MERN Deployment

Google App Engine (Standard)

This continues from the MongoDB MERN tutorial you followed.

Important mapping for this course:

Tutorial Term	Deployment Term
`client/`	**frontend**
`server/`	**backend**

Folks, if React things are not running in devContainer, please run `npm install` in `client/` first & **make sure `host: '0.0.0.0'` is set in `vite.config.js`.**

Your `vite.config.js` should look like this, when running in devcontainer:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    host: '0.0.0.0',
  }
})
```

> Yes, you can modify the dev command in `package.json` to `vite --host 0.0.0.0` as well. But this is cleaner.

📋 Table of Contents

Part 1: Prerequisites

Starting from the MongoDB MERN tutorial lab

- *If you are lazy*... The solution to which is on GitHub,
  just gotta add `config.env` file with your MongoDB URI.
- You can adjust your paths accordingly if you have a different folder structure.

Helpful pointers (in this slide deck):

Yellow Background Slides → Info

Blue Background Slides → Continuation

Green Background Slides → Refactor

1.1 Local Development Setup

Before starting deployment, make sure your MERN app works locally:

```
# Terminal 1 – Start the backend
cd mern
npm install --prefix server
node --env-file=config.env server
# In the tutorial,
# they have an odd choice of keeping config.env outside server/...
```

```
# Terminal 2 – Start the frontend
cd mern/client
npm install
npm run dev
```

✅ Test: Create, edit, and delete records at http://localhost:5173

1.2 Folder Structure (Based on Tutorial)

```
mern/
  client/   (frontend – React)
  server/   (backend – Express)
```

For deployment, we will:

1. Update React to use relative URLs (not `localhost:5050`)
2. Configure Vite proxy (so dev mode still works!)
3. Build the React `client`
4. **Copy** the build (`dist/`) into `server/`
5. Serve the build from the Express `server`
6. Deploy ONLY the `server` folder (which now contains `dist/`)

1.3 Adding Google Cloud CLI Feature

1. To install gcloud CLI in your DevContainer,
   add the following to your `devcontainer.json` (Basically extend the json file!).

```
// devcontainer.json
  "features": {
    "ghcr.io/dhoeric/features/google-cloud-cli:1": {"version": "latest"}
  }
// ... other settings ...
```

Checkout the feature repo for more details: jajera/features

1.3 (continued) Adding Google Cloud CLI Feature

2. Rebuild your DevContainer to apply changes. (Open the Command Palette and select "Dev Containers: Rebuild Container".)
3. Verify gcloud installation:

```
gcloud --version
```

At this stage, we have a working MERN app locally &
Google Cloud CLI installed in devcontainer.

Part 2: Code Changes

2.1 Update API URLs in React Components

The tutorial hardcodes `http://localhost:5050`. This won't work in production!

Open `client/src/components/Record.jsx`

Find and replace **all 3 occurrences**:

```
// BEFORE (localhost won't work on App Engine!)
"http://localhost:5050/record"

// AFTER (relative URL - works everywhere!)
"/record"
```

2.1 (continued) - Update RecordList.jsx

Open `client/src/components/RecordList.jsx`

Find and replace **all occurrences**:

```
// BEFORE
"http://localhost:5050/record"

// AFTER
"/record"
```

> **Why relative URLs?** In production, frontend and backend share the same domain. `/record` automatically uses the current domain.

2.1 (continued) - Configure Vite Proxy

But wait! React cannot send API requests to the backend which is running on port `5050` !

Solution: Configure Vite to proxy API requests to your backend.

Edit `client/vite.config.js`:

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  server: {
    host: '0.0.0.0',
    proxy: {
      '/record': 'http://localhost:5050'
    }
  }
})
```

2.2 Modify Server to Serve React Build

Edit `server/server.js` - Add this import at the top:

```
import { join } from "path";
```

2.2 (continued) - Add Static File Serving

Add this code AFTER your route imports, BEFORE `app.listen()`:

```
// Serve React build files (Node 20.11+)
app.use(express.static(join(import.meta.dirname, "dist")));

// Handle React routing - send all other requests to React
app.get("*", (req, res) => {
  res.sendFile(join(import.meta.dirname, "dist", "index.html"));
});
```

> ⚠️ **Vite uses `dist/` not `build/`.** Older Create React App used `build/`.
>
> 📢 We serve from `server/dist/` - we'll copy the build there before deploying!
>
> ⚠️ Older LLMs may suggest using `__dirname`, which doesn't work in ES Modules.

