thunder Client
* Clear database on MongoDB
* Register User
* JSON Content = send
* Response
* {
* "msg": "user created"
* }
* MongoDB
* Check user data
* Password is hashed

**Hash Utils**

* Utils folder – create passwordUtils.js file
* passowordUtils.js
* authController.js
* delete the import
* import bcrypt from "bcryptjs";
* import the hassPassword
* import { hashPassword } from "../utils/passwordUtils.js";
* update the register function (removing other items and placing them on passwordUtils.js)
* export const register = async (req, res) => {
* const isFirstAccount = (await User.countDocuments()) === 0;
* req.body.role = isFirstAccount ? "admin" : "user";
* const hashedPassword = await hashPassword(req.body.password);
* req.body.password = hashedPassword;
* const user = await User.create(req.body);
* res.status(StatusCodes.CREATED).json({ msg: "user created" });
* };
* passwordUtils.js
* import bcrypt from "bcryptjs";
* export async function hashPassword(password) {
* const salt = await bcrypt.genSalt(10);
* const hashedPassword = await bcrypt.hash(password, salt);
* return hashedPassword;
* }
* Thunder Client
* Register User
* JSON Content – Send
* Check MongoDB is the password is hashed

Validate Login

* Thunder Client
* Login User
* JSON Content
* {
* "email": "john@gmail.com",
* "password": "secret123"
* }
* Send –
* Login
* validationMiddleware.js
* add function at the bottom
* export const validateLoginInput = withValidationErrors([
* body("email")
* .notEmpty()
* .withMessage("email is required")
* .isEmail()
* .withMessage("invalid email format"),
* body("password").notEmpty().withMessage("password is required"),
* ]);
* authRouter.js
* import validateInput
* import { validateLoginInput, validateRegisterInput } from "../middleware/validationMiddleware.js";
* update the login router
* router.post("/login", validateLoginInput, login);
* Thunder Client
* Login User
* JSON Content – leave out the password
* Send
* Error
* {
* "msg": "password is required"
* }

**Login Logic**

* authController.js
* import UnauthenticatedError (the bottom one)
* import { hashPassword } from "../utils/passwordUtils.js";
* import { UnauthenticatedError } from "../errors/customErrors.js";
* check if the user exist
* check if the password is correct
* export const login = async (req, res) => {
* // check if user exists
* // check if password is correct
* const user = await User.findOne({ email: req.body.email });
* if (!user) throw new UnauthenticatedError("invalid credentials");
* res.send("login route");
* };
* Utils folder
* passwordUtils.js
* at the bottom below hassPassword
* export const comparePassword = async (password, hashedPassword) => {
* const isMatch = await bcrypt.compare(password, hashedPassword);
* return isMatch;
* };
* authController
* import comparedPassword
* import { hashPassword, comparePassword } from "../utils/passwordUtils.js";
* update the login function
* export const login = async (req, res) => {
* const user = await User.findOne({ email: req.body.email });
* if (!user) throw new UnauthenticatedError("invalid credentials");
* const isPasswordCorrect = await comparePassword(
* req.body.password,
* user.password
* );
* if (!isPasswordCorrect) throw new UnauthenticatedError("invalid credentials");
* res.send("login");
* };
* Refactor the code
* Updated login function
* export const login = async (req, res) => {
* const user = await User.findOne({ email: req.body.email });
* const isValidUser =
* user && (await comparePassword(req.body.password, user.password));
* if (!isValidUser) throw new UnauthenticatedError("invalid credentials");
* res.send("login");
* };
* Thunder Client
* Login User
* JSON Content
* Remove one password digit
* Send
* Error
* {
* "msg": "invalid credentials"
* }

**JWT**

* JSON Web Token
* Is a compact and secure way of transmitting data between parties. It is often used to authenticate and authorize users in web applications and APIs. JWTs contain information about the user and additional metadata, and can be used to securely transmit this information
* [Useful Resource](https://jwt.io/introduction)
* Install the package
* npm i jsonwebtoken
* utils folder – create a file – tokenUtils.js
* tokenUtils.js
* import the library
* import jwt from "jsonwebtoken";
* the function (still on tokenUtils.js
* export const createJWT = (payload) => {
* const token = jwt.sign(payload, process.env.JWT\_SECRET, {
* expiresIn: process.env.JWT\_EXPIRES\_IN,
* });
* return token;
* };
* authController
* import createJWT (the bottom one)
* import { UnauthenticatedError } from "../errors/customErrors.js";
* import { createJWT } from "../utils/tokenUtils.js";
* update the login function
* export const login = async (req, res) => {
* const user = await User.findOne({ email: req.body.email });
* const isValidUser =
* user && (await comparePassword(req.body.password, user.password));
* if (!isValidUser) throw new UnauthenticatedError("invalid credentials");
* const token = createJWT({ userId: user.\_id, role: user.role });
* console.log(token);
* res.json({token})
* };
* JWT ENV
* .env (file) insert at the end (bottom)
* JWT\_SECRET=secret
* JWT\_EXPIRES\_IN=1d
* Debug the token
* [JWT](https://jwt.io/)
* Thunder Client
* Login User
* JSON Content – send
* Response
* {
* "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2NmZhYzY0N2UyZDNlYjc5MmI2MjdmZTAiLCJyb2xlIjoidXNlciIsImlhdCI6MTcyNzcxNzQ3NiwiZXhwIjoxNzI3ODAzODc2fQ.wmC7-6tUVm5FWdtht318H7wWdbHdCbHs\_nMRtqRISK8"
* }
* Copy the code, start after the “ and paste at [www.jwt.io](http://www.jwt.io)

**HTTP Only Cookie**

* Is a cookie that can’t be accessed by JavaScript running in the browser. It is designed to help prevent cross-site scripting (XSS) attacks, which can be used to steal cookies and other sensitive information
* Is designed to be inaccessible to JavaScript running in the browser. It is primarily used for authentication purposes and is more secure way of storing sensitive information like user tokens.
* Local storage, on the other hand, is a browser-based storage mechanism that is accessible to JavaScript, and is used to store application data like preferences or user-generated content
* authController.js
* create token inside the login function – update function
* export const login = async (req, res) => {
* const user = await User.findOne({ email: req.body.email });
* const isValidUser =
* user && (await comparePassword(req.body.password, user.password));
* if (!isValidUser) throw new UnauthenticatedError("invalid credentials");
* const token = createJWT({ userId: user.\_id, role: user.role });
* const oneDay = 1000 \* 60 \* 60 \* 24;
* res.cookie("token", token, {
* httpOnly: true,
* expires: new Date(Date.now() + oneDay),
* secure: process.env.NODE\_ENV === "production",
* });
* res.status(StatusCodes.CREATED).json({ msg: "user logged in" });
* };
* Thunder Client
* Login User
* JSON Content :
* {
* "email": "beth@gmail.com",
* "password": "secret123"
* }
* Response
* {
* "msg": "user logged in"
* }
* Clean up database
* MongoBD
* Delete both jobs and users

**Authenticate User Setup**

* Models folder
* JobModels.js
* Update the JobSchema by adding created by
* jobLocation: {
* type: String,
* default: "my city",
* },
* createdBy: {
* type: mongoose.Types.ObjectId,
* ref: "User",
* },
* },
* { timestamps: true }
* Middleware folder – create authMiddleware.js file
* authMiddleware.js
* create authicateUser function
* export const authenticateUser = async (req, res, next) => {
* console.log("auth middleware");
* next();
* };
* server.js
* import authenticaUser
* //Middleware
* import errorHandlerMiddleware from "./middleware/errorHandlerMiddleware.js";
* import { authenticateUser } from "./middleware/authMiddleware.js";
* add the authenticateUser in the job router (the top one)
* app.use("/api/v1/jobs", authenticateUser, jobRouter);
* app.use("/api/v1/auth", authRouter);
* Thunder Client
* Job Routes
* Get All Job – send
* Response
* {
* "jobs": []
* }
* Server terminal
* [nodemon] starting `node server.js`
* server running on PORT 5100....
* auth middleware
* GET /api/v1/jobs 200 85.177 ms - 11
* GET /api/v1/jobs 200 85.177 ms – 11

**Verify Cookie**

* [Cookie Parser](https://www.npmjs.com/package/cookie-parser)
* Install package
* npm i cookie-parser
* server.js
* import the library
* import mongoose from "mongoose";
* import cookieParser from "cookie-parser"
* add cookier parser middle (the top one)
* app.use(cookieParser());
* app.use(express.json());
* authMiddleware.js
* import file and update the function
* import { UnauthenticatedError } from "../errors/customErrors.js";
* export const authenticateUser = async (req, res, next) => {
* const { token } = req.cookies;
* if (!token) {
* throw new UnauthenticatedError("authentication invalid");
* }
* next();
* };

**Verify JWT**

* tokenUtils.js
* at verifyJWT function at the bottom
* export const verifyJWT = (token) => {
* const decoded = jwt.verify(token, process.env.JWT\_SECRET);
* return decoded;
* };
* authMiddleware.js
* import verifyJWT file (the bottom one)
* import { UnauthenticatedError } from "../customErrors.js";
* import { verifyJWT } from "../utils/tokenUtils.js";
* update the authenticaUser function
* export const authenticateUser = async (req, res, next) => {
* const { token } = req.cookies;
* if (!token) {
* throw new UnauthenticatedError("authentication invalid");
* }
* try {
* const { userId, role } = verifyJWT(token);
* req.user = { userId, role };
* next();
* } catch (error) {
* throw new UnauthenticatedError("authentication invalid");
* }
* };
* Add console of the getAllJobs controller
* jobController.js
* export const getAllJobs = async (req, res) => {
* console.log(req.user);
* const jobs = await Job.find();
* res.status(StatusCodes.OK).json({ jobs });
* };
* Thunder Client
* Get All Jobs – send
* Response
* {
* "jobs": []
* }
* Server terminal
* [nodemon] starting `node server.js`
* server running on PORT 5100....
* { userId: '66fac647e2d3eb792b627fe0', role: 'user' }
* GET /api/v1/jobs 200 340.166 ms – 11

Add User to Job Routes

* jobController
* update getAllJobs
* export const getAllJobs = async (req, res) => {
* console.log(req.user);
* const jobs = await Job.find({ createdBy: req.user.userId });
* res.status(StatusCodes.OK).json({ jobs });
* };
* Update createJob
* export const createJob = async (req, res) => {
* req.body.createdBy = req.user.userId;
* const job = await Job.create(req.body);
* res.status(StatusCodes.CREATED).json({ job });
* };
* Thunder Client
* Create Job
* Body – JSON – JSON Content
* {
* "company": "coding addict",
* "position": "backend-end",
* "jobStatus": "pending",
* "jobType": "full-time",
* "jobLocation": "florida"
* }
* Send
* Response
* "createdBy": "66fac647e2d3eb792b627fe0",
* Still on Thunder Client
* No go to Register User
* Body – JSON Content
* {
* "name": "sifiso",
* "email": "dhlamini@gmail.com",
* "password": "secret123",
* "lastName": "dhlamini",
* "location": "joburg",
* "role": "admin"
* }
* Login User
* {
* "email": "dhlamini@gmail.com",
* "password": "secret123"
* }
* Get All Jobs – send
* {
* "jobs": []
* }
* Because this user didn’t created the jobs

Validate Owner

* Currently a different user can edit, delete the job if they have the job id
* Checking if the user is the actual owner of the job
* validationMiddleware.js
* update the validateParam function
* export const validateIdParam = withValidationErrors([
* param("id").custom(async (value, { req }) => {
* const isValidMongoId = mongoose.Types.ObjectId.isValid(value);
* if (!isValidMongoId) throw new BadRequestError("invalid MongoDB id");
* const job = await Job.findById(value);
* if (!job) throw new NotFoundError(`no job with id ${value}`);
* const isAdmin = req.user.role === "admin";
* const isOwner = req.user.userId === job.createdBy.toString();
* if (!isAdmin && !isOwner)
* throw UnauthorizedError("not authorized to access this route");
* }),
* ]);
* Update also the withValidationErrors function
* const withValidationErrors = (validateValues) => {
* return [
* validateValues,
* (req, res, next) => {
* const errors = validationResult(req);
* if (!errors.isEmpty()) {
* const errorMessages = errors.array().map((error) => error.msg);
* if (errorMessages[0].startsWith("no job")) {
* throw new NotFoundError(errorMessages);
* }
* if (errorMessages[0].startsWith('not authorized')) {
* throw new UnauthorizedError('not authorized to access this route');
* }
* throw new BadRequestError(errorMessages);
* }
* next();
* },
* ];
* };

