

AWS Setup Guide

I am building a serverless web application called StayMate using React for frontend, Node.js + Express for backend, and PostgreSQL on RDS. My goal is to deploy this project using AWS learner lab environment. I want a step-by-step understanding of how to **configure the following AWS services** for this project, including best practices, permissions, and how they interact:

1. **Amazon S3** – For hosting the frontend and storing uploaded images.

- How to create a bucket for the frontend
- How to configure static website hosting
- Folder structure for frontend assets and images
- Bucket permissions for public access in learner lab
- Best practices to avoid deleting uploaded images during deployments

2. **AWS Lambda** – For hosting the backend serverless functions.

- How to deploy Node.js Express backend as Lambda
- How to manage multiple endpoints (GET, POST, PUT, DELETE)
- How to configure environment variables for RDS connection
- Permissions needed for Lambda to access S3 and RDS
- How to update an existing Lambda function

3. **Amazon API Gateway** – To expose Lambda functions as REST APIs.

- How to create a new API Gateway for the backend
- How to integrate Lambda functions
- How to configure CORS for React frontend
- How to deploy API and get the endpoint URL

4. **Amazon RDS (PostgreSQL)** – For relational database.

- How to create an RDS instance in learner lab
- How to configure security groups to allow Lambda

or local access

- How to connect using DBeaver and Node.js
- Best practices for public access in learner lab environment

5. **Amazon Cognito** – For authentication.

- How to create a user pool
- How to integrate with React frontend and Node.js backend
- How to restrict roles (student, hostel_owner, mess_owner)
- How to handle JWTs in backend API calls

6. **Amazon CloudFront** – For CDN and frontend distribution.

- How to create a distribution for the S3 frontend
- How to configure origin and caching policies
- How to update distribution when frontend changes

7. **AWS IAM** – To configure permissions and roles.

- What roles/policies are required for Lambda, S3, API Gateway, and RDS
- How to assign least privilege in learner lab environment
- How to handle permissions for image upload and API access

8. **Amazon CloudWatch** – For logging and monitoring.

- How to view Lambda logs
- How to create alerts for errors

Please give a **step-by-step practical guide** for each service in the order above, including any **learner lab restrictions** and **workarounds**. Include **tips for beginners** to avoid common pitfalls and maintain a working end-to-end connection between frontend, backend, and database.

Absolutely — here's a **step-by-step practical guide** to configure all AWS services you'll need to deploy your **StayMate serverless web application** using the AWS

Learner Lab environment. This will cover setup, best practices, permissions, and service interactions from **frontend to backend and database**, while pointing out common beginner pitfalls and Learner Lab restrictions/workarounds.

1) Amazon S3 — Frontend Hosting & Image Storage

Create an S3 Bucket for Frontend

1. Open the AWS Console → S3 → Create bucket

- Give a globally unique name (e.g., `staymate-frontend-12345`).
- Choose your AWS region.

2. Uncheck “Block all public access” — needed to serve a public static site in learner lab. AWS Document...

Learner Lab Tip: Some policies may restrict block public access change — if so, use **CloudFront + Origin Access Control (OAC)** instead of making bucket public.

Enable Static Website Hosting

1. Go to the bucket → Properties → Static website hosting → Enable.

2. Set:

- **Index document:** `index.html`
- **Error document:** `index.html` (for SPA routing)

3. Save. You now see a **website endpoint URL**. AWS Document...

Folder Structure (Example)

```
bash
```

```
/build
    index.html
    favicon.ico
    /static
        js/
        css/
/images
    uploads/      ← user uploaded images
```

- Keep `/images/uploads` separate from frontend assets to avoid overwriting on redeployments.

 Copy code

Bucket Permissions & Best Practices

- For static assets, grant public read via bucket policy:

```
json

{

  "Version": "2012-10-17",

  "Statement": [{

    "Effect": "Allow",
    "Principal": "*",
    "Action": "s3:GetObject",
    "Resource": "arn:aws:s3:::staymate-frontend-12345/*"
  }]
}
```

AWS Document...

 Copy code

- For user uploads (images):

- Do not make the entire bucket public — instead use **Lambda presigned URLs** for secure uploads.
- This avoids accidental public exposure and accidental overwrites/deletions.

! Avoid Overwriting Images on Deploy

 When uploading new builds, only sync static assets, **don't delete the images folder**:

bash

```
aws s3 sync ./build s3://staymate-frontend-12345 --exclude "images/uploads/*"
```

 Copy code

This preserves user-uploaded images.

2) AWS Lambda — Backend Logic

Your express backend will run as serverless functions.

Deploy Node.js & Express as Lambda (Using Serverless or AWS SAM)

You *cannot* run an Express app directly on Lambda — you wrap it with an adapter:

Option A — Serverless Framework

1. `npm install serverless serverless-http`

2. `app.js` → export handler:

js

```
const serverless = require("serverless-http");
module.exports.handler = serverless(app);
```

 Copy code

3. Create `serverless.yml` with:

yaml

```
functions:
  app:
    handler: app.handler
    events:
      - http:
          path: /
          method: ANY
      - http:
          path: "{proxy+}"
          method: ANY
```

4. `sls deploy` → Lambda + API Gateway created.

Option B — AWS SAM

 Copy code

- Write a `template.yaml`, define Lambda function + API events.

