Project 4 –Blogsite

Alright now then.. It's time to write some documentation. I have no absolute clue on how to write proper documentation but this is a rather large project and I thought of writing up something for it.

This is a blogsite. Basically I wanted to make a website where you can be authenticated and post your own personal blogs. Currently anyone can post anything. This is of course something I want to change for obvious reasons.

Now one of the main features in this project is to post any pictures the users want to. This required me to well learn about storage and them image compression as well as the optimal amount of pixels for a website.

Width – 2500 px

As for compression webP exists so a major function of the backend is to convert the image to webp. This is done by using multer and sharp. I will include the code as I go along.

Technologies used

I will describe all of them in detail.

Frontend

- React.
- React-router-dom – It's used to increase the amount of pages.
- Vite – Essentially CRA but faster.
- Tailwind – CSS library.

- Firebase – I use firebase for authorization by google and to get images for the articles from firebase storage.
- ES Lint – Vite doesn't exactly have the OG error handling as far I know and therefore I added this.
- Framer-motion – I use framer-motion for animations. Majorly for the page transition animation.
- React-icons – To add graphics.
- PostCSS – vite requires this? <need more information>

Backend

The backend is essentially an API to save data into a database (Mongodb atlas). Additionally the images uploaded are converted to webP as it gives high quality and small file sizes.

(seriously it compresses them really well. 90% of the file size sometimes) Point to note - webP however is only readable on the browser.

The image Is then uploaded to firebase storage. The pixel size of the image is reduced. Two images are uploaded to the firebase, one is the quality version and the other is the thumbnail

- Node js
- Express js
- Cors – This allows the site to receive information of API. I have to figure this out properly. Currently it is open to all kinda thing.

- Nodemon – automatically refreshes the server with each change.

- Dotenv – To save important stuff (passwords and whatnot)

- Firebase admin – let's me interact with firebase to send the images received

- Multer – It is essentially used to save the images in memory and access buffer data (note… I don't know what buffer data exactly is. I know it's the image basically without the name or any other information but that's it)

- Mongoose – the database used is mongodb and mongoose is used to interact with it. Makes things a lot easier.

- Sharp – It's used to essentially modify the image. In this it is used to reduce the pixel with and change the format to webp.

I'll start with the backend because well the frontend is relatively larger and a lot of stuff's being added/removed.

Backend

App.js

The major part of the code is contained within app.js.

```
const express = require("express");

const app = express();
const multer = require("multer");
const upload = multer({ storage: multer.memoryStorage() });
const cors = require("cors");
const notFound = require("./middleware/not-found");
const connectdb = require("./db/connect");
const blogs = require("./routes/blog");
const sharp = require("sharp");
require("dotenv").config();
```

First all the required libraries are imported.

- Express is the one of the basic required things well…for everything.

- Multer – It's a middleware for handling multipart/ form-data. Basically for the handling the process of uploaded data.

  By default forms submit enctype attribute in application/www-form-urlencoded and this supports alphanumerical data.
  Multipart/form-data enctype handles files.
  You can directly specify the enctype in the form or by using the formData API.

```
let formData = new FormData();

    formData.append("file", file);
    formData.append("userName", userName);
```

Multer only processes multi-part data.
It attaches the values in req.body to objects like req.file or req.files which holds information about em.

In the const upload we essentially tell multer to hold the file in memory instead of saving the files at a destination (you can save files by specifying a destination saying multer({des: “uploads/”}).This stores them in memory as buffer objects.

```
app.post("/api/v1/upload", upload.single("file"), (req, res) => {
```

If it’s a single file – upload.single(‘file’)
If there are multiple files – upload.array(‘files’)

In the brackets you can specify the naming of it however it has to match with the name submitted by the formdata on the frontend when it is appended.

Now the files will be available on req.file / req.files

Req.file.buffer - gives a buffer of the entire file.
When using memory storage the file info contains the field called buffer which contains the entire file.
(should only be used with small files)

Req.file.originalname – gives the name of the file

- CORS – cross origin resource sharing.
  It’s an HTTP header based mechanism that allows a server to indicate any origins (domain, schema or port) other than the website’s origins.

  In general the websites can only use API arriving from the same origin for security reasons. CORS headers can be changed in the backend to include the right headers.