ℹ️ Info: What is `import.meta.dirname`?

(nothing to do here, just info)

In **ES Modules** (`import` / `export`), the old `__dirname` doesn't exist.

Node 20.11+ solution (what we use):

```
import.meta.dirname  // Clean and simple!
```

Older workaround (you may see this online):

```
import { dirname } from "path";
import { fileURLToPath } from "url";
const __dirname = dirname(fileURLToPath(import.meta.url));
```

> 📖 StackOverflow Explanation

At this stage, we have updated code to serve React build from Express backend.

Part 3: Build & Test Locally

3.1 Build & Copy the Client (Frontend)

From the `mern` folder:

```
cd mern/client
npm install
npm run build
```

Then copy `dist/` into `server/`:

(manually or via command line: `cp -r dist ../server/`)

This should create the following structure:

```
server/dist/
  index.html
  assets/
    *.js
    *.css
```

⚠️ **Important:** The `dist/` folder must be INSIDE `server/` for deployment!

```
# Make sure dist/ is copied to server/
ls /workspaces/mern-stack-example/mern/server/dist  # Should show index.html and assets/
```

3.2 Local Test (Important!)

Test the **production build** locally before deploying:

```
# Start the backend

cd /workspaces/mern-stack-example/mern
npm install --prefix server
node --env-file=config.env server
```

Visit http://localhost:5050 (See you dont need to run the frontend separately now!)

✅ **Test ALL functionality:**

- View records list, Create a new record, Edit a record, Delete a record

> ⚠️ If you see a blank page, make sure you ran `cp -r dist ../server/`

We now have a working production build locally!

You an use `Ctrl + C` to stop the server.

Part 4: Google Cloud Setup

4.1 Create app.yaml (in server folder)

Create `mern/server/app.yaml`:

Add `environment variables` and runtime config.

```
runtime: nodejs22
entrypoint: node server.js

env_variables:
  NODE_ENV: "production"
  ATLAS_URI: "mongodb+srv://your-mongodb-atlas-connection-string"
```

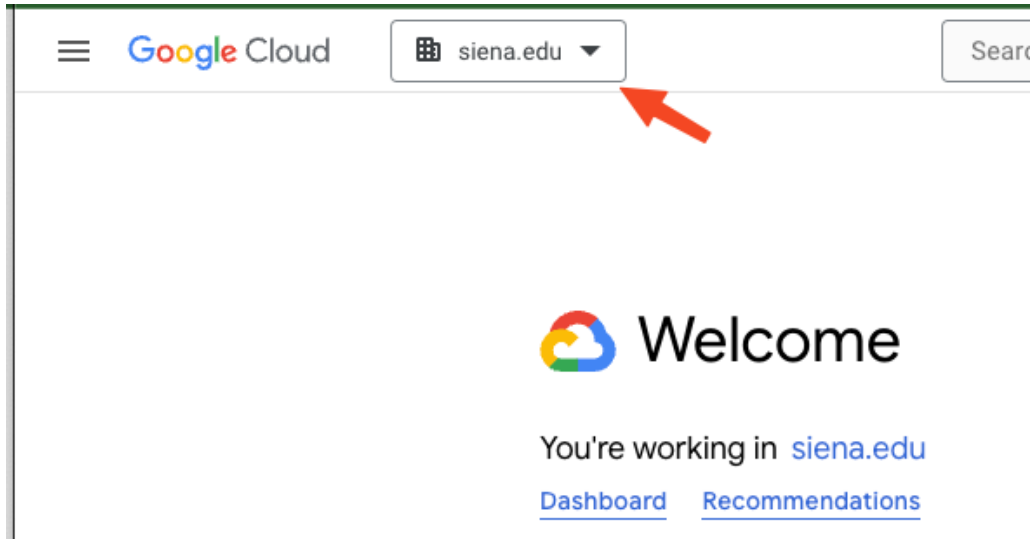
> ⚠️ Replace `your-mongodb-atlas-connection-string` with your actual MongoDB Atlas URI (from `mern/config.env`).