Logout Controller

* authController.js
* at the bottom of the page add the logout function
* export const logout = (req, res) => {
* res.cookie("token", "logout", {
* httpOnly: true,
* expires: new Date(Date.now()),
* });
* res.status(StatusCodes.OK).json({ msg: "user logged out!" });
* };
* Import the logout function (the bottom one)
* const router = Router();
* import { login, logout, register } from "../controllers/authController.js";
* insert the logout router (the bottom one)
* router.post("/login", validateLoginInput, login);
* router.get("/logout", logout);
* Thunder Client
* Collections
* Jobify
* Auth Routes
* Duplicate Login User
* Rename Logout User
* Change Method to GET
* {{URL}}/auth/logout
* Body – JSON Content – remove content
* Send
* Response
* {
* "msg": "user logged out!"
* }
* Get All Jobs
* Send
* Response
* {
* "msg": "authentication invalid"
* }

**User Routes**

* Controllers folder – create userController.js file
* userController.js
* import { StatusCodes } from "http-status-codes";
* import User from "../models/UserModel.js";
* import Job from "../models/JobModel.js";
* export const getCurrentUser = async (req, res) => {
* res.status(StatusCodes.OK).json({ msg: "get current user" });
* };
* export const getApplicationStats = async (req, res) => {
* res.status(StatusCodes.OK).json({ msg: "application stats" });
* };
* export const updateUser = async (req, res) => {
* res.status(StatusCodes.OK).json({ msg: "update user" });
* };
* authMiddleware.js
* remove the async
* export const authenticateUser =  (req, res, next) => {
* routes folder – create userRouter.js file
* userRouter.js
* import { Router } from "express";
* const router = Router();
* import {
* getCurrentUser,
* getApplicationStats,
* updateUser,
* } from "../controllers/userController.js";
* router.get("/current-user", getCurrentUser);
* router.get("/admin/app-stats", getApplicationStats);
* router.patch("/update-user", updateUser);
* export default router;
* server.js
* import the userRouter (the bottom one)
* //routers
* import jobRouter from "./routes/jobRouter.js";
* import authRouter from "./routes/authRouter.js";
* import userRouter from "./routes/userRouter.js"
* add the middleware (the middle one)
* app.use("/api/v1/jobs", authenticateUser, jobRouter);
* app.use("/api/v1/users", authenticateUser, userRouter);
* app.use("/api/v1/auth", authRouter);
* Thunder Client
* Login first
* Auth Routes
* Login User
* JSON Content:
* {
* "email": "dhlamini@gmail.com",
* "password": "secret123"
* }
* Send
* Response
* {
* "msg": "user logged in"
* }
* Create User Routes
* Jobify
* Create folder – User Routes
* User Routes
* New Request – Get Current User
* Get Current User
* Method – GET
* {{URL}}/users/current-user
* Send
* Response
* {
* "msg": "get current user"
* }
* Duplicate Current User
* Rename – Get Application Stats
* Get Application Stats
* Method – GET
* {{URL}}/users/admin/app-stats
* Response
* {
* "msg": "application stats"
* }
* Duplicate Get Application Stats
* Rename – Update User
* Method – Patch
* {{URL}}/users/update-user
* Send
* Response
* {
* "msg": "update user"
* }

**Get Current User**

* userController.js
* update getCurrentUser
* export const getCurrentUser = async (req, res) => {
* const user = await User.findOne({ \_id: req.user.userId });
* res.status(StatusCodes.OK).json({ user });
* };
* Thunder Client
* User Routes
* Get Current User
* Send
* Response
* {
* "user": {
* "\_id": "66faf6deacfe93dbd60d23a0",
* "name": "sifiso",
* "email": "dhlamini@gmail.com",
* "password": "$2a$10$Eph7A3p1Ed09LNqMNLzz8uhNaeflKasa/A8qG/.U3FOeLBTTx671S",
* "lastName": "dhlamini",
* "location": "joburg",
* "role": "user",
* "\_\_v": 0
* }
* }
* Auth Routes
* Login User
* JSON Content
* {
* "email": "beth@gmail.com",
* "password": "secret123"
* }
* Send
* {
* "msg": "user logged in"
* }
* User Routes
* Get Current User
* {
* "user": {
* "\_id": "66faf19e5d76114a33859dd8",
* "name": "mola",
* "email": "beth@gmail.com",
* "password": "$2a$10$/mj.hqUZSnP5DHQJcMp8pO3zKnj.gQr9WowUe0sulUr7eOXmrUKgW",
* "lastName": "good",
* "location": "cape town",
* "role": "admin",
* "\_\_v": 0
* }
* }
* Fix the issue of sending the password even though it is hashed (hide it on database)
* "password": "$2a$10$/mj.hqUZSnP5DHQJcMp8pO3zKnj.gQr9WowUe0sulUr7eOXmrUKgW",
* Models/UserModel.js
* UserModel.js
* Add the UserSchema at the bottom above export default
* UserSchema.methods.toJSON = function () {
* var obj = this.toObject();
* delete obj.password;
* return obj;
* };
* export default mongoose.model("User", UserSchema);
* userController.js
* update the getCurrentUser
* export const getCurrentUser = async (req, res) => {
* const user = await User.findOne({ \_id: req.user.userId });
* const userWithoutPassword = user.toJSON();
* res.status(StatusCodes.OK).json({ user: userWithoutPassword });
* };
* Thunder Client
* User Routes
* Get Current User
* Send
* Response
* {
* "user": {
* "\_id": "66faf19e5d76114a33859dd8",
* "name": "mola",
* "email": "beth@gmail.com",
* "lastName": "good",
* "location": "cape town",
* "role": "admin",
* "\_\_v": 0
* }
* }
* The password is now not appearing

**Update User**

* userController.js
* update the updateUser function
* export const updateUser = async (req, res) => {
* const updatedUser = await User.findByIdAndUpdate(req.user.userId, req.body);
* res.status(StatusCodes.OK).json({ msg: "user updated" });
* };
* Thunder Client
* User Routes
* Method – Patch
* JSON Content
* {
* "name": "beth",
* "email": "beth@gmail.com",
* "lastName": "smith",
* "location": "florida"
* }
* Response
* {
* "msg": "user updated"
* }
* User Routes
* Get Current User
* Method – Get
* {{URL}}/users/current-user
* Send
* Response
* {
* "user": {
* "\_id": "66fc3cc4989e7c535f1c114c",
* "name": "beth",
* "email": "beth@gmail.com",
* "lastName": "smith",
* "location": "florida",
* "role": "user",
* "\_\_v": 0
* }
* }
* Validation for the user update
* validationMiddleware.js
* at the bottom of the page
* export const validateUpdateUserInput = withValidationErrors([
* body("name").notEmpty().withMessage("name is required"),
* body("email")
* .notEmpty()
* .withMessage("email is required")
* .isEmail()
* .withMessage("invalid email format")
* .custom(async (email, { req }) => {
* const user = await User.findOne({ email });
* if (user && user.\_id.toString() !== req.user.userId) {
* throw new Error("email already exists");
* }
* }),
* body("location").notEmpty().withMessage("location is required"),
* body("lastName").notEmpty().withMessage("last name is required"),
* ]);
* userRouter.js
* insert the validation function of the update router (the bottom one)
* router.get("/admin/app-stats", getApplicationStats);
* router.patch("/update-user", validateUpdateUserInput, updateUser);
* import the validateUpdateUserIput function
* import { validateUpdateUserInput } from "../middleware/validationMiddleware.js";
* Thunder Client
* User Routes
* Update User
* JSON Content
* {
* "name": "beth",
* "email": "beth@gmail.com",
* "location": "florida"
* }
* {
* "msg": "last name is required"
* }
* Remove from the req.body the password
* userController.js
* update the updateUser function
* export const updateUser = async (req, res) => {
* const obj = { ...req.body };
* delete obj.password;
* const updatedUser = await User.findByIdAndUpdate(req.user.userId, obj);
* res.status(StatusCodes.OK).json({ msg: "user updated" });
* };
* Thunder Client
* Update User
* JSON Content
* {
* "name": "beth",
* "email": "beth@gmail.com",
* "lastName": "smith",
* "location": "florida",
* "password": "secrete"
* }
* Server terminal console log(obj)
* {
* name: 'beth',
* email: 'beth@gmail.com',
* lastName: 'smith',
* location: 'florida'
* }
* PATCH /api/v1/users/update-user 200 240.518 ms – 22
* From the output above the password doesn’t appear

Get Application Stats

* userController.js
* update the getApplicationStats
* export const getApplicationStats = async (req, res) => {
* const users = await User.countDocuments();
* const jobs = await Job.countDocuments();
* res.status(StatusCodes.OK).json({ users, jobs });
* };
* Thunder Client
* Auth Routes
* Get Application Stats
* Method – Get
* {{URL}}/users/admin/app-stats
* Send
* Response
* {
* "users": 2,
* "jobs": 2
* }
* Create a middleware to prevent other users from checking your stats
* authMiddleware.js
* create the function at the bottom of the page
* export const authorizePermissions = (...roles) => {
* return (req, res, next) => {
* if (!roles.includes(req.user.role)) {
* throw new UnauthorizedError("Unauthorized to access this route");
* }
* next();
* };
* };
* userRouter
* import function(bottom one) and add authorize (bottom one)
* import { validateUpdateUserInput } from "../middleware/validationMiddleware.js";
* import { authorizePermissions } from "../middleware/authMiddleware.js";
* router.get("/current-user", getCurrentUser);
* router.get(
* "/admin/app-stats",
* authorizePermissions("admin"),
* getApplicationStats
* );

**Proxy Setup**

* A proxy is front-end development is a server that acts as an intermediary between the client-side application and an external API, helping to bypass security restrictions and handle requests to different domains.
* It allows the font-end application to make API requests through the proxy, which forwards the request to the external API and returns the responses back to the application
* Only in dev env
* A must since cookies are sent back to the same server
* Spin up both servers
* Run up both server on VSCode
* PS C:\Users\sbdhl\OneDrive\Desktop\2024 MERN\jobify> npm run dev
* PS C:\Users\sbdhl\OneDrive\Desktop\2024 MERN\jobify\client> npm run dev
* Divide the VSCode terminal into 2
* Open up the front End
* ➜ Local: <http://localhost:5173/>
* Set up a dummy route in the server
* server.js
* above the jobRouter
* app.get("/api/v1/test", (req, res) => {
* res.json({ msg: "test route" });
* });
* app.use("/api/v1/jobs", authenticateUser, jobRouter);
* client – src – main.js (the fetch method)
* import "./index.css";
* fetch("http://localhost:5100/api/v1/test")
* .then((res) => res.json())
* .then((data) => console.log(data));
* createRoot(document.getElementById("root")).render(
* when you go to the application in the browser on the console you will have a lot of errors
* The solution is to setup the proxy on front end during development
* The front end and the back end should be on the same server
* Client – vite.config.js
* vite.config.js
* update the defineConfig function
* export default defineConfig({
* plugins: [react()],
* server: {
* proxy: {
* "/api": {
* target: "http://localhost:5100/api",
* changeOrigin: true,
* rewrite: (path) => path.replace(/^\/api/, ""),
* },
* },
* },
* });
* Update the fetch method main.js
* fetch("api/v1/test")
* .then((res) => res.json())
* .then((data) => console.log(data));
* On the web browser we should see the following message
* *{msg: 'test route'}*
* This code configures a proxy rule for the development server, specifically for requests that start with /api. Let's go through each property:

**Concurrently**

* The concurrently npm package is a utility that allows you to run multiple commands concurrently in the same terminal window. It provides a convenient way to execute multiple tasks or processes simultaneously
* Download the package
* npm i concurrently
* close both terminals
* Jobify – package.json
* "scripts": {
* "setup-project": "npm i && cd client && npm i",
* "server": "nodemon server",
* "client": "cd client && npm run dev",
* "dev": "concurrently --kill-others-on-fail \" npm run server\" \" npm run client\""
* },
* Open up the terminal
* Cd to the root (Jobify)
* Npm run dev