{note – Maybe add your own origin to the server side cors header}

- Sharp – Used to convert the image to webp as well as change the pixel width of the image. Essentially this a library to aid In image editing.

```
app.post("/api/v1/upload", upload.single("file"), (req, res) => {
  sharp(req.file.buffer)
    .resize({ width: 2500 })
    .toFormat("webp")
    .toBuffer()
    .then(async (data) => {
      const fileName =
        req.file.originalname.split(".")[0] +
        " " +
        req.body.userName +
        " " +
        new Date().toJSON().slice(0, 10) +
        ".webp";
      await app.locals.bucket
        .file(`Images/${fileName}`)
        .createWriteStream()
        .end(data);
      console.log("it worked?", fileName);
      res.status(200).json();
    });
```

Sharp takes in the buffer provided by multer and resizes it to the ideal image width which is 2500 and transfers it to webp.

- Dotenv – This is essentially to keep passwords and API keys from being served to the browser.

- Express,urlencoded() and express.json() - middleware. Majorly for post requests.

  Middleware Is called between the request and sending the response.
  Express.json() – used to recognize the incoming request object as json.
  Express.urlencoded() - used to recognize the incoming object as a string or an array.

- Firebase-admin – used to basically access firebase stuff on the server side. Essentially basically an API. It allows you to use the google resources as well.

```
const serviceAccount = require("./firebase/firebase-serviceaccuntKey.json");

```

The service account contains the credentials for the account which are used to authorize the account provided by the firebase service account.

```
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  storageBucket: process.env.BUCKET_URI,
});
app.locals.bucket = admin.storage().bucket();
```

This basically initializes the firebase account in the app giving access to the storage bucket.

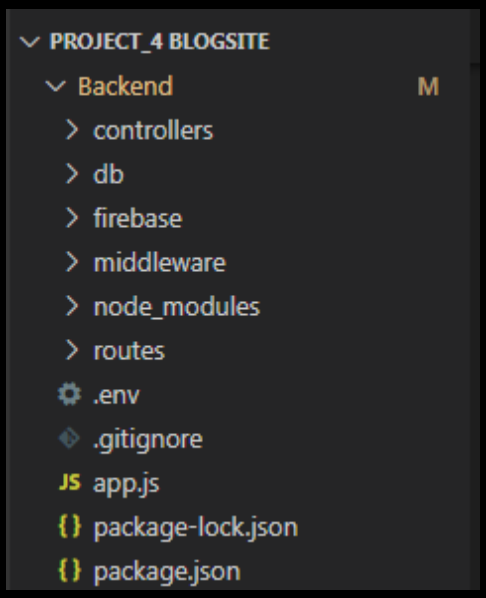
Process.env.BUCKET_URI contains the link of the storage bucket which you can find using the firebase site.

App.locals is basically this thing that lets you variables throughout the app. Here the admin.storage().bucket is saved in app.locals.bucket.

```
app.post("/api/v1/upload", upload.single("file"), (req, res) => {
  sharp(req.file.buffer)
    .resize({ width: 2500 })
    .toFormat("webp")
    .toBuffer()
    .then(async (data) => {
      const fileName =
        req.file.originalname.split(".")[0] +
        " " +
        req.body.userName +
        " " +
        new Date().toJSON().slice(0, 10) +
        ".webp";
      await app.locals.bucket
        .file(`Images/${fileName}`)
        .createWriteStream()
        .end(data);
```

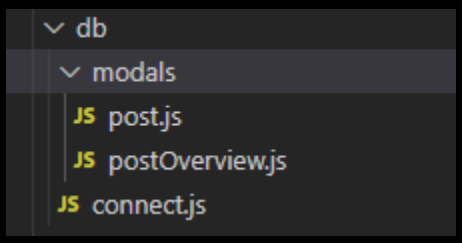
```
    console.log("it worked?", fileName);
    res.status(200).json();
  });
```

The file name is passed in to app.locals.bucket.file. we can add the image/ file to a folder by including it in the name.



File structure

- Mongoose – This essentially is a dependency to make it easier to use mongodb.



Starting with connecting it the server to the database. The connect file contains the …well connection stuff.

```
const mongoose = require("mongoose");

const connectdb = (url) => {
  return mongoose.connect(url, {
    useNewUrlParser: true,
    useCreateIndex: true,
    useFindAndModify: false,
    useUnifiedTopology: true,
  });
};

module.exports = connectdb;
```

In connect js

```
const port = process.env.PORT || 5000;
const start = async () => {
  try {
    await connectdb(process.env.MONGO_URI);
    app.listen(port, console.log("It's working boss"));
  } catch (error) {
    console.log(error);
  }
};
```

In app js

In connect js the objects present in the mongoose.connect are not required in the newer versions as this is for an older version.