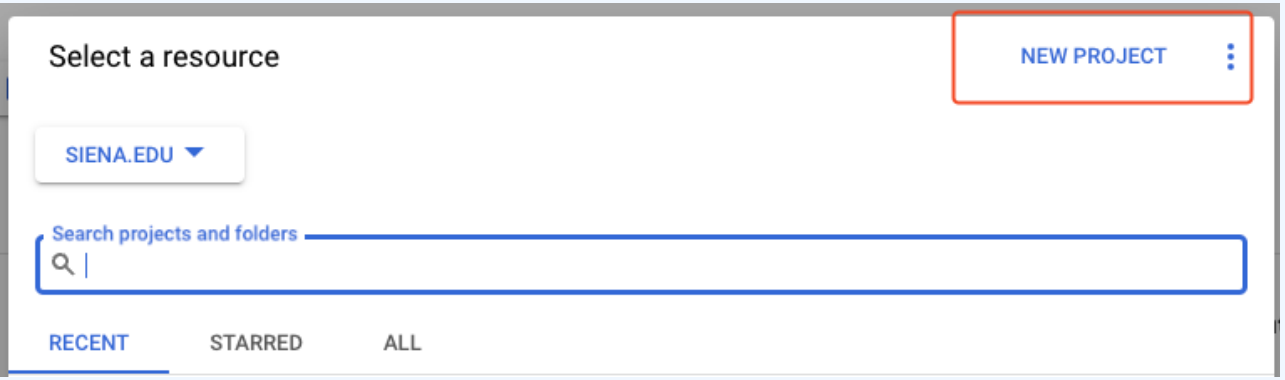
🚀 Let's Deploy to Google App Engine!

4.2 Google App Engine Setup (via UI)

Feel free to use CLI if you are pro.

1. Go to the **Google Cloud Console** and click on `Console`
2. Click on "Select project" dropdown next to the Google Cloud Logo.