Manage Multiple HTTP Methods

API Gateway handles:

Method	Purpose
GET	Read
POST	Create
PUT	Update
DELETE	Delete

In express:

```
js

app.get("/api/hotels", ...);
app.post("/api/hotels", ...);
app.put("/api/hotels/:id", ...);
app.delete("/api/hotels/:id", ...);
```

API Gateway will proxy these to your Lambda via "ANY" + {proxy+} .

 Copy code

Env Vars for RDS Connection

In Lambda console → Configuration → Environment variables:

```
nginx
```

```
PGHOST
```

```
PGPORT
```

```
PGDATABASE
```

```
PGUSER
```

```
PGPASSWORD
```

Ensure these are encrypted using **Lambda's encryption helpers (KMS)**.

 Copy code

Lambda Permissions (IAM)

Lambda needs:

1. S3 access (for image reads/writes)

- s3:PutObject , s3:GetObject , s3:DeleteObject on uploads folder.

2. RDS access

- Uses Security Group allowing Lambda SG to talk to RDS SG.

3. CloudWatch Logs

- logs>CreateLogGroup
- logs>CreateLogStream
- logs:PutLogEvents

4. Least privilege

- Attach custom policy limiting to only required resources.

Learner Lab may require attaching an existing **LabRole** that has broad access.

Course Hero

Update Existing Lambda

Whenever backend code changes:

1. Zip the code and dependencies (or use serverless deploy).
 2. Go to Lambda console → Select function → Upload ZIP.
 3. Test with new API Gateway endpoint.
-

3) Amazon API Gateway — Make HTTP APIs

Create a New API

1. AWS Console → API Gateway → HTTP API (simpler than REST API).
 2. Connect to your Lambda function via **integrations**.
 3. Map routes:
 - `/api/hotels` → Lambda
 - `/api/images` → Lambda
-

CORS for React

Enable CORS on each route:

- `Access-Control-Allow-Origin: *` (or your frontend domain).
- `Access-Control-Allow-Methods: GET, POST, PUT, DELETE`.

This lets your React app make API calls.

Deploy & Get API URL

After configuring routes:

- Click **Deploy**
 - You'll see the API URL — use this in your React fetch calls.
-

4) Amazon RDS (PostgreSQL)

Create RDS PostgreSQL Instance

1. RDS → Create Database → PostgreSQL.
2. Choose free tier / smallest instance for learner lab.
3. Set username/password.

 **Learner Lab Restrictions:** You may not be able to assign public accessibility. In that case use VPC + Security group workarounds.

Security Groups

1. RDS Security Group: inbound allow Lambda SG, local IP (if needed) on port **5432**.
2. Lambda SG → allowed to reach RDS SG.

Best practice: don't make RDS publicly accessible unless absolutely needed in learner lab.

Connect via DBeaver & Node.js

In DBeaver:

- Host: endpoint from RDS console.
- Port: **5432**
- Username/password you set.

In Node.js:

```
js
```

```
const { Pool } = require('pg');
const pool = new Pool({ ...env });
```

 Copy code

5) Amazon Cognito — Authentication

Create User Pool

AWS Console → Cognito → Create user pool:

- Configure attributes (email, name, roles custom).
- Enable email/phone verification.

Roles (student, hostel_owner, mess_owner)

In Cognito:

- Create custom attributes for user role.
 - In React login, request tokens (JWTs).
 - Cognito issues access, ID tokens with role claims.
-

Integrating into React

Use AWS Amplify Auth or Cognito Hosted UI:

- `amazon-cognito-auth-js` or AWS Amplify library to login/signup.
-

JWT Handling in Backend

In Node.js:

```
js

const { CognitoJwtVerifier } = require("aws-jwt-verify");

const verifier = CognitoJwtVerifier.create({ userPoolId: "..." });
```

 Copy code

Verify JWT in each API request and enforce role permissions.

6) Amazon CloudFront — CDN for Frontend

Create CloudFront Distribution

1. Origin: your S3 bucket (or API Gateway for dynamic endpoints).
 2. Default behavior: serve static files.
-

Configure Caching & HTTPS

- Use **HTTPS only**.
 - Cache static files **longer** (immutable builds).
 - Invalidate **only changed assets** using invalidation patterns.
-

Update After Frontend Change

1. Upload new build to S3.
2. Run CloudFront Invalidations for changed files.

lua

```
aws cloudfront create-invalidation ...
```

 Copy code

7) AWS IAM — Permissions & Roles

Required Roles

1. Lambda Execution Role

- Allow RDS access, S3 access, CloudWatch Logs.

2. API Gateway Role

- Often integrated automatically with Lambda.

3. Cognito Roles

- Authenticated vs. unauthenticated roles for IAM.

Least Privilege

- Give only necessary actions.
- Do not use wildcards like "Resource": "*" .
- Always scope to your bucket ARN / specific tables / functions.

8) Amazon CloudWatch — Logs & Monitoring

View Lambda Logs

Lambda console → Monitor → **View logs in CloudWatch**.

Create Alerts

1. CloudWatch → Alarms → Metric → Lambda Errors.
2. Send notifications (SNS).

Tips for Beginners & Learner Lab Workarounds

- ✓ Always enable CORS on API Gateway for React.
 - ✓ Use presigned URLs for image upload instead of public buckets.
 - ✓ Don't delete images folder on deploy.
 - ✓ If learner lab blocks public access, use CloudFront + OAC.
 - ✓ Use serverless framework to simplify API Gateway + Lambda wiring.
 - ✓ Store secrets (DB creds) in SSM Parameter Store or Secrets Manager.
-

If you'd like, I can also generate [sample Terraform/Serverless config files](#), or a [project-specific deployment checklist](#) you can tick off while building StayMate!