In app js the url is passed on when the server is started. The URL is present in the env file. URL contains the password as well as the name of the database and username.

Modal schemas are built to make a schema allowing the database to essentially build up a template.

```
const mongoose = require("mongoose");

const postSchema = new mongoose.Schema({
  title: {
    type: String,
    required: [true, "You've gotta write something man"],
```

```
  },
  content: {
    type: String,
    required: [true, "You've gotta write something man"],
  },
  name: String,
  image: String,
  thumbnailImg: String,
  email: String,
  date: {
    type: String,
    default: () => new Date().toJSON().slice(0, 10),
  },
  comments: {
    name: String,
    comment: String,
    date: {
      type: String,
      default: () => new Date().toJSON().slice(0, 10),
    },
  },
});

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

Schema

Done using the new mongoose.schema thing. Each new field can have the objects such as the type, default value (as used by the date which saves the date when the post was submitted) and we can set limitations as well as whether a field is required which will throw an error if left empty.

When exporting we can set the collection name by passing it on to the mongoose.modal function followed by the schema

I have made two schemas. One containing the content of the blog as well as the link to the high quality image and the other containing the thumbnail image. Both of them have commonly required stuff.

The one containing the thumbnail image henceforth will be called the overview template as it is used on the mainpage to give an overview of the blog.

To link the two collections,

When a post Is submitted the mainSchema sends back it's object ID to the frontend which is then again sent to the overview template which saves it in an ID key which is set as an objectID by the schema types.

```
const mongoose = require("mongoose");

const Schema = mongoose.Schema,
  ObjectId = Schema.ObjectId;
const postOverviewSchema = new mongoose.Schema({
  title: {
    type: String,
    required: [true, "You've gotta write something man"],
  },

  thumbnailImg: String,
  name: String,
  id: ObjectId,
  date: {
    type: String,
    default: () => new Date().toJSON().slice(0, 10),
  },
});

module.exports = mongoose.model("blogthumbnail", postOverviewSchema);
```

Overview template.

CURRENT DATE –

```
default: () => new Date().toJSON().slice(0, 10),
```

Returns the date of the time the post is submitted.

MONGOOSE HANDLING CONTROLLERS

Basically the controllers js is perform CRUD actions by the server.

It takes in the modals made previously to perform actions on each one.

```
const Post = require("../db/modals/post");
const PostOverview = require("../db/modals/postOverview");
const asyncWrapper = require("../middleware/async");

const getAllBlogs = asyncWrapper(async (req, res) => {
  const blogs = await Post.find({});
  res.status(200).json({ blogs });
});

const getAllBlogoverviews = asyncWrapper(async (req, res) => {
  const blogs = await PostOverview.find({});
  res.status(200).json({ blogs });
});

const createBlog = asyncWrapper(async (req, res) => {
  const blogs = await Post.create(req.body);
  const thing = blogs._id;
  res.status(201).json({ ID: thing });
});

const createBlogoverview = asyncWrapper(async (req, res) => {
  const blogs = await PostOverview.create(req.body);
  res.status(201).json({ blogs });
});

const getBlog = asyncWrapper(async (req, res) => {
  const { id: blogID } = req.params;
  const blogs = await Post.findOne({ _id: blogID });
  if (!blogs) {
    return res.status(404).json({ msg: "nah, boss." });
  }
  res.status(200).json({ blogs });
});

const updateBlog = asyncWrapper(async (req, res) => {
  const { id: blogID } = req.params;
  const blog = await Post.findOneAndUpdate({ _id: blogID }, req.body, {
    new: true,
    runValidators: true,
  });
  if (!blog) {
    return res.status(404).json({ msg: "nah, boss. Nothing to update" });
  }
  res.status(200).json({ blog });
});

const deleteBlog = asyncWrapper(async (req, res) => {
  const { id: blogID } = req.params;
```

```
  const blog = await Post.findOneAndDelete({ _id: blogID });
  if (!blog) {
    return res.status(404).json({ msg: "nah, boss. Can't find it" });
  }
  res.status(200).json({ blog });
});

module.exports = {
  getAllBlogs,
  createBlog,
  getBlog,
  updateBlog,
  deleteBlog,
  createBlogoverview,
  getAllBlogoverviews,
};
```

The asyncWrapper is a middleware to catch errors. This is something I got from a tutorial. I'll get to more on it later on when I have more clarity on it.