**Axios**

* Axios is a popular JavaScript library that simplifies the process of making HTTP request from web browser or Node.js. It provides a simple and elegant API for performing asynchronous HTTP requests, supporting features such as making GET, POST, PUT and DELETE request, handling request and response header, handling request cancellation, and more
* [Axios Docs](https://axios-http.com/docs/intro)
* Install package on client side
* npm i axios@
* client – main.jsx
* import package
* function
* delete fetch
* import axios from "axios";
* const data = await axios.get("/api/v1/test");
* console.log(data);
* createRoot(document.getElementById("root")).render(
* check the console log on the browser
* setup a custom instance of axios
* client/src/utils/customFetch.js
* customFetch.js
* import axios from "axios";
* const customFetch = axios.create({
* baseURL: "/api/v1",
* });
* export default customFetch;
* main.jsx
* import "./index.css";
* import customFetch from "./utils/customFetch.js";
* const data = await customFetch.get("/test");
* console.log(data);
* createRoot(document.getElementById("root")).render(
* after you are done remove the everything it was for demonstration

**Section 10: Register Page**

**React Router Action Intro**

* React Router – Action
* Route actions are the “writes” to route loader “reads”. They provide a way for apps to perform data mutations with simple HTML and HTTP semantics while React Router abstracts away the complexity of asynchronous UI and revalidation.
* This gives you the simple mental model of HTML + HTTP (where the browser handles the asynchrony and revalidation) with the behaviour and UX capabilities of modern SPAs

First Action

* Src – pages – Register.jsx
* Register.jsx
* Import form, redirect, useNavigation (the top one)
* import { Form, redirect, useNavigation, Link } from "react-router-dom";
* import Wrapper from "../assets/wrappers/RegisterAndLoginPage";
* client – src – App.jsx
* App.jsx
* Insert action of the register page
* element: <Landing />,
* },
* {
* path: "register",
* element: <Register />,
* action: () => {
* console.log("hello there");
* return null;
* },
* },
* The action function should always return something. If not it will send out an error
* Replace the form element with the Form component
* [FormData API - JS Nuggets](https://youtu.be/5-x4OUM-SP8)
* [FormData API - React ](https://youtu.be/WrX5RndZIzw)
* Register.jsx
* <Wrapper>
* <Form method="post" className="form">
* Go to the web browser
* Navigate to the register page
* Submit
* Console.log
* hello there
* But the best way to set up actions to the set it up in the Register and export it
* Register.jsx
* Botton one, the export function
* import { FormRow, Logo } from "../components";
* export const action = async (data) => {
* console.log(data);
* return null;
* };
* App.jsx
* Import the action function
* } from "./pages";
* import { action as registerAction } from './pages/Register';
* registerAction is an alias name
* replace the function with the imported one
* element: <Landing />,
* },
* {
* path: "register",
* element: <Register />,
* action: registerAction,
* },
* Test
* Go to the web browser
* Register page
* Submit
* Console.log
* An object with a lot of properties

Register User Complete

* Thunder Client
* Auth Routes
* Register User
* Reference the FormData Api JS Nuggets and FormData API React
* Clear MongoDB data
* Both the jobs and users
* Register.jsx
* Update the action function
* Import customFetch (the bottom one)
* import { FormRow, Logo } from "../components";
* import customFetch from "../utils/customFetch";
* update the action function
* export const action = async ({ request }) => {
* const formData = await request.formData();
* const data = Object.fromEntries(formData);
* try {
* await customFetch.post("./auth/register", data);
* return null;
* } catch (error) {
* console.log(error);
* return error;
* }
* };
* Test
* Web browser
* Register Page
* Submit
* Inspect
* Network
* Go to MongoDB (website)
* Return null mean that after we send the request we do nothing
* The details in the website will show up in the database
* After registering you have to redirect the user
* Update the action function
* export const action = async ({ request }) => {
* const formData = await request.formData();
* const data = Object.fromEntries(formData);
* try {
* await customFetch.post("./auth/register", data);
* return redirect("/login");
* } catch (error) {
* console.log(error);
* return error;
* }
* };
* Testing the error
* Register Page
* Send the same details that is already in the Database
* AxiosError – response – data
* {msg: 'email already exists'}
* Change name and email but make the password shorter
* Submit
* AxiosError – response – data
* {msg: 'password must be at least 8 characters long'}
* Provide all the correct values and submit
* We will be redirected to the login page

Navigation State

* useNavigation
* this hook tells you everything you need to know about a page navigation to build pending navigation indicators and optimistic UI on data mutations. Things like
* a) Global loading indicators
* b) Adding busy indicators to submit buttons
* navigation state
* idle – There is no navigation pending
* submitting – a route action is being called due to a form submission using POST, PUT, PATCH, or DELETE
* loading – The loaders for the next routes are being called to render the next page
* Register.jsx
* Import useNavigaton
* import { Form, redirect, useNavigation, Link } from "react-router-dom";
* Update the Register function (navigation, isSubmitting)
* return error;
* }
* };
* const Register = () => {
* const navigation = useNavigation();
* Console.log(navigaton)
* const isSubmitting = navigation.state === "submitting";
* return (
* <Wrapper>
* Go to submit button
* Update the submit button (check button)
* <FormRow type="password" name="password" defaultValue="secret123" />
* <button type="submit" className="btn btn-block" isabled={isSubmitting}>
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* Go to the browser
* Register page
* Insert new user details
* Submit
* Check console

**React Toastify**

* [React Toastify](https://fkhadra.github.io/react-toastify/introduction)
* Install the package
* npm i react-toastify
* To will provide nice alerts to the user
* Main.jsx
* Import the library
* import "react-toastify/dist/ReactToastify.css";
* import "./index.css";
* import { ToastContainer } from "react-toastify";
* add the toast component
* Go to Register.jsx
* Import toast library (the bottom one)
* import customFetch from "../utils/customFetch";
* import { toast } from "react-toastify";
* update the try and catch block (check try and catch block)
* export const action = async ({ request }) => {
* const formData = await request.formData();
* const data = Object.fromEntries(formData);
* try {
* await customFetch.post("/auth/register", data);
* toast.success("Registration successful");
* return redirect("/login");
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* return error;
* }
* };
* Go to the browser
* Submit already uploaded data
* Pop up error message
* Now upload new user details
* Success popup message

**Section 11: Dashboard and Login**

**Login user**

* Go to client – src – pages – Login.jsx
* Copy and paste the imports below (new:
* import { Link, Form, redirect, useNavigation } from "react-router-dom";
* import Wrapper from "../assets/wrappers/RegisterAndLoginPage";
* import { FormRow, Logo } from "../components";
* import customFetch from "../utils/customFetch";
* import { toast } from "react-toastify";
* add action function (export const action)
* import { toast } from "react-toastify";
* export const action = async ({ request }) => {
* return null;
* };
* Go to App.jsx
* Import the action function (the bottom one)
* import { action as registerAction } from "./pages/Register";
* import { action as loginAction } from "./pages/Login";
* insert in on the login route (same file App.jsx)
* action: registerAction,
* },
* {
* path: "login",
* element: <Login />,
* action: loginAction,
* },
* Go to Login.jsx
* Update the action function
* export const action = async ({ request }) => {
* const formData = await request.formData();
* const data = Object.fromEntries(formData);
* try {
* await customFetch.post("/auth/login", data);
* toast.success("Login successful");
* return redirect("/dashboard");
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* return error;
* }
* };
* Update the Login function
* const Login = () => {
* const navigation = useNavigation();
* const isSubmitting = navigation.state === "submitting";
* return (
* Still on the Login Function now update the login button
* <FormRow type="password" name="password" defaultValue="secret123" />
* <button type="submit" className="btn btn-block" isabled={isSubmitting}>
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* Nothing happened because of the Form Component
* Insert the Form component
* Login function (Form component and post method)
* return (
* <Wrapper>
* <Form method="post" className="form">
* Go to browser
* Login page
* Submit
* Inspect
* Application
* Cookies

**UserActionData Hook**

* Optional for this project but will include its notes
* Login.jsx
* Import useActionData (the top one)
* import { Link, Form, redirect, useNavigation , useActionData } from "react-router-dom";
* import Wrapper from "../assets/wrappers/RegisterAndLoginPage";
* update the action function
* const data = Object.fromEntries(formData);
* const errors = {msg: ""}
* if(data.password.length < 3){
* errors.msg = 'password too short'
* return errors
* }
* Update the login component (function)
* const Login = () => {
* const errors = useActionData()
* return (
* Add the output under the H4 heading
* <Logo />
* <h4>login</h4>
* {errors?.msg && <p style={{ color: "red" }}>{errors.msg}</p>}
* Go to browser
* Login page
* Upload short password
* Submit
* Delete everything add on this section (not applicable for this app)

**Loaders**

* Each route can define a “loader” function to provide data to the route element before it renders.
* Must return a value
* DashboardLayout.jsx
* Create a loader function
* import { checkDefaultTheme } from "../App";
* export const loader = async () => {
* return "hello world";
* };
* App.jsx
* Import the loader function (the bottom one)
* import { action as loginAction } from "./pages/Login";
* import { loader as dashboardLoader } from "./pages/DashboardLayout";
* still on App.js
* navigate to dashboardlayout route
* action: loginAction,
* },
* {
* path: "dashboard",
* element: <DashboardLayout />,
* loader: dashboardLoader,
* children:
* DashboardLayout.jsx
* Import redirect and useLoaderData (the top one)
* import { Outlet, redirect, useLoaderData } from 'react-router-dom';
* import Wrapper from "../assets/wrappers/Dashboard";
* update the
* DashboardLayout function
* const DashboardLayout = () => {
* const data = useLoaderData();
* console.log(data);
* // temp
* Browser
* Refresh the dashboardlayout page
* Console – hello world

**Get Current User**

* We get the current user in the dashboardlayout page
* Will provide two information. Info about the current user and authentication for the dashboard
* Import the customFetch (the bottom one)
* import { checkDefaultTheme } from "../App";
* import customFetch from "../utils/customFetch";
* when we login the cookie is generated with the user details
* DashboardLayout.jsx
* Update the loader function
* export const loader = async () => {
* try {
* const { data } = await customFetch.get("/users/current-user");
* return data;
* } catch (error) {
* return redirect("/");
* }
* };
* Browser
* Application
* Clear up everything (clear all icon underneath lighthouse)
* DashboardLayout.jsx
* Remove the temp value (hard coded) (top 2 lines)
* const DashboardLayout = () => {
* const { user } = useLoaderData();
* const [showSidebar, setShowSidebar] = useState(false);
* Browser
* Top right button will show the name of the logged person
* Inserting context in the outlet component
* If you want to pass the name to the pages
* <Navbar />
* <div className="dashboard-page">
* <Outlet context={{ user }} />
* </div>

**Logout**

* DashboardLayout.jsx
* Import useNavigate
* import { Outlet, redirect, useLoaderData, useNavigate} from "react-router-dom";
* update DashboardLayout function (bottom one)
* const DashboardLayout = () => {
* const { user } = useLoaderData();
* const navigate = useNavigate();
* Import toast and customFetch
* import { checkDefaultTheme } from "../App";
* import customFetch from "../utils/customFetch";
* import { toast } from "react-toastify";
* update the logout function
* const logoutUser = async () => {
* navigate("/");
* await customFetch.get("/auth/logout"); //clear cookies
* toast.success("Logging out...");
* };
* Browser
* Logout
* Redirect to home page
* Inspect
* Application
* As you logout the cookies are clear and when you log are there are generated again

**Section 12: Jobs Pages**

**Add Job Structure**

* AddJjobs.jsx
* Add imports
* import { FormRow } from "../components";
* import Wrapper from "../assets/wrappers/DashboardFormPage";
* import { useOutletContext } from "react-router-dom";
* import { JOB\_STATUS, JOB\_TYPE } from "../../../utils/constants";
* import { Form, useNavigation, redirect } from "react-router-dom";
* import { toast } from "react-toastify";
* import customFetch from "../utils/customFetch";
* const AddJobs = () => {
* update the AddJobs function
* const AddJobs = () => {
* const { user } = useOutletContext();
* const navigation = useNavigation();
* const isSubmitting = navigation.state === "submitting";
* return (
* <Wrapper>
* <Form method="post" className="form">
* <h4 className="form-title">add job</h4>
* <div className="form-center">
* <FormRow type="text" name="position" />
* <FormRow type="text" name="company" />
* <FormRow
* type="text"
* labelText="job location"
* name="jobLocation"
* defaultValue={user.location}
* />
* <button
* type="submit"
* className="btn btn-block form-btn "
* disabled={isSubmitting}
* >
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* </div>
* </Form>
* </Wrapper>
* );
* };
* Browser
* Login
* AddJob