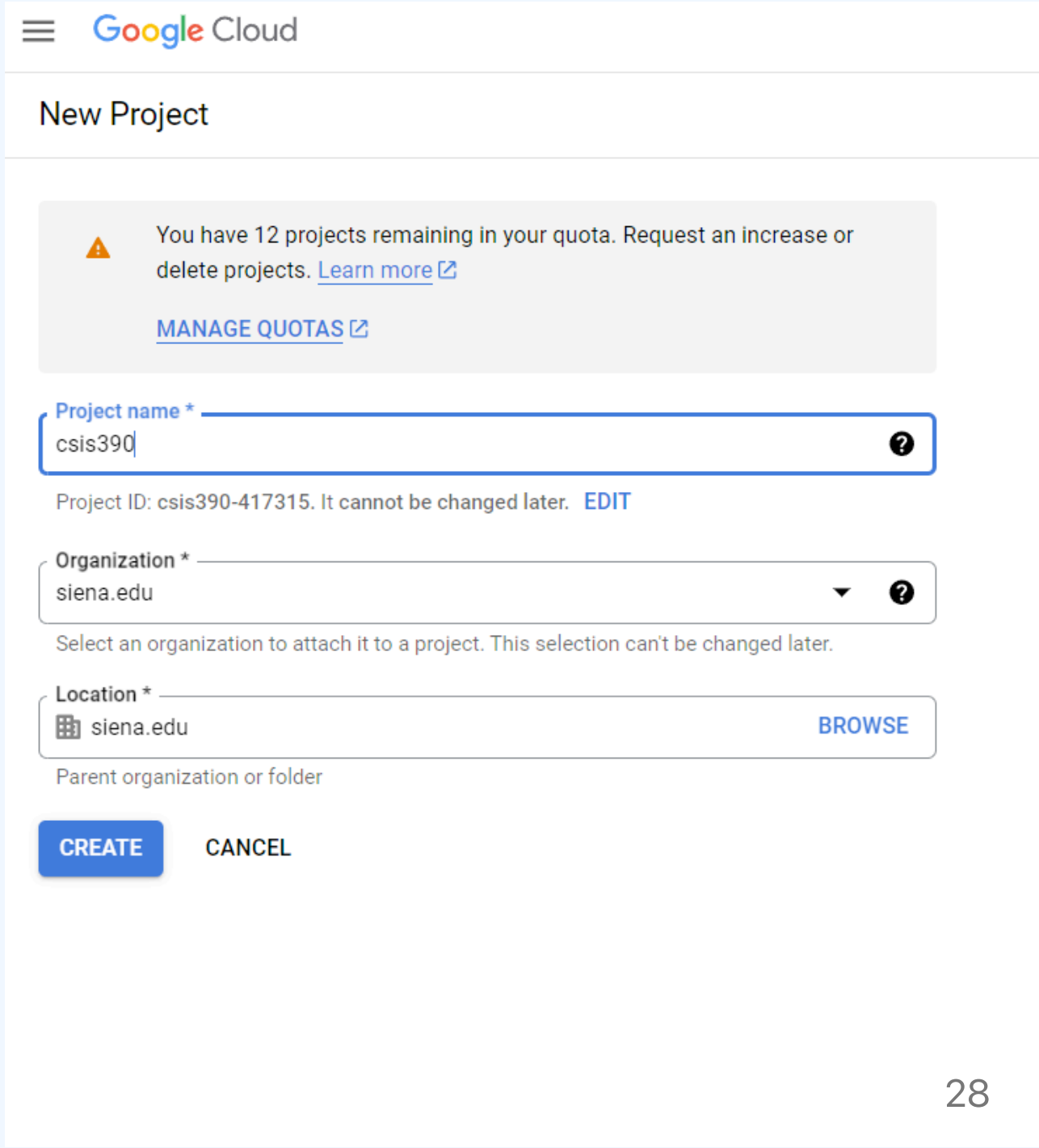
4.2 (continued) - Create New Project

3. Click on the `New Project` Button

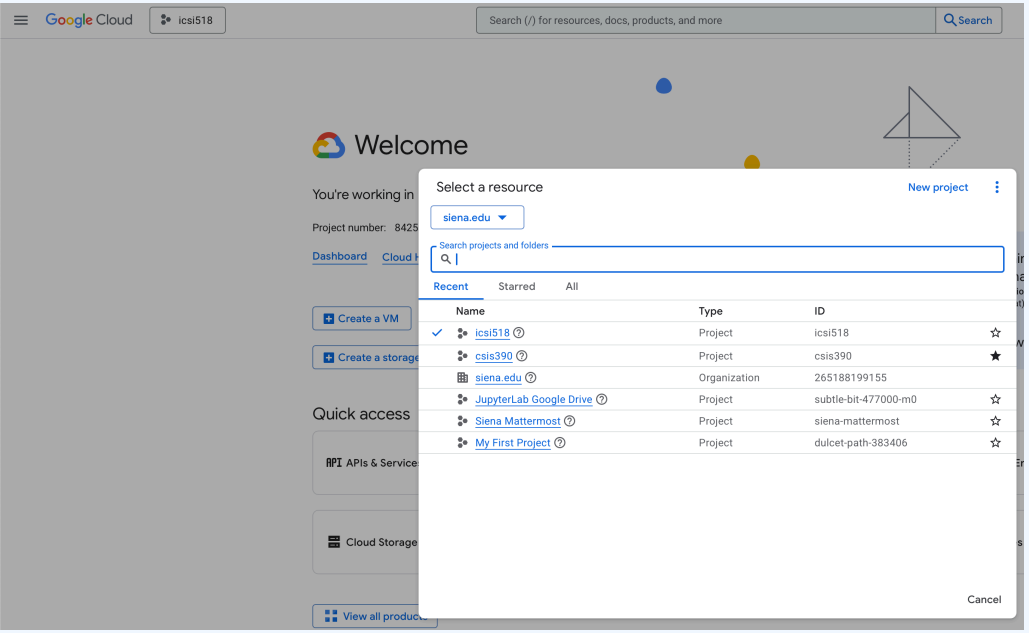
4.2 (continued) - Project Details

4. Enter the Project Name (e.g., `icsi518`) and select `Billing Account for Education`
    - Location and Organization may vary - choose any value and proceed
    - It's just a name, you can choose anything unique! 🎉

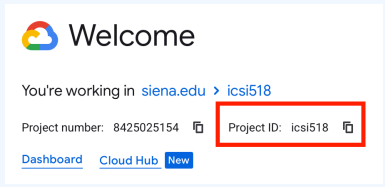
- It should take a few seconds to create the project.



4.2 (continued) - Open the Project



- Click on that project from the project selector page to open it.
- Once inside the project, note down the `Project ID` from the top bar.



> ⚠️ Please replace `PROJECT_ID` in the upcoming commands with your actual Project ID!

4.3 Authenticate gcloud CLI

Inside the `mern/server` folder

```
cd /workspaces/mern-stack-example/mern/server
```

```
gcloud auth login
gcloud config set project PROJECT_ID
```

> 📖 Please **READ** the command output. Follow the link to authenticate your account.
>
> It should give you a code to paste back into the terminal.

Example (using your project details):

```
gcloud config set project icsi518
```

At this stage, we have Google Cloud project set up & gcloud CLI authenticated.