     ...  Sources

explain ini detail to setup cloudwatch

Absolutely! Let's go deep dive into setting up [Amazon CloudWatch](#) for your StayMate serverless app — step by step, including [Lambda logging](#), [metrics](#), and [alerts](#). I'll explain it like you're actually going to configure it tomorrow.

Amazon CloudWatch Setup for StayMate

CloudWatch is the AWS service for [monitoring](#), [logging](#), and [alerting](#). For StayMate, we mainly use it to:

1. View logs from Lambda backend.
 2. Monitor metrics like errors, invocations, duration.
 3. Set alarms to notify you of issues.
-

Step 1: Ensure Lambda is Sending Logs to CloudWatch

By default, Lambda functions automatically log to CloudWatch, but only if the Lambda execution role has permission.

Lambda IAM Role

1. Go to [IAM](#) → [Roles](#) → Your Lambda Role.
2. Attach AWS managed policy:

nginx

AWSLambdaBasicExecutionRole

 Copy code

This gives:

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

 This is enough to start logging Lambda events to CloudWatch.

Step 2: Add Logging in Node.js Lambda

In your backend code:

js

```
exports.handler = async (event) => {
    console.log("Received event:", JSON.stringify(event));
    console.error("This is an error log");
    return {
        statusCode: 200,
        body: JSON.stringify({ message: "Hello from Lambda" })
    };
};
```

- `console.log()` → appears as **INFO** in CloudWatch Logs.
- `console.error()` → appears as **ERROR** in CloudWatch Logs.

 Copy code

Tip: Use structured logging like JSON for better metrics filtering.

Step 3: Access CloudWatch Logs

1. Go to AWS Console → CloudWatch → Logs → Log groups
2. Look for your Lambda function log group:

pgsql

/aws/lambda/<function-name>

 Copy code

3. Click → Log streams → see timestamped logs.

- Each Lambda invocation gets a log stream.
 - Logs include:
 - Start/End request IDs
 - Console logs
 - Errors/exceptions
-

Step 4: Monitor Metrics

CloudWatch automatically collects **Lambda metrics**:

- **Invocations** → # of times Lambda ran
 - **Duration** → execution time
 - **Errors** → failed invocations
 - **Throttles** → when concurrency limit hit
 - **IteratorAge** → if using streams (optional)
1. Go to CloudWatch → Metrics → Lambda Metrics
 2. Select your function → view metrics graphically.
 3. You can filter by time, function, error type, etc.
-

Step 5: Create CloudWatch Alarms

Alarms notify you when something goes wrong.

Example: Alert on Lambda errors

1. Go to CloudWatch → Alarms → Create alarm
 2. Select Lambda Metrics → By Function Name → Errors
 3. Set threshold:
 - e.g., if errors > 0 for 1 evaluation period (5 min).
 4. Configure Actions:
 - Send SNS notification to email/Slack/etc.
 5. Name and create alarm.
-  Now, if any Lambda invocation fails, you get notified immediately.
-

Step 6: Enable CloudWatch Insights for Advanced Logging

CloudWatch Logs Insights lets you query logs like SQL:

sql

```
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc
| limit 20
```

- Helps find failed API calls, DB errors, image upload issues.
- Very handy for serverless debugging.

 Copy code

Step 7: Monitor API Gateway & S3 (Optional)

CloudWatch can also monitor:

- **API Gateway**
 - Latency, 4xx/5xx errors
- **S3**
 - Bucket request metrics (via CloudWatch Metrics)

For your learner lab, focus first on **Lambda logs & alarms** — everything else can be added later.

Step 8: Best Practices

1. Use structured logs (JSON) — easier filtering and visualization.
2. Set retention period:
 - CloudWatch logs can accumulate. For learner lab, 1–3 days is enough.
3. Create alarms for critical failures only — don't spam notifications.
4. Use Log Insights queries to troubleshoot API or DB issues quickly.
5. Do not log sensitive info (passwords, tokens) — even in learner lab.

Summary of CloudWatch Setup for StayMate

Step	Purpose
1	Attach <code>AWSLambdaBasicExecutionRole</code> to Lambda
2	Add <code>console.log</code> / <code>console.error</code> in Node.js
3	View log streams under <code>/aws/lambda/<function></code>

Step	Purpose
4	Monitor metrics (Invocations, Duration, Errors)
5	Create alarms (e.g., Errors > 0 triggers SNS)
6	Use Logs Insights to query logs
7	Optional: monitor API Gateway & S3 metrics
8	Best practices: structured logs, retention, avoid sensitive info

If you want, I can draw a visual diagram showing how CloudWatch interacts with Lambda, API Gateway, RDS, and S3 for StayMate — that makes it extremely clear for debugging and monitoring.

Do you want me to do that next?



teach how to configure SES

Absolutely! Let's go step by step to configure Amazon SES (Simple Email Service) for your StayMate project, including sending emails from your backend (Node.js Lambda) for things like user registration, password resets, or notifications. I'll explain practical setup, restrictions in learner lab, and best practices.

Amazon SES Setup for StayMate

SES lets you send and receive emails via AWS. In your project, you'll mainly use it to send emails from Lambda or backend Node.js.