**Select Input**

* Update the AddJobs function
* Add form-row div between .form-center and button
* Place the content in the div on the separate component and import that component
* Client – src – component
* Create a file – FormRowSelect.jsx
* FormRowSelect.jsx
* const FormRowSelect = ({name, labelText, list, defaultValue = ""}) => {
* return (
* <div className="form-row">
* <label htmlFor={name} className="form-label">
* {labelText || name}
* </label>
* <select
* name={name}
* id={name}
* className="form-select"
* defaultValue={defaultValue}
* >
* {list.map((itemValue) => {
* return (
* <option key={itemValue} value={itemValue}>
* {itemValue}
* </option>
* );
* })}
* </select>
* </div>
* );
* };
* export default FormRowSelect;
* client – src – component – index.js
* import FormRowSelect component (bottom one)
* export { default as Navbar } from "./Navbar";
* export {default as FormRowSelect} from "./FormRowSelect"
* AddJobs.jsx
* Import FormRowSelect
* import { FormRow, FormRowSelect } from "../components";
* import Wrapper from "../assets/wrappers/DashboardFormPage";
* between FormRow and button add the component
* <FormRowSelect
* labelText="job status"
* name="jobStatus"
* defaultValue={JOB\_STATUS.PENDING}
* list={Object.values(JOB\_STATUS)}
* />
* <FormRowSelect
* labelText="job type"
* name="jobType"
* defaultValue={JOB\_TYPE.FULL\_TIME}
* list={Object.values(JOB\_TYPE)}
* />

Create Job Functionality

* AddJobs.jsx
* Add the action function
* import customFetch from "../utils/customFetch";
* export const action = async ({ request }) => {
* return null
* };
* App.jsx
* Import action function
* import { loader as dashboardLoader } from "./pages/DashboardLayout";
* import { action as addJobAction } from "./pages/AddJobs";
* insert it in the AddJobs route
* loader: dashboardLoader,
* children: [
* {
* index: true,
* element: <AddJobs />,
* action: addJobAction,
* },
* AddJobs.jsx
* Update the action function
* export const action = async ({ request }) => {
* const formData = await request.formData();
* const data = Object.fromEntries(formData);
* try {
* await customFetch.post("/jobs", data);
* toast.success("Job added successfully");
* return null;
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* return error;
* }
* };
* Browser
* Add Job page
* Inspect
* Network
* Fetch/XHR
* Redirect after adding the job
* AddJobs.jsx
* Update the action function on the try block (return line)
* try {
* await customFetch.post("/jobs", data);
* toast.success("Job added successfully");
* return redirect("all-jobs");
* }

Add Job CSS

* See the video

**All Jobs Structure**

* Client – components – create a component – JobsContainer.jsx
* JobsContainer.jsx
* Rafce
* Client – components – create a component – SeachContainer.jsx
* SearchContainer.jsx
* Rafce
* Export both components to client – components – index.js (at the bottom)
* export {default as JobsContainer} from "./JobsContainer"
* export {default as SearchContainer} from "./SearchContainer"
* AllJobs.jsx
* Import libraries and component and files
* import { toast } from "react-toastify";
* import { JobsContainer, SearchContainer } from "../components";
* import customFetch from "../utils/customFetch";
* import { useLoaderData } from "react-router-dom";
* import { useContext, createContext } from "react";
* const AllJobs = () => {
* AllJobs.jsx
* Create a loader function (under the import)
* import { useContext, createContext } from "react";
* export const loader = async () => {
* return null;
* };
* App.jsx
* Import the loader function (componenet) (the bottom one)
* import { action as addJobAction } from "./pages/AddJobs";
* import {loader as AllJobsLoader} from "./pages/AllJobs"
* add the loader on the AllJobs route (still on App.jsx)
* element: <Stats />,
* },
* {
* path: "all-jobs",
* element: <AllJobs />,
* loader: allJobsLoader,
* },
* AllJobs.jsx
* Update the loader function
* export const loader = async () => {
* try {
* const { data } = await customFetch.get("/jobs");
* return {
* data,
* };
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* return error;
* }
* };
* Use the useLoaderData and insert components (still on App.jsx)
* const AllJobs = () => {
* const { data } = useLoaderData();
* Console.log(data)
* return (
* <>
* <SearchContainer />
* <JobsContainer />
* </>
* );
* };
* Browser
* Login
* AllJobs
* Shows Seach Container and Jobs Container
* Inspect
* Console log
* Show data console log from AllJobs function

**All Jobs Context**

* AllJobs.jsx
* Create jobs context (middle line)
* };
* const AllJobsContext = createContext();
* const AllJobs = () => {
* update the AllJobs return div(< >) (return block)
* console.log(data);
* return (
* <AllJobsContext.Provider value={{ data }}>
* <SearchContainer />
* <JobsContainer />
* </AllJobsContext.Provider>
* );
* Set up the hook (top export)
* };
* export const useAllJobsContext = () => useContext(AllJobsContext);
* export default AllJobs;

**Jobs Container JSX**

* we know render all the jobs
* client – src – components – create Job.jsx
* Job.jsx
* const Job = () => {
* return <h3>Job</h3>;
* };
* export default Job;
* JobsContainer.jsx
* Import the Job file(component) and Wrapper
* import Job from "./Job"
* import Wrapper from "../assets/wrappers/JobsContainer";
* const JobsContainer = () => {
* import useAllJobsContenx (bottom one) (still on JobsContainer.jsx)
* import Wrapper from "../assets/wrappers/JobsContainer";
* import { useAllJobsContext } from "../pages/AllJobs";
* update the JobsContainer function (still on JobsContainer.jsx)
* const JobsContainer = () => {
* const { data } = useAllJobsContext();
* const { jobs } = data;
* if (jobs.length === 0) {
* return (
* <Wrapper>
* <h2>No jobs to display...</h2>
* </Wrapper>
* );
* }
* return (
* <Wrapper>
* <div className="jobs">
* {jobs.map((job) => {
* return <Job key={job.\_id} {...job} />;
* })}
* </div>
* </Wrapper>
* );
* };

**Jobs Container CSS (optional)**

* See the video

**Job Component**

* [Dayjs Docs](https://day.js.org/docs/en/installation/installation)
* Install dayjs package
* npm i dayjs
* client – src – components – create JobInfo.jsx file
* JobInfo.jsx
* Rafce
* const JobInfo = () => {
* return <h4>JobInfo</h4>;
* };
* export default JobInfo;
* Job.jsx
* Import the following libraries
* import { FaLocationArrow, FaBriefcase, FaCalendarAlt } from "react-icons/fa";
* import { Link, Form } from "react-router-dom";
* import Wrapper from "../assets/wrappers/Job";
* import JobInfo from "./JobInfo";
* import day from "dayjs";
* import advancedFormat from "dayjs/plugin/advancedFormat";
* day.extend(advancedFormat);
* const Job = () => {
* Outline the properties
* const Job = ({
* id,
* position,
* company,
* jobLocation,
* jobType,
* createdAt,
* jobStatus,
* }) => {
* console.log(createdAt);
* return <h3>Job</h3>;
* Update the Job function
* const Job = ({
* \_id,
* position,
* company,
* jobLocation,
* jobType,
* createdAt,
* jobStatus,
* }) => {
* const date = day(createdAt).format("MMM Do, YYYY");
* return (
* <Wrapper>
* <header>
* <div className="main-icon">{company.charAt(0)}</div>
* <div className="info">
* <h5>{position}</h5>
* <p>{company}</p>
* </div>
* </header>
* <div className="content">
* <div className="content-center">
* <JobInfo icon={<FaLocationArrow />} text={jobLocation} />
* <JobInfo icon={<FaCalendarAlt />} text={date} />
* <JobInfo icon={<FaBriefcase />} text={jobType} />
* <div className={`status ${jobStatus}`}>{jobStatus}</div>
* </div>
* <footer className="actions">
* <Link className="btn edit-btn">Edit</Link>
* <Form>
* <button type="submit" className="btn delete-btn">
* Delete
* </button>
* </Form>
* </footer>
* </div>
* </Wrapper>
* );
* };
* JobInfo.jsx
* Import wrapper
* import Wrapper from "../assets/wrappers/JobInfo";
* update JobInfo function
* const JobInfo = ({ icon, text }) => {
* return (
* <Wrapper>
* <span className="job-icon">{icon}</span>
* <span className="job-text">{text}</span>
* </Wrapper>
* );
* };

**Job Component CSS (option)**

* See the video

**Edit Job Setup**

* To edit you first need to Get Single Job
* {{URL}}/jobs/id
* App.jsx
* Add EditJob on the import
* Admin,
* Profile,
* EditJob,
* } from "./pages";
* Its Route will be on the DashboardLayout as one of the children
* Dashboard route
* path: "dashboard",
* element: <DashboardLayout />,
* loader: dashboardLoader,
* children: [
* {
* Insert EditJob
* element: <Admin />,
* },
* {
* path: 'edit-job',
* element: <EditJob/>
* }
* Job Component (assets – component)
* Job.jsx
* Edit the Job function – second – add to=””
* <footer className="actions">
* <Link to="/edit-job" className="btn edit-btn">
* Edit
* </Link>
* In the Browser AllJobs when clicking edit it gives an error
* Add dashboard to the link
* <footer className="actions">
* <Link to="/dashboard/edit-job" className="btn edit-btn">
* Edit
* </Link>
* We can also you the double dot .. in the place of dashboard
* <footer className="actions">
* <Link to="../edit-job" className="btn edit-btn">
* Edit
* </Link>
* Double dots
* <footer className="actions">
* <Link to="../edit-job" className="btn edit-btn">
* Edit
* </Link>
* When clicking edit bring in back the details of the job clicked
* <footer className="actions">
* <Link to={`../edit-job/${\_id}`} className="btn edit-btn">
* Edit
* </Link>
* <Form>
* For now it will give us error because the params are not set yet
* App.jsx
* Update the path for the EditJob route
* {
* path: "edit-job/:id",
* element: <EditJob />,
* },

**Route Params**

* EditJob.jsx (client – src - pages)
* Import the libraries
* import { FormRow, FormRowSelect } from "../components";
* import Wrapper from "../assets/wrappers/DashboardFormPage";
* import { useLoaderData } from "react-router-dom";
* import { JOB\_STATUS, JOB\_TYPE } from "../../../utils/constants";
* import { Form, useNavigation, redirect } from "react-router-dom";
* import { toast } from "react-toastify";
* import customFetch from "../utils/customFetch";
* create the loader function (still on the EditJob.jsx)
* import customFetch from "../utils/customFetch";
* export const loader = async () => {
* return null;
* };
* Underneath the loader function create an action function
* export const action = async () => {
* return null;
* };
* const EditJob = () => {
* import the above two functions
* App.jsx
* Import function (first 2 lines)
* import { loader as editJobLoader } from "./pages/EditJob";
* import { action as editJobAction } from "./pages/EditJob";
* export const checkDefaultTheme = () => {
* go to EditJob route (still on App.jsx)
* {
* path: "edit-job/:id",
* element: <EditJob />,
* loader: editJobLoader,
* action: editJobAction,
* },
* Accessing the parameters
* EditJob.jsx
* import customFetch from "../utils/customFetch";
* export const loader = async ({ params }) => {
* console.log(params);
* return null;
* };
* Browser
* AllJobs
* Edit button
* Inspect
* Console log
* Will display id
* We can also have access of the id from the EditJob component
* Import useParams
* import { useLoaderData, useParams } from "react-router-dom";
* update the EditJob component
* const EditJob = () => {
* const params = useParams();
* console.log(params);
* return <h1>EditJob Page</h1>;
* Browser
* Inspect
* Console log
* Two console log output with same id

**Edit Job Loader**

* EditJob.jsx
* Update the loader function
* export const loader = async ({ params }) => {
* try {
* const { data } = await customFetch.get(`/jobs/${params.id}`);
* return data;
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* return redirect("/dashboard/all-jobs");
* }
* };
* Update the EditJob component
* const EditJob = () => {
* const { job } = useLoaderData();
* console.log(job);
* return <h1>EditJob Page</h1>;
* Browser
* Inspect
* Console log
* Job details in the console output