Part 5: Deploy

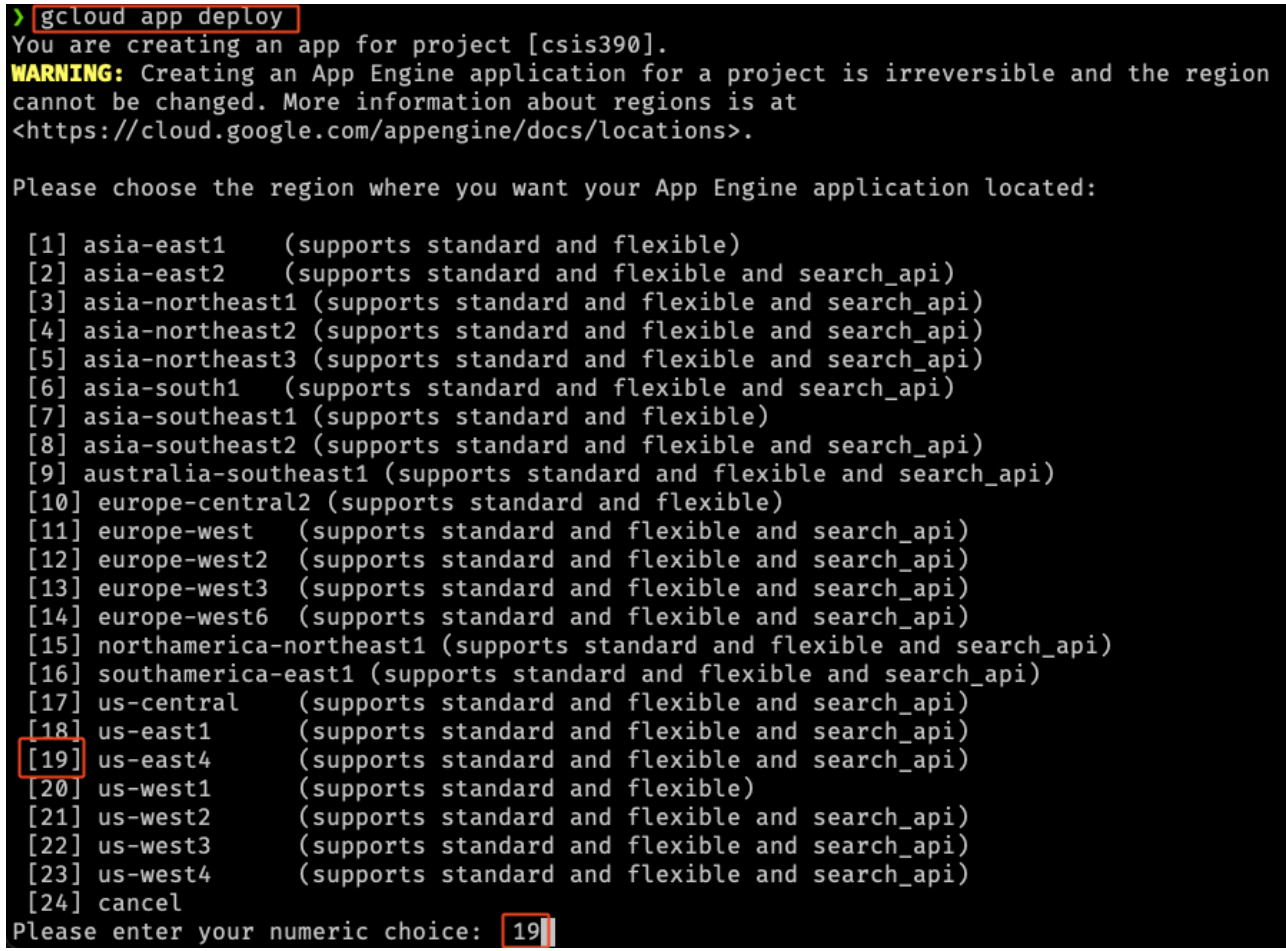
5.1 Deploy to App Engine

Inside the `mern/server` folder

```
gcloud app deploy
```

Choose "Y" for prompts.

- **It will fail the first time. Just try twice.**
- as the first time it asks for confirmation and fails to deploy...



open your app:

```
gcloud app browse
```

🎉 Your MERN app is now live!

5.2 Check Logs

If something goes wrong, check the logs:

```
gcloud app logs tail -s default
```

or open in the Cloud Console:

https://console.cloud.google.com/logs/

At this stage, we have deployed our MERN app to Google App Engine!

Part 6: Reference

6.1 How It Works in Production

Frontend and backend share the same domain → relative URLs work!

6.2 Common Issues & Fixes

Problem	Solution
`404` on page refresh	Add `app.get("*")` catch-all route
API calls fail	Update `localhost:5050` → `/record`
Missing `dist/` folder	Run `npm run build` in `client/`, then `cp -r dist ../server/`
Blank page on App Engine	Forgot to copy `dist/` to `server/` before deploying!
MongoDB errors	Check `ATLAS_URI` in `app.yaml`
Build errors	Delete `node_modules`, run `npm install`

6.3 Files Changed Summary

File	Change
`client/src/components/Record.jsx`	`localhost:5050` → relative URL
`client/src/components/RecordList.jsx`	`localhost:5050` → relative URL
`client/vite.config.js`	Add proxy config for dev mode
`server/server.js`	Add static file serving from `dist/`
`server/app.yaml`	**New file** - App Engine config
`server/dist/`	**Copied** - React build files

6.4 Deployment Checklist

- [ ] Updated all `localhost:5050` URLs to relative paths
- [ ] Added Vite proxy config for dev mode
- [ ] Modified `server.js` to serve React build from `dist/`
- [ ] Ran `npm run build` in `client/`
- [ ] **Copied `dist/` to `server/`** (`cp -r dist ../server/`)
- [ ] Tested locally at `http://localhost:5050`
- [ ] Created `app.yaml` with correct `ATLAS_URI`
- [ ] Ran `gcloud app deploy`
- [ ] Verified app works at App Engine URL

6.5 Summary

Mapping:

- `client` = frontend (React)
- `server` = backend (Express)

Key Steps:

1. Update hardcoded URLs to relative paths
2. Configure Vite proxy (dev mode still works!)
3. Build React → `client/dist/`
4. **Copy `dist/` into `server/`**
5. Test locally
6. Deploy `server/` folder to App Engine

At this stage, you have successfully deployed a MERN app to Google App Engine! Share that URL with friends! 🎉

Bonus A: Adding a New API Endpoint

This will help you go further towards the capstone project!