Step 1: Check SES Region & Learner Lab Restrictions

1. SES is region-specific. Go to AWS Console → SES → Region.

- For Learner Lab, you might be limited to US East (N. Virginia) or Oregon.
2. By default, new SES accounts are in **sandbox mode**:
- Can only send emails to **verified addresses** (you or test emails).
 - To send to anyone, you need to request **production access** — Learner Lab usually keeps sandbox mode.

Learner Lab Tip: While sandboxed, always verify **sender and recipient emails** for testing.

Step 2: Verify Email Addresses or Domain

SES requires **verified identities** to prevent spam.

Option A — Verify Single Email

1. Go to SES → **Verified identities** → **Create identity** → **Email Address**.
2. Enter your sending email (e.g., `noreply@staymate.com`).
3. Click **Verify** → AWS sends a confirmation email → click the link.

Option B — Verify Domain (Optional)

1. Go to SES → **Verified identities** → **Create identity** → **Domain**.
2. Add **TXT records** in your **DNS** to verify domain.
 - Allows sending from any email under that domain.
3. Recommended if you want `noreply@staymate.com`.

For Learner Lab and testing, verifying a single email is enough.

Step 3: Configure IAM Role for Lambda

Lambda needs permission to send emails via SES.

Create/Update Lambda Execution Role

1. Go to IAM → Roles → <LambdaRole> → **Attach policy**.
2. Use a **custom policy** for SES:

json

```
{  
  "Version": "2012-10-17",  
  "Statement": [
```

```
{  
    "Effect": "Allow",  
    "Action": [  
        "ses:SendEmail",  
        "ses:SendRawEmail"  
    ],  
    "Resource": "*"  
}  
]  
}
```

Best practice: restrict `"Resource"` to only your verified email or domain.

 Copy code

Step 4: Send Email from Node.js Lambda

Install AWS SDK in your Lambda project:

bash

```
npm install aws-sdk
```

 Copy code

Example code to send a simple email:

js

```
const AWS = require("aws-sdk");  
AWS.config.update({ region: "us-east-1" }); // SES region  
  
const ses = new AWS.SES();  
  
const params = {  
    Destination: {  
        ToAddresses: ["testuser@example.com"], // verified recipient  
    },  
    Message: {  
        Body: {  
            Text: { Data: "Hello! Welcome to StayMate." },  
        },  
        Subject: { Data: "Welcome to StayMate" },  
    },  
    Source: "noreply@yourverifiedemail.com", // verified sender  
};
```

```
ses.sendEmail(params, (err, data) => {
  if (err) {
    console.error("Error sending email:", err);
  } else {
    console.log("Email sent:", data.MessageId);
  }
});
```

Step 5: Sending HTML Emails

js

```
Message: {
  Body: {
    Html: {
      Data: `<h1>Welcome to StayMate</h1><p>Your account is ready!</p>`,
    },
  },
  Subject: { Data: "Welcome to StayMate" },
}
```

 Copy code

Step 6: Test in Sandbox Mode

- Only verified sender → verified recipient.
- Use your personal email to test SES integration.
- Logs errors in Lambda → check CloudWatch.

Step 7: Best Practices

1. Avoid hardcoding emails: use environment variables for sender email.
2. Use SES with Lambda asynchronously:

js

```
await ses.sendEmail(params).promise();
```

 Copy code

3. Error handling: capture bounced or rejected emails.

4. Sandbox mode:

- All emails must be verified.
 - For production, request SES move to **production** (not possible in learner lab).
-

Step 8: Optional – SES with Templates

- SES supports **email templates** for dynamic content.
- You can define:

```
json

{
    "TemplateName": "WelcomeTemplate",
    "SubjectPart": "Welcome to StayMate",
    "HtmlPart": "<h1>Hello {{name}}</h1><p>Thanks for joining!</p>",
    "TextPart": "Hello {{name}}, Thanks for joining!"
}
```

- Then send using `SendTemplatedEmail`.

 Copy code

Good for sending personalized emails without editing code.

Summary Table

Step	Purpose
1	Choose SES region & understand sandbox restrictions
2	Verify sender email or domain
3	Give Lambda IAM role <code>ses:SendEmail / ses:SendRawEmail</code>
4	Send email via AWS SDK in Node.js Lambda
5	Use HTML for rich emails if needed
6	Test using verified recipient (sandbox mode)
7	Use environment variables, async calls, error handling

Step	Purpose
8	Optional: Use SES templates for dynamic emails

If you want, I can show a full StayMate workflow diagram that connects **React frontend → API Gateway → Lambda → RDS → SES** so you can see exactly how emails are triggered for signup, password reset, or booking notifications.

Do you want me to make that diagram next?



how to implement cloudfront

Perfect! Let's go **step by step** to implement **Amazon CloudFront** for your **StayMate** frontend and optionally for your backend API. I'll explain **how it interacts with S3 and Lambda/API Gateway, best practices, and learner lab restrictions**.

Amazon CloudFront Setup for StayMate

CloudFront is a **CDN (Content Delivery Network)**. It caches content close to your users, making your app **faster** and more **secure**.

In StayMate, we'll use CloudFront to:

1. Serve your **React frontend** from S3.
2. Optionally cache static API responses (read-only endpoints) from API Gateway/Lambda.
3. Add HTTPS automatically with AWS-managed certificates.