Edit Job JSX

* EditJob.jsx
* Update the EditJob component
* const EditJob = () => {
* const { job } = useLoaderData();
* const navigation = useNavigation();
* const isSubmitting = navigation.state === "submitting";
* return (
* <Wrapper>
* <Form method="post" className="form">
* <h4 className="form-title">edit job</h4>
* <div className="form-center">
* <FormRow type="text" name="position" defaultValue={job.position} />
* <FormRow type="text" name="company" defaultValue={job.company} />
* <FormRow
* type="text"
* labelText="job location"
* name="jobLocation"
* defaultValue={job.jobLocation}
* />
* <FormRowSelect
* name="jobStatus"
* labelText="job status"
* defaultValue={job.jobStatus}
* list={Object.values(JOB\_STATUS)}
* />
* <FormRowSelect
* name="jobType"
* labelText="job type"
* defaultValue={job.jobType}
* list={Object.values(JOB\_TYPE)}
* />
* <button
* type="submit"
* className="btn btn-block form-btn "
* disabled={isSubmitting}
* >
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* </div>
* </Form>
* </Wrapper>
* );
* };

**Edit Job Action**

* EditJob.jsx
* Update the action function
* export const action = async ({ request, params }) => {
* const formData = await request.formData();
* const data = Object.fromEntries(formData);
* try {
* await customFetch.patch(`/jobs/${params.id}`, data);
* toast.success("Job edited successfully");
* return redirect("/dashboard/all-jobs");
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* return error;
* }
* };
* Browser
* Edit
* Change details
* Submit
* For now it is not changing (troubleshoot)

**Delete Job**

* Method post
* {{URL}}/jobs/id
* Job.jsx
* Update the Job component (function) – the delete button
* </Link>
* <Form method="post" action={`../delete-job/${\_id}`}>
* <button type="submit" className="btn delete-btn">
* Delete
* </button>
* </Form>
* App.jsx
* Add the delete route (underneath EditJob route)
* {
* path: "edit-job/:id",
* element: <EditJob />,
* loader: editJobLoader,
* action: editJobAction,
* },
* {
* path: "delete-job/:id"
* }
* Client – src – pages – DeleteJob.jsx
* DeleteJob.jsx
* Import the libraries
* import { redirect } from "react-router-dom";
* import customFetch from "../utils/customFetch";
* import { toast } from "react-toastify";
* create action function (still on DeleteJob.jsx)
* export const action = async ({params}) => {
* try {
* await customFetch.delete(`/jobs/${params.id}`);
* toast.success("Job deleted successfully");
* } catch (error) {
* toast.error(error.response.data.msg);
* }
* return redirect("/dashboard/all-jobs");
* }
* const DeleteJob = () => {
* import the action function (component to App.jsx)
* App.jsx
* import {action as editJobAction} from "./pages/EditJob";
* import {action as deleteJobAction} from "./pages/DeleteJob"
* update the deleteJob route (still on App.jsx)
* action: editJobAction,
* },
* {
* path: "delete-job/:id",
* action: deleteJobAction,
* },
* DeleteJob.jsx
* Remove/delete the DeleteJob component
* const DeleteJob = () => {
* return <h1>DeleteJob Page</h1>;
* };
* export default DeleteJob;

Section 13: Admin Page

Admin Page Setup

* {{URL}}/users/admin/
* This page will be available to the user whose role is admin
* Client – src – pages – Admin.jsx
* Admin.jsx
* Import the libraries
* import { FaSuitcaseRolling, FaCalendarCheck } from "react-icons/fa";
* import { useLoaderData, redirect } from "react-router-dom";
* import customFetch from "../utils/customFetch";
* import Wrapper from "../assets/wrappers/StatsContainer";
* import { toast } from "react-toastify";
* const Admin = () => {
* create the loader function (still on Admin.jsx)
* export const loader = async () => {
* try {
* const response = await customFetch.get("/users/admin/app-stats");
* return response.data;
* } catch (error) {
* toast.error("You are not authorized to view this page");
* return redirect("/dashboard");
* }
* };
* Update the Admin component
* const Admin = () => {
* const { users, jobs } = useLoaderData();
* return (
* <Wrapper>
* <h2>admin page</h2>
* </Wrapper>
* );
* };
* App.jsx
* Import the loader component
* import { action as deleteJobAction } from "./pages/DeleteJob";
* import {loader as adminLoader} from "./pages/Admin"
* add the loader to the admin route
* {
* path: "admin",
* element: <Admin />,
* loader: adminLoader,
* },
* {
* path: "edit-job/:id",
* Browser
* Click Admin
* Will display admin page
* Remove the link from navlinks if the user is not admin
* Client – src – components – navlinks
* NavLinks.jsx
* Update the NavLilnks component (line 4,5,6)
* return (
* <div className="nav-links">
* {links.map((link) => {
* const { text, path, icon } = link;
* const { role } = user;
* if (role !== "admin" && path === "admin") return;

Admin Page Complete

* Create a statistic file
* Client – src – component – create file - StatItem.jsx
* StatItems.jsx
* Rafce
* const StatItem = () => {
* return <div>StatItem</div>;
* };
* export default StatItem;
* insert the params
* import the Wrapper (top one)
* import Wrapper from "../assets/wrappers/StatItem";
* const StatItem
* update the StatItem component
* const StatItem = ({ count, title, icon, color, bcg }) => {
* return (
* <Wrapper color={color} bcg={bcg}>
* <header>
* <span className="count">{count}</span>
* <span className="icon">{icon}</span>
* </header>
* <h5 className="title">{title}</h5>
* </Wrapper>
* );
* };
* Export StatItem component to index.js
* Client – src – component – index.js
* Index.js (the bottom one)
* export { default as SearchContainer } from "./SearchContainer";
* export {default as StatItem} from "./StatItem"
* Admin.jsx
* Import StatItem component
* import { toast } from "react-toastify";
* import { StatItem } from "../components";
* add the component on Admin function (component)
* const Admin = () => {
* const { users, jobs } = useLoaderData();
* return (
* <Wrapper>
* <StatItem/>
* </Wrapper>
* Add the params for the StatItem component
* const Admin = () => {
* const { users, jobs } = useLoaderData();
* return (
* <Wrapper>
* <StatItem
* title="current users"
* count={users}
* color="#e9b949"
* bcg="#fcefc7"
* icon={<FaSuitcaseRolling />}
* />
* <StatItem
* title="total jobs"
* count={jobs}
* color="#647acb"
* bcg="#e0e8f9"
* icon={<FaCalendarCheck />}
* />
* </Wrapper>
* );
* };

Section 14: Profile Page

**Avatar Images**

* Get two images from pexels
* [pexels](https://www.pexels.com/search/person/)

**Public Folder**

* Making assets publicly available
* const \_\_dirname = dirname(fileURLToPath(import.meta.url));
* app.use(express.static(path.resolve(\_\_dirname, "./public")));
* create a public folder in the root (jobify)
* public
* create a folder – uploads
* make the imports
* server.js (under routers imports)
* import userRouter from "./routes/userRouter.js";
* //public
* import { dirname } from "path";
* import { fileURLToPath } from "url";
* import path from "path";
* underneath the morgan middleware (the top one)
* const \_\_dirname = dirname(fileURLToPath(import.meta.url));
* if (process.env.NODE\_ENV === "development") {
* add the middleware (the top one)
* app.use(express.static(path.resolve(\_\_dirname, "./public")));
* app.use(cookieParser());
* the public folder in the root is where we will store temporary our images
* test
* upload image on public folder
* http://localhost:5100/imageName

**User Schema Update**

* clear data on MongoDB (users and jobs)
* root (jobify) – models – userModel.js
* userModel.js
* update UserSchema (last part)
* default: "user",
* },
* avatar: String,
* avatarPublicId: String,
* });
* Register a user on the Browser (front end)

**Profile Page Structure**

* Client – src – pages – Profile.jsx
* Import the libraries
* import { FormRow } from "../components";
* import Wrapper from "../assets/wrappers/DashboardFormPage";
* import { useOutletContext } from "react-router-dom";
* import { useNavigation, Form } from "react-router-dom";
* import customFetch from "../utils/customFetch";
* import { toast } from "react-toastify";
* const Profile = () => {
* enable to send formData not JSON data (below). Because we are sending a file to the server (image)
* encType="multipart/form-data"
* update the Profile component (function)
* const Profile = () => {
* const { user } = useOutletContext();
* const { name, lastName, email, location } = user;
* const navigation = useNavigation();
* const isSubmitting = navigation.state === "submitting";
* return (
* <Wrapper>
* <Form method="post" className="form" encType="multipart/form-data">
* <h4 className="form-title">profile</h4>
* <div className="form-center">
* <div className="form-row">
* <label htmlFor="avatar" className="form-label">
* Select an image file (max 0.5 MB):
* </label>
* <input
* type="file"
* id="avatar"
* name="avatar"
* className="form-input"
* accept="image/\*"
* />
* </div>
* <FormRow type="text" name="name" defaultValue={name} />
* <FormRow
* type="text"
* labelText="last name"
* name="lastName"
* defaultValue={lastName}
* />
* <FormRow type="email" name="email" defaultValue={email} />
* <FormRow type="text" name="location" defaultValue={location} />
* <button
* className="btn btn-block form-btn"
* type="submit"
* disabled={isSubmitting}
* >
* {isSubmitting ? "submitting..." : "save changes"}
* </button>
* </div>
* </Form>
* </Wrapper>
* );
* };
* Browser
* Click profile
* Choose file functionality not working yet

**Profile Action**

* Update User
* {{URL}}/users/update-user
* Method patch
* When upload a file (image) we still need to do some work on the server side
* Create the action function
* export const action = async ({ request }) => {
* const formData = await request.formData();
* const file = formData.get("avatar");
* if (file && file.size > 500000) {
* toast.error("Image size too large");
* return null;
* }
* try {
* await customFetch.patch("/users/update-user", formData);
* toast.success("Profile updated successfully");
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* }
* return null;
* };
* Export it to App.jsx
* App.jsx (the bottom one)
* import { loader as adminLoader } from "./pages/Admin";
* import {action as profileAction} from "./pages/Profile"
* add the action into the profile route (the bottom one)
* element: <AllJobs />,
* loader: allJobsLoader,
* },
* {
* path: "profile",
* element: <Profile />,
* action: profileAction,
* },

**Setup Multer**

* Multer is a pupular middleware package for handling multipart/form-data in Node.js web applications. It is commonly used for handling file uploads. Multer simplifies the process of accepting and storing files submitted through HTTP requests by providing an easy-to-use API. It integrates seamlessly with Express.js and allows developers to define upload destination, file size limits, and other configurations.
* Create multerMiddleware.js
* Jobify – middleware – multerMiddleware.js
* multerMiddleware.js
* import the multer and setup the functionality
* import multer from "multer";
* const storage = multer.diskStorage({
* destination: (req, file, cb) => {
* // set the directory where uploaded files will be stored
* cb(null, "public/uploads");
* },
* filename: (req, file, cb) => {
* const fileName = file.originalname;
* // set the name of the uploaded file
* cb(null, fileName);
* },
* });
* const upload = multer({ storage });
* export default upload;
* we will use it in the userRoute.js
* userRoute.js
* import the upload (the bottom one)
* import { authorizePermissions } from "../middleware/authMiddleware.js";
* import upload from "../middleware/multerMiddleware.js";
* insert upload in the router (middle one)
* getApplicationStats
* );
* router.patch("/update-user", upload.single('avatar'), validateUpdateUserInput, updateUser);
* export default router;
* userController.js
* export const updateUser = async (req, res) => {
* console.log(req.file);

