Class Activity 12: Deploy on the cloud

(SUMBIT ALL YOUR CODE ON MOODLE)

In this activity, we will deploy our full-stack application online.

Pushing your project on GitHub

1. Initialize Git in your React project

In your project root (where package.json is):

```
# Initialize git repo
git init
```

2. Set up .gitignore

If you used Create React App, it likely already has one. If not, create .gitignore file in the root:

```
touch .gitignore
```

inside the file add at least:

```
node_modules
dist
build
.env
.DS_Store
```

Stage it:

```
git add .gitignore
```

3. Stage and commit your project

```
# Stage all files
git add .

# First commit
git commit -m "Initial commit of React project"
```

If Git asks for your name/email:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

Then re-run the commit if needed.

4. Create a new GitHub repository

1. Go to GitHub.
2. Click **New** (top-left) → **New repository**.
3. Choose:
   - **Owner**
   - **Repository name** (e.g. my-react-app)
   - Visibility: **Public** or **Private**
4. **Do NOT** initialize with README, .gitignore, or license (since you already did locally).
5. Click **Create repository**.

GitHub will now show you commands like:

```
git remote add origin https://github.com/<your-username>/<repo-name>.git
git branch -M main
git push -u origin main --force
```

5. Link local repo to GitHub and push

From your project folder, run (replace with your actual URL):

```
# Add remote
git remote add origin https://github.com/<your-username>/<repo-name>.git

# Rename branch to main (if not already)
git branch -M main

# Push to GitHub
git push -u origin main  --force
```

6. Verify

- Refresh your GitHub repo page.
- You should see your React project files there.

Deploy on Netlify.com

1. Build your Vite app locally (optional but good check)

From your project root (where vite.config.ts and package.json are):

```
npm install
npm run build
```

Vite outputs to dist/ by default. That's what Netlify will serve.

---

2. Deploy via GitHub (recommended)

Since your code is on GitHub:

1. Go to Netlify → log in.
2. In the dashboard, click **"Add new site" → "Import an existing project"**.
3. Choose **GitHub** and authorize Netlify if it asks.
4. Select your repo.

Netlify will ask for build settings:

- **Branch to deploy**: main (or your branch name)
- **Build command**:

  ```
  npm run build
  ```
- **Publish directory**:

  ```
  dist
  ```

Click **Deploy site**.

Netlify will:

- Install dependencies
- Run *npm run build*
- Serve the *dist* folder

You'll get a URL like https://your-site-name.netlify.app.

Every push to that branch triggers a new deploy automatically.

---

3. Environment variables for Vite

If you have env vars (e.g., API URLs):

In code, Vite uses import.meta.env.VITE_SOMETHING.

On Netlify:

1. Go to your site → **Site settings → Build & deploy → Environment → Environment variables**.
2. Add keys like:
    - VITE_API_URL = https://your-api.com
3. Save, then **Trigger deploy → Clear cache and deploy site**.

Deploying a Spring Boot Application on Render.com Using Docker

Deploying a Spring Boot app to Render.com (a cloud platform similar to Heroku with a free tier) via Docker is straightforward. Render supports Docker natively, allowing automatic builds from your Git repository or deployment of pre-built images from Docker Hub. This guide focuses on the GitHub integration method for easier CI/CD (continuous integration/continuous deployment), but I'll note the alternative Docker Hub approach. Render's free tier spins down inactive services after 15 minutes (with ~50-second cold starts),so if you don't accept that upgrade for production.

Prerequisites:

- A Spring Boot project (Maven or Gradle) with a .jar file under the build/libs folder.
- GitHub account with your repo pushed making sure the .jar file is also pushed onto Github.
- Docker installed locally for testing (optional but recommended).
- Render account (free signup at render.com).

Step 1: Prepare Your Spring Boot Application

- Ensure your app runs on port 8080 by default (Spring Boot's default).
- To make it Render-compatible, bind to all interfaces and use an environment variable for the port. In src/main/resources/application.properties (or application.yml): server.port=${PORT:8080}

This uses Render's PORT env var (dynamic, e.g., 10000) or falls back to 8080 locally. Source: Render Community Forum.

- Build and test locally. Make sure there a .jar file under the build/libs folder. If not you must solve this problem first.
- Commit and push to GitHub. Ensure your repo is public (or private with Render access).

Step 2: Create a Dockerfile

Place a Dockerfile (no extension) in your **project's root directory**. Use a multi-stage build for efficiency: one stage compiles the JAR, the next runs it with a slim JRE image. This keeps the final image small (~200MB) and secure.