Step 1: Create an S3 Bucket (if not already)

1. Go to **AWS Console → S3 → Create bucket** (or use existing one with React build).
2. Enable **Static Website Hosting**:

- **Index Document:** `index.html`
 - **Error Document:** `index.html` (for SPA routing)
3. Upload your React build folder.
 4. Make sure your bucket permissions allow CloudFront access:
 - Either public read **or** use **Origin Access Control (OAC)**.

Learner Lab Tip: Public access may be restricted. Using OAC is better — CloudFront fetches content securely without making your bucket public.

Step 2: Create a CloudFront Distribution

1. Go to CloudFront → Create Distribution → Web (or use new “CloudFront distributions” console).
2. Origin Settings:
 - **Origin Domain Name:** your S3 bucket (not website endpoint if using OAC)
 - **Origin Access:** if using OAC, create a new OAC and attach it.
3. Default Cache Behavior:
 - Viewer Protocol Policy: **Redirect HTTP to HTTPS**
 - Allowed HTTP Methods: `GET, HEAD, OPTIONS` (for frontend, no POST needed)
 - Cache Policy: choose **CachingOptimized** (default)
 - Can also create a **custom cache policy** if you want long TTL for static assets.
4. Distribution Settings:
 - Alternate Domain Names (CNAMEs): optional if you use custom domain (e.g., `www.staymate.com`)
 - SSL Certificate: **Default CloudFront certificate (*.cloudfront.net)** for beginner/learner lab
5. Click **Create Distribution**.

CloudFront takes ~10–15 minutes to deploy.

Step 3: Point Frontend Requests to CloudFront

Once the distribution is deployed, you get a **CloudFront domain name**, e.g.:

d123abcd1234.cloudfront.net

 Copy code

- This replaces the S3 website URL in your React app.
- Browser requests go to CloudFront → CloudFront fetches from S3 → serves to user.

Step 4: Handling SPA Routing (React)

React uses **client-side routing** (`react-router`).

- CloudFront must redirect **404s** to `index.html`.
- Steps:
 1. Go to CloudFront → Behaviors → Error Pages
 2. Create Custom Error Response:
 - HTTP Error Code: `404`
 - Customize Response: Yes
 - Response Page Path: `/index.html`
 - HTTP Response Code: `200`

This ensures URLs like `/hotels/123` load React app instead of giving 404.

Step 5: Update CloudFront When Frontend Changes

When you deploy a new React build:

1. Upload new `build` folder to S3.
2. Run **Invalidation** on CloudFront for changed files:

bash

```
aws cloudfront create-invalidation --distribution-id <ID> --paths "/*"
```

 Copy code

- This forces CloudFront to fetch **latest files** from S3.
- You can invalidate selectively for faster updates (`/static/js/*`).

Step 6: Optional – Frontend + API Gateway Integration

If you want **CloudFront to proxy API requests**:

1. Create another cache behavior:

- Path pattern: /api/*
- Origin: API Gateway
- Allowed Methods: GET, POST, PUT, DELETE, OPTIONS
- Cache Policy: CachingDisabled (dynamic API, don't cache)

2. Enable CORS on API Gateway so React + CloudFront works.

Benefit: now React app + API are on the same CloudFront domain, avoiding CORS issues.

Step 7: Best Practices

1. Use OAC instead of making S3 public — more secure.
 2. Cache static assets long, e.g., /static/js/*.js → 1 year TTL.
 3. Invalidate only changed files to reduce costs.
 4. Force HTTPS for security.
 5. Custom domain:
 - Add Route 53 record pointing to CloudFront distribution.
 - Attach SSL certificate from AWS Certificate Manager.
-

Step 8: Learner Lab Considerations

- Learner Lab may restrict:
 - Custom domains → stick to default CloudFront URL.
 - Some caching policies → use default.
 - OAC might not be fully available → you may need public bucket temporarily.
-

Summary Table for StayMate CloudFront

Step	Purpose
1	Upload React build to S3 bucket
2	Create CloudFront distribution pointing to S3
3	Use CloudFront domain for frontend access
4	SPA routing: 404 → index.html

Step	Purpose
5	Update distribution: invalidate changed files after deploy
6	Optional: Proxy API requests via CloudFront
7	Best practices: OAC, HTTPS, caching, selective invalidation
8	Learner lab workaround: may need public S3

I can also make a **full architecture diagram showing React → CloudFront → S3 + Lambda/API Gateway → RDS + SES + Cognito** for StayMate — this will make it crystal clear how everything flows.

Do you want me to make that diagram next?



i cant use ses, cloudfront as its a learner lab, any alternatives

Ah! Got it — since **AWS Learner Lab** restricts SES and CloudFront, you'll need **workarounds** to achieve similar functionality for **StayMate**. Here's a detailed breakdown of alternatives and how to implement them.

1 SES Alternative (Email Sending)

Problem: SES is sandboxed in learner lab, so you can't send emails to arbitrary addresses.