Cloudinary

* [Cloudinary](https://cloudinary.com/)
* Cloudinary is a cloud-based media management platform that helps businesses store, optimize, and deliver images and videos across the web. It provides developers with an easy way to upload, manipulate, and server media assets, enabling faster and more efficient delivery of visual content on websites applications. Cloudinary also offers features like automatic resizing, format conversion, and responsive delivery to ensure optimal user experiences across different devices and network conditions
* Cloudinary will help us with retain our media on the production side because as soon as the app is in sleep we will lose our media.
* [www.cloudinary.com](http://www.cloudinary.com)
* Sign up for free
* Sign up with email
* Email: [sivngonja@gmail.com](mailto:sivngonja@gmail.com)
* Password: Convagency@cular9
* GET API Key
* Jobify - .env
* Post the following on .env
* CLOUD\_NAME=
* CLOUD\_API\_KEY=
* CLOUD\_API\_SECRET=
* CLOUD\_NAME is dczjrcdi4 (Cloud Name)
* Fill out the rest
* CLOUD\_NAME=dczjrcdi4
* CLOUD\_API\_KEY=823691941629658
* CLOUD\_API\_SECRET=ExQPyQGCY\_K--P6SOKOc5Rihl2M
* Server.js
* Import cloudinary the bottom one
* import cookieParser from "cookie-parser";
* import cloudinary from "cloudinary";
* insert the cloudinary config (the bottom one)
* import { authenticateUser } from "./middleware/authMiddleware.js";
* cloudinary.config({
* cloud\_name: process.env.CLOUD\_NAME,
* api\_key: process.env.CLOUD\_API\_KEY,
* api\_secret: process.env.CLOUD\_API\_SECRET,
* });

**Update User functionality**

* Controller – userController.js
* userController.js
* import cloudinary and promises
* import Job from "../models/JobModel.js";
* import cloudinary from 'cloudinary'
* import { promises as fs } from "fs";
* update the updateUser function
* export const updateUser = async (req, res) => {
* const newUser = { ...req.body };
* delete newUser.password;
* if (req.file) {
* const response = await cloudinary.v2.uploader.upload(req.file.path);
* await fs.unlink(req.file.path);
* newUser.avatar = response.secure\_url;
* newUser.avatarPublicId = response.public\_id;
* }
* const updatedUser = await User.findByIdAndUpdate(req.user.userId, newUser);
* if (req.file && updatedUser.avatarPublicId) {
* await cloudinary.v2.uploader.destroy(updatedUser.avatarPublicId);
* }
* res.status(StatusCodes.OK).json({ msg: "update user" });
* };
* Client – src – components – LogoutContainer.jsx
* Already updated
* Browser
* Register new user
* Login
* Profile
* Attach img
* Submit
* Pic will show next to the profile name
* Cloudinary
* Login
* Go to media library

**Submit Button Component**

* Client – src – component – create a SubmitBtn.jsx file
* SubmitBtn.jsx
* Rafce
* const SubmitBtn = () => {
* return <h1>SubmitBtn</h1>;
* };
* export default SubmitBtn;
* pages – AddJobs.jsx
* cut the following
* <button
* type="submit"
* className="btn btn-block form-btn "
* disabled={isSubmitting}
* >
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* Paste it on SubmitBtn.jsx
* const SubmitBtn = () => {
* return <button
* type="submit"
* className="btn btn-block form-btn "
* disabled={isSubmitting}
* >
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* };
* export default SubmitBtn;
* AddJobs.jsx
* Cut the following
* const navigation = useNavigation();
* const isSubmitting = navigation.state === "submitting";
* import use useNavigation
* SubmitBtn.jsx
* import { useNavigation } from "react-router-dom";
* const SubmitBtn = ({ formBtn }) => {
* const navigation = useNavigation();
* const isSubmitting = navigation.state === "submitting";
* return (
* <button
* type="submit"
* className={`btn btn-block ${formBtn && "form-btn"}`}
* disabled={isSubmitting}
* >
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* );
* };
* export default SubmitBtn;
* client – src – components - index.js
* export (the bottom one)
* export { default as StatItem } from "./StatItem";
* export { default as SubmitBtn } from "./SubmitBtn";
* AddJobs.jsx
* Remove useNavigation import
* Add SubmitBtn import
* import { FormRow, FormRowSelect, SubmitBtn } from "../components";
* import Wrapper from "../assets/wrappers/DashboardFormPage";
* add the SubmitBtn component (the top line)
* <SubmitBtn formBtn />
* </div>
* </Form>
* </Wrapper>
* EditJob.jsx
* Remove useNavigation import
* Import SubmitBtn component
* import { FormRow, FormRowSelect, SubmitBtn } from "../components";
* Delete the following
* const navigation = useNavigation();
* const isSubmitting = navigation.state === "submitting";
* detele the button
* <button
* type="submit"
* className="btn btn-block form-btn "
* disabled={isSubmitting}
* >
* {isSubmitting ? "submitting..." : "submit"}
* </button>
* add the SubmitBtn Component
* <SubmitBtn formBtn/>
* </div>
* </Form>
* </Wrapper>
* Profile.jsx
* Follow the step as EditJob.jsx above
* Login.jsx
* Follow the steps as EditJob.jsx above
* Register.jsx
* Follow the steps as EditJob.jsx above
* Test in the Browser (login, register a new user etc)

Create Test User

* Feel free to use one of the ChatGPT options
* {
* "name": "Zippy",
* "email": "test@test.com",
* "password": "secret123",
* "lastName": "ShakeAndBake",
* "location": "Codeville"
* }
* Thunder Client
* Auth Routes
* Register User
* Paste the value above
* Client – src – pages – Login.jsx
* Update the Login function
* const Login = () => {
* const navigate = useNavigate()  //import
* const loginDemoUser = async () => {
* const data = {
* email: 'test@test.com',
* password: 'secret123',
* };
* try {
* await customFetch.post('/auth/login', data);
* toast.success('take a test drive');
* navigate('/dashboard');
* } catch (error) {
* toast.error(error?.response?.data?.msg);
* }
* };
* return (
* Insert the loginDemoUser function in the explore app button
* <button type="button" className="btn btn-block" onClick={loginDemoUser}>explore the app</button>
* Browser
* Logout if logged in
* Click Explore the app
* We will log in the test user login details

Restrict Access to Test User

* Jobify – middleware – authMiddleware.js
* Import Baderror
* import { UnauthenticatedError, BadRequestError } from "../errors/customErrors.js";
* Update the authenticateUser function (the try block)
* try {
* const { userId, role } = verifyJWT(token);
* const testUser = userId === '67054693428a5033a37bfb35';  //test user Id from mongodb
* req.user = { userId, role, testUser };
* next();
* } catch (error) {
* Create a checkForTestUser middleware (underneath authorizePermissions)
* export const checkForTestUser = (req, res, next) => {
* if (req.user.testUser) {
* throw new BadRequestError("Demo User. Read Only!");
* }
* next();
* };
* Jobify – routes – jobRouter.js
* jobRouter.js
* import the checkForTestUser middleware (the bottom one)
* } from "../middleware/validationMiddleware.js";
* import { checkForTestUser } from "../middleware/authMiddleware.js";
* insert it in the createJob route
* router.route("/").get(getAllJobs).post(checkForTestUser, validateJobInput, createJob);
* add it in updateJob
* .patch(checkForTestUser, validateJobInput, validateIdParam, updateJob)
* Add to deleteJob route
* .delete(checkForTestUser, validateIdParam, deleteJob);
* userRouter.js
* import the checkForTestUser middleware (the bottom one)
* import { validateUpdateUserInput } from "../middleware/validationMiddleware.js";
* import { authorizePermissions, checkForTestUser } from "../middleware/authMiddleware.js";
* add the middleware in the update user route
* router.patch(
* "/update-user",
* checkForTestUser,
* upload.single("avatar"),
* validateUpdateUserInput,
* updateUser
* );
* export default router;
* Browser
* Explore the app
* Try to add a job
* Error message – Demo user only
* The demo user can only view all jobs and stats

**Mockaroo**

* [Mockaroo ](https://www.mockaroo.com/)
* Providing jobs by default for the restricted demo user
* Go to the website
* Remove all of the fields (x – sign)
* Click ADD ANOTHER FIELD to add field one by one
* jobStatus – custom list
* createdAt – datetime
* date formate – ISO 8610 (UTC)
* Rows 100
* Format – JSON
* Untick include null values
* Preview
* Generate
* Rename mockData.json
* Move file to jobity- utils

**Populate DB**

* Create new file on root (jobify) – populate.js
* Polulate.js
* Set up the following
* import { readFile } from "fs/promises";
* import mongoose from "mongoose";
* import dotenv from "dotenv";
* dotenv.config();
* import Job from "./models/JobModel.js";
* import User from "./models/UserModel.js";
* try {
* await mongoose.connect(process.env.MONGO\_URL);
* // const user = await User.findOne({ email: 'john@gmail.com' });
* const user = await User.findOne({ email: "test@test.com" });
* const jsonJobs = JSON.parse(
* await readFile(new URL("./utils/mockData.json", import.meta.url))
* );
* const jobs = jsonJobs.map((job) => {
* return { ...job, createdBy: user.\_id };
* });
* await Job.deleteMany({ createdBy: user.\_id });
* await Job.create(jobs);
* console.log("Success!!!");
* process.exit(0);
* } catch (error) {
* console.log(error);
* process.exit(1);
* }
* Test
* Exit development (CTRL + C)
* Node populate
* Please note that when you copy the generate mockData file when renaming just leave the .json
* Npm run dev
* Browser
* Login as the testuser
* All Jobs
* A Generated list will appear
* Both the test user and admin will have the populated data

**Section 15: Stats Page**

**Stats Route Setup**

* Jobify – controllers – jobController.js
* jobController.js
* create a new controller
* import mongoose and day.js libraries (the last to bottom)
* import { StatusCodes } from "http-status-codes";
* import mongoose from "mongoose";
* import day from "dayjs";
* create showStats controller (at the bottom)
* export const showStats = async (req, res) => {
* res.send("stats");
* };
* jobRouter.js
* import the showStats controller
* deleteJob,
* showStats,
* } from "../controllers/jobControllers.js";
* Add the showStats routee (the middle one)
* router
* .route("/")
* .get(getAllJobs)
* .post(checkForTestUser, validateJobInput, createJob);
* router.route('/stats').get(showStats)
* router
* Thunder Client
* Collections
* Jobify
* Job Routes
* Duplicate Get All Jobs
* Rename Show Stats
* Method GET
* {{URL}}/jobs/stats
* Send
* Response
* {
* "msg": "authentication invalid"
* }
* Login user
* Send
* “stats”
* For now we will hard code the value but in the coming section we will set them dynamically
* jobControllers.js
* update the showStats function
* export const showStats = async (req, res) => {
* const defaultStats = {
* pending: 22,
* interview: 11,
* declined: 4,
* };
* let monthlyApplications = [
* {
* date: "May 23",
* count: 12,
* },
* {
* date: "Jun 23",
* count: 9,
* },
* {
* date: "Jul 23",
* count: 3,
* },
* ];
* res.status(StatusCodes.OK).json({ defaultStats, monthlyApplications });
* };
* Thunder Client
* Show Stats
* Send
* Response
* "defaultStats": {
* "pending": 22,
* "interview": 11,
* "declined": 4
* },
* "monthlyApplications": [
* {
* "date": "May 23",
* "count": 12
* },
* {
* "date": "Jun 23",
* "count": 9
* },
* {
* "date": "Jul 23",
* "count": 3
* }
* ]
* }

**Group up Job Status**

* [MongoDB Docs](https://www.mongodb.com/docs/manual/core/aggregation-pipeline/)
* The MongoDB aggregation pipeline is like a factory line for data. Data enters, it goes through different stages like cleaning, sorting, or grouping, and comes out at the end changed in some way. It’s a way to process data inside MongoDB
* The showStats controller will be continued on the next section

Group by Monthly Applications

* Still working on the showStats controller (jobControllers.js)
* export const showStats = async (req, res) => {
* let stats = await Job.aggregate([
* { $match: { createdBy: new mongoose.Types.ObjectId(req.user.userId) } },
* { $group: { \_id: "$jobStatus", count: { $sum: 1 } } },
* ]);
* stats = stats.reduce((acc, curr) => {
* const { \_id: title, count } = curr;
* acc[title] = count;
* return acc;
* }, {});
* const defaultStats = {
* // for new users 0 data
* pending: stats.pending || 0,
* interview: stats.interview || 0,
* declined: stats.declined || 0,
* };
* let monthlyApplications = await Job.aggregate([
* { $match: { createdBy: new mongoose.Types.ObjectId(req.user.userId) } },
* {
* $group: {
* \_id: { year: { $year: "$createdAt" }, month: { $month: "$createdAt" } },
* count: { $sum: 1 },
* },
* },
* { $sort: { "\_id.year": -1, "\_id.month": -1 } },
* { $limit: 6 },
* ]);
* monthlyApplications = monthlyApplications
* .map((item) => {
* const {
* \_id: { year, month },
* count,
* } = item;
* const date = day()
* .month(month - 1)
* .year(year)
* .format("MMM YY");
* return { date, count };
* })
* .reverse();
* res.status(StatusCodes.OK).json({ defaultStats, monthlyApplications });
* };
* Thunder Client
* Show Stats
* Send
* Response
* {
* "defaultStats": {
* "pending": 35,
* "interview": 29,
* "declined": 36
* },
* "monthlyApplications": [
* {
* "date": "May 24",
* "count": 5
* },
* {
* "date": "Jun 24",
* "count": 10
* },
* {
* "date": "Jul 24",
* "count": 10
* },
* {
* "date": "Aug 24",
* "count": 8
* },
* {
* "date": "Sep 24",
* "count": 8
* },
* {
* "date": "Oct 24",
* "count": 7
* }
* ]
* }