In your project, you will need to add new backend features...

Want to extend your app? Let's add a simple `/api/hello` endpoint!

This shows how easy it is to:

1. Add new backend functionality
2. Call it from the frontend
3. Redeploy with one command

A.1 Add API Route to Express

Edit `server/server.js` - Add this BEFORE the static file serving:

```
// Custom API endpoint
app.get("/api/hello", (req, res) => {
  res.json({
    message: "Hello from the backend!",
    timestamp: new Date().toISOString()
  });
});
```

> ⚠️ Place this **before** the `app.get("*")` catch-all route!
>
> Notice how you have `record` routes and now `api` routes.
>
> Your AI Friend can help you create more complex routes as needed.

A.2 Update Vite Proxy

Edit `client/vite.config.js` - Add the new `/api` route:

So things work locally in dev mode.

```
export default defineConfig({
  plugins: [react()],
  server: {
    proxy: {
      '/record': 'http://localhost:5050',
      '/api': 'http://localhost:5050'  // ← Add this!
    }
  }
})
```

A.3 Call API from React (Try it from any Component)

You can call your new API from any React component:

```
// Example: fetch the API
const [greeting, setGreeting] = useState("");

useEffect(() => {
  fetch("/api/hello")
    .then(res => res.json())
    .then(data => setGreeting(data.message + " @ " + data.timestamp));
}, []);

return <p>{greeting}</p>;
```

I know, useEffect is not ideal for data fetching...

We will refactor to `useSWR` in Bonus B!

A.4 Test Locally

```
# Terminal 1 - Start the backend
cd /workspaces/mern-stack-example/mern
npm install --prefix server
node --env-file=config.env server

# Terminal 2 - Frontend (dev mode)
cd /workspaces/mern-stack-example/mern/client
npm run dev
```

Test your new endpoint:

- Direct: http://localhost:5050/api/hello
- Via proxy: http://localhost:5173/api/hello

A.5 Rebuild & Redeploy

```
# Rebuild the client (if you made React changes)
cd /workspaces/mern-stack-example/mern/client
npm run build

# Copy dist to server (IMPORTANT!)
cp -r dist ../server/

# Redeploy to App Engine
cd ../server
gcloud app deploy
```

That's it! 🎉 Your updated app is live!

```
# Open your app
gcloud app browse
```

A.6 Deployment Workflow Summary

First time: Follow all the steps (1-7)

Future updates:

```
# 1. Make your changes (backend and/or frontend)
#    Things will work locally for both React and Express
# 2. If frontend changed, rebuild AND copy:
cd client && npm run build && cp -r dist ../server/

# 3. Redeploy:
cd ../server && gcloud app deploy
```

It's that simple! ✨

> 💡 Pro tip: You can create a script to automate steps 2-3!

At this stage, you have added a new API endpoint and redeployed your MERN app! 🎉

Bonus B: Refactoring to useSWR

🚫 Stop Using `useEffect` for Data Fetching!

The tutorial uses `useEffect` + `fetch` for loading data. **This is problematic!**

Issues with `useEffect` for fetching:

- ⚠️ Race conditions
- ⚠️ No caching (refetches on every mount)
- ⚠️ No loading/error states built-in
- ⚠️ Memory leaks if component unmounts
- ⚠️ No automatic revalidation

Solution: Use `useSWR` - a proper data fetching library... **We spoke about this in class!**

B.1 Install SWR

In the `client` folder:

```
cd client
npm install swr
```

> SWR = "stale-while-revalidate" - a caching strategy that returns cached data first, then fetches fresh data.

- We have a submodule on this on Brightspace. linking slides again:
  Slides on SWR
  Slides on SWR extended

B.2 Create a Fetcher Function

Create `client/src/fetcher.js`:

```
export const fetcher = (url) => fetch(url).then((res) => res.json());
```

This simple function is reused by all `useSWR` calls.