Alternative Options:

A) Use Third-Party Email Services (Easier)

- SendGrid, Mailgun, Gmail SMTP
- Pros:
 - Free tiers available for small-scale testing

- Can send to any email (no sandbox restrictions)
- How to integrate in Node.js:

js

```
// Using Nodemailer + Gmail
const nodemailer = require("nodemailer");

const transporter = nodemailer.createTransport({
  service: "gmail",
  auth: {
    user: process.env.GMAIL_USER,
    pass: process.env.GMAIL_PASSWORD
  }
});

const mailOptions = {
  from: process.env.GMAIL_USER,
  to: "recipient@example.com",
  subject: "Welcome to StayMate",
  text: "Hello! Your account is ready."
};

transporter.sendMail(mailOptions, (err, info) => {
  if(err) console.error(err);
  else console.log("Email sent:", info.response);
});
```

- **Tips for learner lab:** store credentials in `.env` or Lambda environment variables.

B) Mock Emails for Testing

- If your goal is **demo only**, you can log emails to CloudWatch instead of sending them.

js

```
console.log("Sending email to:", userEmail, "Subject:", subject, "Body:", body);
```

 Copy code

- Later, when deployed outside learner lab, you can replace this with SendGrid or SES.

2 CloudFront Alternative (Frontend CDN / HTTPS)

Problem: Learner lab restricts creating CloudFront distributions.

A) Use S3 Static Website Directly

- S3 website URL works for small projects.
- Limitations:
 - HTTP only (no HTTPS)
 - No caching at edge nodes
- Steps:
 1. Enable **Static Website Hosting** in S3.
 2. Upload React build folder.
 3. Access via S3 website endpoint:

```
php-template
```

```
http://<bucket-name>.s3-website-<region>.amazonaws.com
```

 Copy code

- For SPA routing: set **index.html** as error document.

B) Use API Gateway + Lambda as Proxy

- If you need **HTTPS**, you can serve your frontend through **API Gateway**:
 1. Create a Lambda function that **serves the React build files**.
 2. API Gateway URL provides **HTTPS endpoint**.
- Pros:
 - Works in learner lab
 - HTTPS supported
- Cons:
 - No caching
 - Slightly more setup

C) Use Local Reverse Proxy / Ngrok for Demo

- For testing, you can serve React locally via `npm start` and expose it using **ngrok**.
 - Free HTTPS tunnel for demo purposes.
 - Example:

```
bash
```

ngrok http 3000

 Copy code

- Use URL in your Lambda / API calls for testing.

3 Adjusted Architecture in Learner Lab

Since SES & CloudFront are blocked, here's a simplified **StayMate workflow**:

java

React [Frontend](#) (S3 Website / Local)

|

| HTTPS/HTTP requests

v

API Gateway → Lambda (Node.js Express)

|

 Copy code

| Queries

v

Amazon [RDS](#) (PostgreSQL)

- **Email Sending:** use SendGrid or Nodemailer SMTP (instead of SES)
- **Frontend Hosting:** S3 static website (instead of CloudFront)
- **HTTPS for frontend:** API Gateway proxy or ngrok (for demo)

4 Tips for Learner Lab Limitations

1. **Testing Email:** always verify sender & recipient if using SES sandbox; otherwise use SendGrid/Gmail SMTP.
2. **Frontend Deployment:** accept S3 website URL for now; simulate CDN caching locally if needed.
3. **HTTPS for local demo:** use ngrok.
4. **Prepare for future deployment:** Once outside learner lab, replace S3+API Gateway with CloudFront and SES for production-level behavior.

If you want, I can **rewrite your StayMate deployment guide entirely for learner lab**, removing SES and CloudFront and giving **step-by-step instructions** for S3, Lambda, API Gateway, RDS, Cognito, and email alternatives.

Do you want me to do that?



AWS Report

Role: Act as a Senior Cloud Solutions Architect and Academic Technical Writer.

Task: Generate a complete, professional LaTeX technical report for a university semester project.

Project Title: StayMate – A Centralized Hostel & Mess Booking Marketplace

Project Domain: Cloud Computing / AWS Serverless Architecture

Project Overview:

StayMate is a centralized digital platform designed to help university students search, compare, and book verified hostels and mess facilities. The system is implemented using AWS Serverless Architecture to ensure scalability, security, and cost efficiency.

Architecture Requirements:

- Frontend: React.js application hosted on Amazon S3 (static website hosting)
- Backend: AWS Lambda (Node.js or Python)
- API Layer: Amazon API Gateway (REST API)
- Authentication: Amazon Cognito User Pools
- Storage: Amazon S3 (images, static assets)
- Notifications: Amazon SNS (booking notifications)
- Monitoring: Amazon CloudWatch
- Networking: Explanation of VPC, private subnets, NAT Gateway, and NLB (conceptual)

Functional Requirements:

- Users can search hostels and mess facilities by city, price range, and type.
- Users can view property details including images and mess menus.
- Users can create bookings.
- Property owners can upload listings and images using pre-signed S3 URLs.
- Booking notifications are sent using SNS.