**Stats Page Setup**

* Front-end part
* Create four components
* StatsContainer, and ChartsContainer (import/export)
* AreaChart, BarChart (local)
* Client – src – components
* Create following components
* AreaChart.jsx
* Rafce
* BarChart.jsx
* Rafce
* StatsContainer
* Rafce
* ChartsContainer.jsx
* Rafce
* Components – index.js
* Export (the last two)
* export { default as SubmitBtn } from "./SubmitBtn";
* export { default as ChartsContainer } from "./ChartsContainer";
* export { default as StatsContainer } from "./StatsContainer";
* client – src – pages – Stats.js
* App.jsx
* Import the loader (bottom one)
* import { action as profileAction } from "./pages/Profile";
* import {loader as statsLoader} from "./pages/Stats"
* insert it to the stats route
* action: addJobAction,
* },
* {
* path: "stats",
* element: <Stats />,
* loader: statsLoader,
* },
* Stats.js
* Complete setup
* import { ChartsContainer, StatsContainer } from "../components";
* import customFetch from "../utils/customFetch";
* import { useLoaderData } from "react-router-dom";
* export const loader = async () => {
* try {
* const response = await customFetch.get("/jobs/stats");
* return response.data;
* } catch (error) {
* return error;
* }
* };
* const Stats = () => {
* const { defaultStats, monthlyApplications } = useLoaderData();
* return (
* <>
* <StatsContainer defaultStats={defaultStats} />
* {monthlyApplications?.length > 0 && (
* <ChartsContainer data={monthlyApplications} />
* )}
* </>
* );
* };
* export default Stats;
* Browser
* Stats tab
* Shows
* StatsContainer and ChartsContainer

**Stats Controller**

* StatsContainer.jsx
* The complete setup (code)
* import { FaSuitcaseRolling, FaCalendarCheck, FaBug } from "react-icons/fa";
* import Wrapper from "../assets/wrappers/StatsContainer";
* import StatItem from "./StatItem";
* const StatsContainer = ({ defaultStats }) => {
* const stats = [
* {
* title: "pending applications",
* count: defaultStats?.pending || 0,
* icon: <FaSuitcaseRolling />,
* color: "#f59e0b",
* bcg: "#fef3c7",
* },
* {
* title: "interviews scheduled",
* count: defaultStats?.interview || 0,
* icon: <FaCalendarCheck />,
* color: "#647acb",
* bcg: "#e0e8f9",
* },
* {
* title: "jobs declined",
* count: defaultStats?.declined || 0,
* icon: <FaBug />,
* color: "#d66a6a",
* bcg: "#ffeeee",
* },
* ];
* return (
* <Wrapper>
* {stats.map((item) => {
* return <StatItem key={item.title} {...item} />;
* })}
* </Wrapper>
* );
* };
* export default StatsContainer;
* Browser
* Check stats tab

**BarChart Container**

* ChatsContainer.jsx
* Complete code
* import { useState } from "react";
* import BarChart from "./BarChart";
* import AreaChart from "./AreaChart";
* import Wrapper from "../assets/wrappers/ChartsContainer";
* const ChartsContainer = ({ data }) => {
* const [barChart, setBarChart] = useState(true);
* return (
* <Wrapper>
* <h4>Monthly Applications</h4>
* <button type="button" onClick={() => setBarChart(!barChart)}>
* {barChart ? "Area Chart" : "Bar Chart"}
* </button>
* {barChart ? <BarChart data={data} /> : <AreaChart data={data} />}
* </Wrapper>
* );
* };
* export default ChartsContainer;
* Browser
* Stats tab

**Charts**

* [recharts](https://recharts.org/en-US/)
* Package
* npm i recharts@2.5.0
* client – src – components – AreaChart.jsx
* AreaChart.jsx
* Full code
* import {
* ResponsiveContainer,
* AreaChart,
* Area,
* XAxis,
* YAxis,
* CartesianGrid,
* Tooltip,
* } from "recharts";
* const AreaChartComponent = ({ data }) => {
* return (
* <ResponsiveContainer width="100%" height={300}>
* <AreaChart data={data} margin={{ top: 50 }}>
* <CartesianGrid strokeDasharray="3 3" />
* <XAxis dataKey="date" />
* <YAxis allowDecimals={false} />
* <Tooltip />
* <Area type="monotone" dataKey="count" stroke="#2cb1bc" fill="#bef8fd" />
* </AreaChart>
* </ResponsiveContainer>
* );
* };
* export default AreaChartComponent;
* BarChart.jsx
* import {
* BarChart,
* Bar,
* XAxis,
* YAxis,
* CartesianGrid,
* Tooltip,
* ResponsiveContainer,
* } from "recharts";
* const BarChartComponent = ({ data }) => {
* return (
* <ResponsiveContainer width="100%" height={300}>
* <BarChart data={data} margin={{ top: 50 }}>
* <CartesianGrid strokeDasharray="3 3 " />
* <XAxis dataKey="date" />
* <YAxis allowDecimals={false} />
* <Tooltip />
* <Bar dataKey="count" fill="#2cb1bc" barSize={75} />
* </BarChart>
* </ResponsiveContainer>
* );
* };
* export default BarChartComponent;
* Browser
* Refresh
* Toggle between area and bar charts

**Section 16: All Jobs Page**

**Query Params**

* Query parameters, also known as query strings or URL parameters, are used to pass information to a web server through the URL of a webpage. They are typically appended to the end of a URL after a question mark (?) and separated by ampersands (&). Query parameters consist of a key-value pair, where the key represents the parameter name and the value represents the corresponding data being passed. They are commonly used in web applications to provide additional context or parameters for server-side processing or to filter and sort data
* Thunder Client
* Get All Jobs
* Send
* Response
* "jobs": [
* {
* "\_id": "67064e636660d44f020b2322",
* "company": "Oyondu",
* "position": "Tax Accountant",
* "jobStatus": "pending",
* "jobType": "part-time",
* "jobLocation": "Cotton Ground",
* "createdBy": "67026ad86291a25a4bbcd125",
* "createdAt": "2024-04-21T13:36:17.000Z",
* "updatedAt": "2024-04-21T13:36:17.000Z",
* "\_\_v": 0
* },
* And more
* Query
* Query Parameters
* Search ‘Recruiter’
* {{URL}}/jobs?search=Recruiter
* jobControllers.js
* export const getAllJobs = async (req, res) => {
* console.log(req.query);
* Thunder Client
* Get All Jobs
* Send
* Check Terminal
* Update getAllJobs
* export const getAllJobs = async (req, res) => {
* console.log(req.query);
* const jobs = await Job.find({
* createdBy: req.user.userId,
* position: req.query.search,
* });
* res.status(StatusCodes.OK).json({ jobs });
* };

**Search Param**

* Our search criteria have many issues
* Have search criteria based on a condition
* jobControllers.js
* export const getAllJobs = async (req, res) => {
* const { search } = req.query;
* const queryObject = {
* createdBy: req.user.userId,
* };
* if (search) {
* queryObject.position = req.query.search;
* }
* const jobs = await Job.find(queryObject);
* res.status(StatusCodes.OK).json({ jobs });
* };
* Thunder Client
* Get All Jobs
* Query Parameters – nothing
* Send
* We get all the jobs
* Work on the exact match
* Change the if block of the getAllJobs controller
* if (search) {
* queryObject.$or = [
* { position: { $regex: search, $options: "i" } },
* { company: { $regex: search, $options: "i" } },
* ];
* }

**Job Status and Job Type**

* Updata
* export const getAllJobs = async (req, res) => {
* const { search, jobStatus } = req.query;
* const queryObject = {
* createdBy: req.user.userId,
* JobStatus,
* };
* jobStatus and jobType
* export const getAllJobs = async (req, res) => {
* const { search, jobStatus, jobType } = req.query;
* const queryObject = {
* createdBy: req.user.userId,
* };
* if (search) {
* queryObject.$or = [
* { position: { $regex: search, $options: "i" } },
* { company: { $regex: search, $options: "i" } },
* ];
* }
* if (jobStatus && jobStatus !== "all") {
* queryObject.jobStatus = jobStatus;
* }
* if (jobType && jobType !== "all") {
* queryObject.jobType = jobType;
* }
* const jobs = await Job.find(queryObject);
* res.status(StatusCodes.OK).json({ jobs });
* };

**Sorting**

* Still on jobControllers.jsx
* Update getAllJobs (the top one) – descending created date
* const jobs = await Job.find(queryObject).sort("-createdAt");
* res.status(StatusCodes.OK).json({ jobs });
* Thunder client
* Send
* Job in descending order
* Remove – sign for ascending order
* Ascending order for position
* const jobs = await Job.find(queryObject).sort("position");
* -position will be in descending order
* Making the sorting dynamic (update getAllJobs)
* const { search, jobStatus, jobType, sort } = req.query;
* //added sort
* if (jobType && jobType !== "all") {
* queryObject.jobType = jobType;
* }
* const sortOptions = { from
* newest: "-createdAt",
* oldest: "createdAt",
* "a-z": "position",
* "z-a": "-position",
* };
* const sortKey = sortOptions[sort] || sortOptions.newest;
* const jobs = await Job.find(queryObject).sort(sortKey); till
* res.status(StatusCodes.OK).json({ jobs });
* Thunder Client
* Query Parameters
* Sort newest
* In the front end the sortOptions are sorted on the utils – constants.js

**Pagination**

* Update the getAllJobs (both lines)
* const totalJobs = await Job.countDocuments(queryObject);
* res.status(StatusCodes.OK).json({ totalJobs, jobs });
* };
* Thunder Client
* Get All Jobs
* Send
* Total number of jobs displayed
* Setting limit
* const jobs = await Job.find(queryObject).sort(sortKey).limit(1);
* skip one job on the Response (Thunder Client)
* const jobs = await Job.find(queryObject).sort(sortKey).skip(1).limit(1);
* for now leave them empty
* const jobs = await Job.find(queryObject).sort(sortKey).skip().limit();
* set it up dynamically
* update getAllJobs
* const page = Number(req.query.page) || 1;
* const limit = Number(req.query.limit) || 10; //10 Jobs per page
* const skip = (page - 1) \* limit;
* const jobs = await Job.find(queryObject)
* .sort(sortKey)
* .skip(skip)
* .limit(limit);
* const totalJobs = await Job.countDocuments(queryObject);
* const numOfPages = Math.ceil(totalJobs / limit);
* res
* .status(StatusCodes.OK)
* .json({ totalJobs, numOfPages, currentPage: page, jobs })
* };
* Thunder Client
* Send
* {
* "totalJobs": 100,
* "numOfPages": 10,
* "currentPage": 1,
* "jobs": [
* {