Example for Gradle:

```
FROM eclipse-temurin:17-jdk

WORKDIR /app

# Copy your prebuilt JAR from build/libs/ into the image
COPY build/libs/cardatabase-0.0.1-SNAPSHOT.jar app.jar

# Expose the port your Spring Boot app uses (default is 8080)
EXPOSE 8080

# Run the JAR
CMD ["java", "-jar", "app.jar"]
```

- **Why this?** Eclipse Temurin is a reliable OpenJDK distribution. Alpine base is lightweight. Expose uses ${PORT:-8080} for flexibility. Source: Adapted from GitHub Repo Example and DEV Community Guide.
- Test locally: docker build -t your-app . then docker run -p 8080:8080 -e PORT=8080 your-app. Access via http://localhost:8080.

Step 3: Connect Your GitHub Repo to Render

- Log in to Render Dashboard.
- Click **New > Web Service**.
- Choose **Build and deploy from a Git repository**.
- Connect your GitHub account and select your repo/branch (e.g., main).
- Configure:
    - **Name**: e.g., my-spring-app.
    - **Region**: Closest to your users (e.g., Oregon for US).
    - **Runtime**: Select **Docker** (Render auto-detects the Dockerfile).
    - **Build Command**: Leave blank (Dockerfile handles it).
    - **Start Command**: Leave blank (ENTRYPOINT in Dockerfile runs the app).
    - **Instance Type**: Free (512MB RAM, suitable for small apps).
    - **Environment Variables**: Add any needed (e.g., SPRING_PROFILES_ACTIVE=prod, database URLs). For databases, Render offers free PostgreSQL—create one via **New > PostgreSQL** and add its URL as SPRING_DATASOURCE_URL. Another option is to use neon.tech.
- Click **Create Web Service**. Render clones the repo, builds the Docker image, and deploys. Watch logs in real-time.

Step 4: Deploy and Monitor

- Deployment takes 5-10 minutes (build + push). You'll get a URL like https://my-spring-app.onrender.com.
- Test: Visit the URL or use curl https://your-app.onrender.com/your-endpoint.
- Auto-deploys: Push to your GitHub branch triggers rebuilds.
- Logs/Metrics: In Render Dashboard > Your Service > Events/Logs. Check for errors like port binding issues.
- Health Checks: Render pings / by default; add a custom path if needed (e.g., /actuator/health via env var HEALTHCHECK_PATH).

Troubleshooting Common Issues

- **Build Fails**: Ensure Dockerfile paths match (e.g., JAR name). Check logs for Maven/Gradle errors. Use -DskipTests to speed up.

- **Port Binding Error**: Confirm server.address=0.0.0.0 and PORT env var. Render requires binding to 0.0.0.0. Source: Stack Overflow.
- **JAR Corrupt**: Don't commit large JARs to Git (use .gitignore for target/ or build/). Let Docker build it. GitHub warns at 50MB+.
- **Cold Starts**: Free tier sleeps; expect delays. Upgrade to Starter ($7/mo) for always-on.
- **Database**: For PostgreSQL, use Render's managed DB and set env vars like SPRING_DATASOURCE_URL=jdbc:postgresql://host:port/db?user=....
- **HTTPS**: Render auto-provisions free SSL.

Best Practices

- **Security**: Use env vars for secrets (e.g., API keys). Scan images with tools like Trivy.
- **Scaling**: Free tier is single-instance; paid plans auto-scale.
- **CI/CD**: GitHub Actions can build/push images if needed.
- **Costs**: Free for basics; monitor usage (e.g., build minutes).

The above deployment should work with H2 database without any problems. For PostgresSQL see the next section.

Using PostgreSQL as database and hosting online

If you want to host your database using PostgresSQL, you should change the configuration in application.properties to something like this:

```
# Neon PostgreSQL connection
spring.datasource.url=jdbc:postgresql://ep-long-bar-ahngvlye-pooler.c-3.us-east-1.aws.neon.tech/neondb?sslmode=require
spring.datasource.username=neondb_owner
spring.datasource.password=npg_zWrHeKcN74ny
spring.datasource.driver-class-name=org.postgresql.Driver

# JPA/Hibernate settings
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Optional: Connection pool settings for Neon
spring.datasource.hikari.maximum-pool-size=5
spring.datasource.hikari.minimum-idle=2
```

By the way, Render.com can also host a PostgresSQL database

Investigate that. It may be easier.