Report Requirements (MANDATORY):

Generate LaTeX code that includes ALL of the following:

1. Cover Page

- Project title: StayMate
- Subtitle: A Centralized Hostel & Mess Booking Marketplace
- Student names & registration numbers (placeholders)
- Course name
- Semester & year

2. Executive Summary (chapter*)

- Clearly explains problem, solution, AWS services used, and outcomes

3. Table of Contents

4. List of Figures

5. List of Tables

6. Chapter 1: Introduction

- Background
- Motivation
- Domain relevance

7. Chapter 2: Problem Statement

- Clearly defined real-world problem
- Justification

8. Chapter 3: Aim and Objectives

- One clear aim
- Bullet-point objectives aligned with AWS architecture

9. Chapter 4: Proposed Solution

- Functional overview
- Why serverless architecture is used

10. Chapter 5: System Architecture

- Explanation of full AWS architecture
- Include placeholder for architecture diagram using

\includegraphics

11. Chapter 6: AWS Services Used

- Amazon S3
- API Gateway
- AWS Lambda
- Amazon Cognito
- Amazon SNS
- Amazon CloudWatch
- VPC, NAT Gateway, NLB (conceptual explanation)

12. Chapter 7: Implementation Details

- Frontend implementation
- Backend implementation
- API flow
- Image upload using pre-signed URLs

13. Chapter 8: Security and IAM

- Least privilege
- Cognito authorizers
- Private Lambdas
- NAT Gateway usage

14. Chapter 9: Results and Output

- Functional results
- Screenshots mentioned (placeholders only)

15. Chapter 10: Conclusion and Future Work

- Summary
- Future enhancements

16. References

- IEEE-style references
- AWS documentation
- React documentation
- Tailwind CSS documentation

LaTeX Formatting Requirements:

- Use report class (12pt, A4)
- Include required packages (graphicx, geometry, hyperref, setspace)
- 1.5 line spacing
- Proper chapter and section structure

- Roman page numbering for front matter
- Arabic numbering for main content
- IEEE-style bibliography

Output Format:

- Output ONLY the complete LaTeX source code
- Do NOT include explanations or commentary
- Ensure the code is compile-ready in Overleaf

latex

 Copy code

```
\documentclass[12pt,a4paper]{report}

% ----- Packages -----
\usepackage[a4paper,margin=1in]{geometry}
\usepackage{graphicx}
\usepackage{setspace}
\usepackage{hyperref}
\usepackage{tocloft}
\usepackage{titlesec}
\usepackage{float}

% ----- Formatting -----
\onehalfspacing
\hypersetup{
    colorlinks=true,
    linkcolor=black,
    urlcolor=blue,
    citecolor=black
}

% ----- Document -----
\begin{document}

% ----- Cover Page -----
\begin{titlepage}
    \centering
    \vspace*{2cm}
    {\Huge \textbf{StayMate}}\\[0.5cm]
    {\Large \textit{A Centralized Hostel & Mess Booking Marketplace}}\\[2cm]
```

```
{\large \textbf{Semester Project Report}}\\[0.5cm]  
\large Cloud Computing / AWS Serverless Architecture}\\[2cm]
```

```
\textbf{Submitted By}\\[0.3cm]  
Student Name 1 \hspace{1cm} Registration No. XXXXX\\  
Student Name 2 \hspace{1cm} Registration No. XXXXX\\[1.5cm]
```

```
\textbf{Course Name}\\  
Cloud Computing\\[0.5cm]
```

```
\textbf{Semester \& Year}\\  
Fall 2025\\[2cm]
```

```
\vfill  
\textbf{Department of Computer Science}\\  
University Name  
\end{titlepage}
```

```
% ----- Front Matter -----
```

```
\pagenumbering{roman}
```

```
\chapter*{Executive Summary}
```

```
StayMate is a centralized digital platform designed to simplify the process c
```

```
The system leverages modern cloud services such as Amazon S3 for static hosti
```

```
The outcome of this project is a fully functional prototype demonstrating how
```

```
\tableofcontents  
\listoffigures  
\listoftables
```

```
\clearpage  
\pagenumbering{arabic}
```

```
% ----- Chapter 1 -----
```

```
\chapter{Introduction}
```

```
\section{Background}
```

```
Finding suitable accommodation and mess facilities is a recurring challenge f
```

```
\section{Motivation}
```

```
The motivation behind StayMate is to create a trusted, centralized platform t
```

```
\section{Domain Relevance}
```

This project lies in the domain of Cloud Computing and demonstrates the pract

```
% ----- Chapter 2 -----
```

```
\chapter{Problem Statement}
```

Students often struggle to find reliable hostel and mess services due to scat

```
% ----- Chapter 3 -----
```

```
\chapter{Aim and Objectives}
```

```
\section{Aim}
```

To design and implement a scalable, secure, and cost-effective hostel and mes

```
\section{Objectives}
```

```
\begin{itemize}
```

- \item Develop a responsive frontend using React.js.
- \item Implement backend services using AWS Lambda.
- \item Design RESTful APIs using Amazon API Gateway.
- \item Integrate secure authentication using Amazon Cognito.
- \item Enable image uploads using Amazon S3 and pre-signed URLs.
- \item Send booking notifications using Amazon SNS.
- \item Monitor system performance using Amazon CloudWatch.

```
\end{itemize}
```

```
% ----- Chapter 4 -----
```

```
\chapter{Proposed Solution}
```

```
\section{Functional Overview}
```

StayMate allows users to search and compare hostels and mess facilities, view

```
\section{Why Serverless Architecture}
```

Serverless architecture eliminates server management, provides automatic scal

```
% ----- Chapter 5 -----
```

```
\chapter{System Architecture}
```

The system follows a fully serverless architecture on AWS. The frontend is hc

```
\begin{figure}[H]
```

```
    \centering
```

```
    \includegraphics[width=0.9\textwidth]{architecture_diagram_placeholder.png}
```

```
    \caption{StayMate AWS Serverless Architecture}
```

```
\end{figure}
```

```
% ----- Chapter 6 -----
```

```
\chapter{AWS Services Used}
```

```
\section{Amazon S3}
```