**Search Form Setup**

* Working on front end
* SearchContainer.jsx
* Import the libraries
* import { FormRow, FormRowSelect, SubmitBtn } from ".";
* import Wrapper from "../assets/wrappers/DashboardFormPage";
* import { Form, useSubmit, Link } from "react-router-dom";
* import { JOB\_TYPE, JOB\_STATUS, JOB\_SORT\_BY } from "../../../utils/constants";
* import { useAllJobsContext } from "../pages/AllJobs";
* const SearchContainer = () => {
* complete code
* const SearchContainer = () => {
* return (
* <Wrapper>
* <Form className="form">
* <h5 className="form-title">search form</h5>
* <div className="form-center">
* {/\* search position \*/}
* <FormRow type="search" name="search" defaultValue="a" />
* <FormRowSelect
* labelText="job status"
* name="jobStatus"
* list={["all", ...Object.values(JOB\_STATUS)]}
* defaultValue="all"
* />
* <FormRowSelect
* labelText="job type"
* name="jobType"
* list={["all", ...Object.values(JOB\_TYPE)]}
* defaultValue="all"
* />
* <FormRowSelect
* name="sort"
* defaultValue="newest"
* list={[...Object.values(JOB\_SORT\_BY)]}
* />
* <Link to="/dashboard/all-jobs" className="btn form-btn delete-btn">
* Reset Search Values
* </Link>
* {/\* TEMP!!!! \*/}
* <SubmitBtn formBtn />
* </div>
* </Form>
* </Wrapper>
* );
* };
* export default SearchContainer;

**Loader and Query Params**

* accessing the values from the database
* client – src – pages – AllJobs.jsx
* update loader
* export const loader = async ({ request }) => {
* const params = Object.fromEntries([
* ...new URL(request.url).searchParams.entries(),
* ]);
* try {
* const { data } = await customFetch.get("/jobs", {
* params,
* });
* return {
* data,
* searchValues: { ...params },
* };
* } catch (error) {
* toast.error(error.response.data.msg);
* return error;
* }
* };
* Browser
* All Jobs
* Select pending (leave the rest as they are)
* Submit

**Controller Inputs**

* Remove temporary submit button
* Trigger the form submit every time we change something in the input field
* Add onchange function
* Components – FormRow.jsx
* Add onChange
* const FormRow = ({ type, name, labelText, defaultValue, onChange }) => {
* add on below className
* className="form-input"
* onChange={onChange}
* required
* />
* </div>
* FormRowSelect.jsx
* Add onChange
* const FormRowSelect = ({ name, labelText, list, defaultValue = "", onChange }) => {
* add on the component
* className="form-select"
* defaultValue={defaultValue}
* onChange={onChange}
* Add onChange on the FormRow components
* <FormRow
* type="search"
* name="search"
* defaultValue="a"
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* <FormRowSelect
* labelText="job status"
* name="jobStatus"
* list={["all", ...Object.values(JOB\_STATUS)]}
* defaultValue="all"
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* <FormRowSelect
* labelText="job type"
* name="jobType"
* list={["all", ...Object.values(JOB\_TYPE)]}
* defaultValue="all"
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* <FormRowSelect
* name="sort"
* defaultValue="newest"
* list={[...Object.values(JOB\_SORT\_BY)]}
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* Browser
* Make changes on search and result come before clicking the button
* AllJobs.jsx
* Update the AllJobs function
* const AllJobs = () => {
* const { data, searchValues } = useLoaderData();
* return (
* <AllJobsContext.Provider value={{ data, searchValues }}>
* <SearchContainer />
* <JobsContainer />
* </AllJobsContext.Provider>
* );
* };
* SearchContainer.jsx
* Update the SearchContainer Component
* const SearchContainer = () => {
* const { searchValues } = useAllJobsContext();
* const { search, jobStatus, jobType, sort } = searchValues;
* const submit = useSubmit();
* return (
* <Wrapper>
* <Form className="form">
* <h5 className="form-title">search form</h5>
* <div className="form-center">
* {/\* search position \*/}
* <FormRow
* type="search"
* name="search"
* defaultValue={search}
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* <FormRowSelect
* labelText="job status"
* name="jobStatus"
* list={["all", ...Object.values(JOB\_STATUS)]}
* defaultValue={jobStatus}
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* <FormRowSelect
* labelText="job type"
* name="jobType"
* defaultValue={jobType}
* list={["all", ...Object.values(JOB\_TYPE)]}
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* <FormRowSelect
* name="sort"
* defaultValue={sort}
* list={[...Object.values(JOB\_SORT\_BY)]}
* onChange={(e) => {
* submit(e.currentTarget.form);
* }}
* />
* <Link to="/dashboard/all-jobs" className="btn form-btn delete-btn">
* Reset Search Values
* </Link>
* </div>
* </Form>
* </Wrapper>
* );
* };

Debounce

* Create some delay while the user is typing the query (not to search instantly)
* Limit the number of request
* [JS Nuggets - Debounce](https://youtu.be/tYx6pXdvt1s)
* In JavaScript, debounce is a way to limit how often a function gets called. It helps prevent rapid or repeated function executions by introducing a delay. This is useful for task like handling user input, where you want to wait for a pause before triggering an action to avoid unnecessary processing
* SearchContainer.jsx (update the debounce and onChange method)
* const SearchContainer = () => {
* const { searchValues } = useAllJobsContext();
* const { search, jobStatus, jobType, sort } = searchValues;
* const submit = useSubmit();
* const debounce = (onChange) => {
* let timeout;
* return (e) => {
* const form = e.currentTarget.form;
* clearTimeout(timeout);
* timeout = setTimeout(() => {
* onChange(form);
* }, 2000);
* };
* };
* return (
* <Wrapper>
* <Form className="form">
* <h5 className="form-title">search form</h5>
* <div className="form-center">
* {/\* search position \*/}
* <FormRow
* type="search"
* name="search"
* defaultValue={search}
* onChange={debounce((form) => {
* submit(form);
* })}
* />
* <FormRowSelect
* labelText="job status"
* name="jobStatus"
* list={["all", ...Object.values(JOB\_STATUS)]}
* defaultValue={jobStatus}
* onChange={debounce((form) => {
* submit(form);
* })}
* />
* <FormRowSelect
* labelText="job type"
* name="jobType"
* defaultValue={jobType}
* list={["all", ...Object.values(JOB\_TYPE)]}
* onChange={debounce((form) => {
* submit(form);
* })}
* />
* <FormRowSelect
* name="sort"
* defaultValue={sort}
* list={[...Object.values(JOB\_SORT\_BY)]}
* onChange={debounce((form) => {
* submit(form);
* })}
* />
* <Link to="/dashboard/all-jobs" className="btn form-btn delete-btn">
* Reset Search Values
* </Link>
* </div>
* </Form>
* </Wrapper>
* );
* };

**Pagination Setup**

* Will adopt two approaches
* Create a new component
* Client – src – component – PageBtnContainer.jsx
* PageBtnContainer.jsx
* Rafce
* const PageBtnContainer = () => {
* return <div>PageBtnContainer</div>;
* };
* export default PageBtnContainer;
* jobsContainer.jsx
* update the JobsContainer
* const { jobs, totalJobs, numOfPages } = data;
* add h5 after the wrapper
* <Wrapper>
* <h5>
* {totalJobs} job{jobs.length > 1 && "s"} found
* </h5>
* <div className="jobs">
* Import the pagebtncontainer
* import { useAllJobsContext } from "../pages/AllJobs";
* import PageBtnContainer from "./PageBtnContainer";
* add it before the closing wrapper
* </div>
* {numOfPages > 1 && <PageBtnContainer/>}
* </Wrapper>
* PageBtnContainer.jsx
* Import libraries
* import { HiChevronDoubleLeft, HiChevronDoubleRight } from "react-icons/hi";
* import Wrapper from "../assets/wrappers/PageBtnContainer";
* import { useLocation, Link, useNavigate } from "react-router-dom";
* import { useAllJobsContext } from "../pages/AllJobs";

**Render Buttons**

* PageBtnContainer.jsx
* Update PageBtnContainer function
* const PageBtnContainer = () => {
* const {
* data: { numOfPages, currentPage },
* } = useAllJobsContext();
* const { search, pathname } = useLocation();
* const navigate = useNavigate();
* const pages = Array.from({ length: numOfPages }, (\_, index) => index + 1);
* const handlePageChange = (pageNumber) => {
* const searchParams = new URLSearchParams(search);
* searchParams.set("page", pageNumber);
* navigate(`${pathname}?${searchParams.toString()}`);
* };
* return (
* <Wrapper>
* <button
* className="btn prev-btn"
* onClick={() => {
* let prevPage = currentPage - 1;
* if (prevPage < 1) prevPage = numOfPages;
* handlePageChange(prevPage);
* }}
* >
* <HiChevronDoubleLeft />
* prev
* </button>
* <div className="btn-container">
* {pages.map((pageNumber) => (
* <button
* className={`btn page-btn ${pageNumber === currentPage && "active"}`}
* key={pageNumber}
* onClick={() => handlePageChange(pageNumber)}
* >
* {pageNumber}
* </button>
* ))}
* </div>
* <button
* className="btn next-btn"
* onClick={() => {
* let nextPage = currentPage + 1;
* if (nextPage > numOfPages) nextPage = 1;
* handlePageChange(nextPage);
* }}
* >
* next
* <HiChevronDoubleRight />
* </button>
* </Wrapper>
* );
* };

**Pagination Logic – First Approach**

* See the updated PageBtnContainer function above

**Pagination Logic – Complex Approach**

* PageBtnContainer.jsx
* Update the PageBtnContainer function
* const PageBtnContainer = () => {
* const {
* data: { numOfPages, currentPage },
* } = useAllJobsContext();
* const { search, pathname } = useLocation();
* const navigate = useNavigate();
* const handlePageChange = (pageNumber) => {
* const searchParams = new URLSearchParams(search);
* searchParams.set("page", pageNumber);
* navigate(`${pathname}?${searchParams.toString()}`);
* };
* const addPageButton = ({ pageNumber, activeClass }) => {
* return (
* <button
* className={`btn page-btn ${activeClass && "active"}`}
* key={pageNumber}
* onClick={() => handlePageChange(pageNumber)}
* >
* {pageNumber}
* </button>
* );
* };
* const renderPageButtons = () => {
* const pageButtons = [];
* // Add the first page button
* pageButtons.push(
* addPageButton({ pageNumber: 1, activeClass: currentPage === 1 })
* );
* // Add the dots before the current page if there are more than 3 pages
* if (currentPage > 3) {
* pageButtons.push(
* <span className="page-btn dots" key="dots-1">
* ....
* </span>
* );
* }
* // one before current page
* if (currentPage !== 1 && currentPage !== 2) {
* pageButtons.push(
* addPageButton({ pageNumber: currentPage - 1, activeClass: false })
* );
* }
* // Add the current page button
* if (currentPage !== 1 && currentPage !== numOfPages) {
* pageButtons.push(
* addPageButton({ pageNumber: currentPage, activeClass: true })
* );
* }
* // one after current page
* if (currentPage !== numOfPages && currentPage !== numOfPages - 1) {
* pageButtons.push(
* addPageButton({ pageNumber: currentPage + 1, activeClass: false })
* );
* }
* if (currentPage < numOfPages - 2) {
* pageButtons.push(
* <span className=" page-btn dots" key="dots+1">
* ....
* </span>
* );
* }
* // Add the last page button
* pageButtons.push(
* addPageButton({
* pageNumber: numOfPages,
* activeClass: currentPage === numOfPages,
* })
* );
* return pageButtons;
* };
* return (
* <Wrapper>
* <button
* className="prev-btn"
* onClick={() => {
* let prevPage = currentPage - 1;
* if (prevPage < 1) prevPage = numOfPages;
* handlePageChange(prevPage);
* }}
* >
* <HiChevronDoubleLeft />
* prev
* </button>
* <div className="btn-container">{renderPageButtons()}</div>
* <button
* className="btn next-btn"
* onClick={() => {
* let nextPage = currentPage + 1;
* if (nextPage > numOfPages) nextPage = 1;
* handlePageChange(nextPage);
* }}
* >
* next
* <HiChevronDoubleRight />
* </button>
* </Wrapper>
* );
* };

**Section 17: Initial Deploy**

**Local Build**

* Remove default values from inputs in Register and Login
* Navigate to client and build front-end
* cd client && npm run build
* copy/paste all the files/folders to the public in the server
* copy files from client – dist to jobify - public
* from client/dist
* in server.js point to index.html
* put underneath
* app.use("/api/v1/auth", authRouter);
* below the above code
* app.get("\*", (req, res) => {
* res.sendFile(path.resolve(\_\_dirname, "./public", "index.html"));
* });
* Cd to jobify (root)
* On terminal
* node server
* server running on PORT 5100....
* on the browser
* <http://localhost:5100/>
* Shows that we are running on local host

**Render**

* [Render](https://render.com/)
* Login with github
* On github
* New
* Copy **…or push an existing repository from the command line**
* On VSCode terminal
* rm -rf .git
* git init