B.3 Refactor RecordList.jsx

Before (useEffect - bad ❌):

```
import { useEffect, useState } from "react";

function RecordList() {
  const [records, setRecords] = useState([]);

  useEffect(() => {
    async function getRecords() {
      const response = await fetch("/record");
      const records = await response.json();
      setRecords(records);
    }
    getRecords();
  }, []);
  // ... rest of component
}
```

B.3 (continued) - The SWR Version

After (useSWR - good ✅):

```
import useSWR from "swr";
import { fetcher } from "../fetcher";

function RecordList() {
  const { data: records, error, isLoading } = useSWR("/record", fetcher);

  if (isLoading) return <p>Loading...</p>;
  if (error) return <p>Error loading records!</p>;
  // ... rest of component (use records directly)
}
```

> 🎉 No `useState`, no `useEffect`, built-in loading & error states!

B.4 Refactor Record.jsx

`Record.jsx` also uses `useEffect` to load data when editing an existing record.

What we're changing:

1. Replace `useEffect` data fetching → `useSWR`
2. Add conditional fetch (`null` = skip for new records)
3. Keep small `useEffect` to sync data → form state (that's OK!)

B.4 (continued) - Before

Before (useEffect - bad ❌):

```
const [form, setForm] = useState({ name: "", position: "", level: "" });
const [isNew, setIsNew] = useState(true);
const params = useParams();

useEffect(() => {
  async function fetchData() {
    const id = params.id?.toString() || undefined;
    if (!id) return;
    setIsNew(false);
    const response = await fetch(`/record/${params.id}`);
    // ... error handling ...
    const record = await response.json();
    setForm(record);
  }fetchData();
}, [params.id, navigate]);
```

B.4 (continued) - Record.jsx with SWR

After (useSWR - good ✅):

```
import useSWR from "swr";import { fetcher } from "../fetcher";
export default function Record() {
  const [form, setForm] = useState({ name: "", position: "", level: "" });
  const [isNew, setIsNew] = useState(true);
  const params = useParams();const navigate = useNavigate();
  // Fetch only when editing. Pass null to skip fetch for new records.
  const { data: existingRecord, isLoading } = useSWR(
    params.id ? `/record/${params.id}` : null, // conditional fetch!
    fetcher
  );
  // Sync fetched data to form (this useEffect is OK - not fetching!)
  useEffect(() => {
    if (existingRecord) {setForm(existingRecord);setIsNew(false);}
  }, [existingRecord]);
  if (params.id && isLoading) return <p>Loading...</p>;
  // ... rest stays the same (updateForm, onSubmit, JSX)
}
```

ℹ️ Why `params.id ? url : null`?

When **creating** a new record, there's no `params.id`.

Passing `null` as the URL tells SWR: **"Don't fetch anything"**

```
// If params.id exists → fetch that record
// If params.id is undefined → pass null → skip fetch
useSWR(params.id ? `/record/${params.id}` : null, fetcher)
```

This avoids an unnecessary (and failing) network request!

> 💡 This is called **conditional fetching** - a common SWR pattern.

B.5 Why `useSWR` is Better

We went over this in the class,

But ask your AI friend!

You can have deeper conversations about it :)

B.6 Bonus: Mutate After Changes

When you create/update/delete a record, tell SWR to refresh:

```
import useSWR, { mutate } from "swr";

// After a successful POST/PUT/DELETE:
await fetch("/record", { method: "POST", body: ... });

// Revalidate the records list:
mutate("/record");
```

> This triggers a refetch and updates all components using that data!

B.7 Resources

SWR Documentation:

- https://swr.vercel.app

Why not useEffect for fetching:

- React docs on data fetching
- You Might Not Need an Effect

> 💡 The React team themselves recommend using a library like SWR, React Query, or a framework with built-in data fetching.

You can now refactor your MERN app to use SWR for data fetching!

Then Deploy the app again...

Happy Coding!