Used for static website hosting, image storage, and asset management.

```
\section{Amazon API Gateway}
```

Provides RESTful APIs to connect the frontend with backend Lambda functions.

```
\section{AWS Lambda}
```

Handles backend business logic such as searching listings and creating bookings.

```
\section{Amazon Cognito}
```

Manages user authentication, authorization, and user pools.

```
\section{Amazon SNS}
```

Sends booking confirmation and notification messages.

```
\section{Amazon CloudWatch}
```

Used for monitoring, logging, and debugging Lambda functions.

```
\section{VPC, NAT Gateway, and NLB}
```

A Virtual Private Cloud (VPC) is conceptually used to isolate backend services from the public internet.

```
% ----- Chapter 7 -----
```

```
\chapter{Implementation Details}
```

```
\section{Frontend Implementation}
```

The frontend is developed using React.js and hosted on Amazon S3 as a static website.

```
\section{Backend Implementation}
```

The backend consists of multiple AWS Lambda functions written in Node.js or Python.

```
\section{API Flow}
```

The frontend communicates with backend services through REST APIs exposed by Lambda functions.

```
\section{Image Upload Using Pre-Signed URLs}
```

Property owners upload images directly to S3 using pre-signed URLs generated by Lambda functions.

```
% ----- Chapter 8 -----
```

```
\chapter{Security and IAM}
```

Security is enforced using the principle of least privilege. IAM roles restrict access to specific services and actions.

% ----- Chapter 9 -----

\chapter{Results and Output}

The system successfully allows users to browse listings, create bookings, and manage payments.

\begin{figure}[H]

 \centering

 \includegraphics[width=0.8\textwidth]{output_screenshot_placeholder.png}

 \caption{Application Output Screenshot}

\end{figure}

% ----- Chapter 10 -----

\chapter{Conclusion and Future Work}

\section{Conclusion}

StayMate demonstrates how AWS Serverless Architecture can be used to build scalable and reliable travel booking systems.

\section{Future Work}

Future enhancements include payment gateway integration, recommendation systems, and multi-language support.

% ----- References -----

\begin{thebibliography}{99}

\bibitem{aws}

Amazon Web Services Documentation. \url{\url{https://docs.aws.amazon.com}}

\bibitem{react}

React Documentation. \url{\url{https://react.dev}}

\bibitem{tailwind}

Tailwind CSS Documentation. \url{\url{https://tailwindcss.com/docs}}

\bibitem{serverless}

M. Roberts, \textit{Serverless Architectures on AWS}. AWS Whitepapers.

\end{thebibliography}

\end{document}

What tables can we add to make it more detailed

To make the **StayMate technical report** more detailed, academic, and evaluation-friendly, you should add **structured tables** that summarize design decisions, mappings, and results. Below are **high-value tables** that universities and evaluators love because they show clarity, planning, and engineering maturity.

I'll group them **chapter-wise** and explain **why each table matters**. You can then decide which ones to include.

◆ Chapter 1 / 2: Problem & Context Tables

Table 1: Stakeholders and Their Needs

• *Place in Chapter 2 – Problem Statement*

Stakeholder	Problems Faced	System Solution
Students	Unverified hostels, manual booking	Centralized verified listings
Property Owners	No digital platform	Online listing & management
University Admin	Lack of transparency	Structured data & logs

Shows real-world relevance

Table 2: Existing System vs Proposed System

• *Chapter 2*

Aspect	Existing System	StayMate System
Booking Method	Manual / WhatsApp	Online
Verification	None	Verified listings

Aspect	Existing System	StayMate System
Scalability	Low	High (AWS)
Availability	Limited	24/7
<input checked="" type="checkbox"/> Very strong marks booster		

◆ Chapter 3: Objectives & Scope

Table 3: Objectives Mapping with AWS Services

- *Chapter 3 – Aim & Objectives*

Objective	AWS Service Used
Static frontend hosting	Amazon S3
Backend logic execution	AWS Lambda
Secure user authentication	Amazon Cognito
Booking notifications	Amazon SNS
Monitoring & logging	Amazon CloudWatch

- Shows clear architectural thinking

◆ Chapter 4: Functional Understanding

Table 4: Functional Requirements

- *Chapter 4 – Proposed Solution*

Feature	Description
Hostel search	Filter by city, price, type
Mess details	View menus and timings
Booking system	Create and manage bookings

Feature	Description
Image upload	Pre-signed S3 URLs

Table 5: Non-Functional Requirements

• *Chapter 4*

Requirement	Description
Scalability	Auto-scaling serverless backend
Security	IAM + Cognito
Availability	AWS managed services
Cost Efficiency	Pay-as-you-go

✓ Professors love NFR tables

◆ **Chapter 5: Architecture**

Table 6: System Components Description

• *Chapter 5 – System Architecture*

Component	Responsibility
React Frontend	UI & client logic
API Gateway	Request routing
Lambda	Business logic
S3	Storage
Cognito	Authentication

◆ **Chapter 6: AWS Services Deep Dive**

Table 7: AWS Services Justification

📍 *Chapter 6*

AWS Service	Purpose	Reason for Selection
S3	Storage	Highly durable
Lambda	Compute	No server management
API Gateway	API	Secure REST APIs
Cognito	Auth	Managed authentication
SNS	Notify	Event-based messaging

◆ Chapter 7: Implementation

Table 8: API Endpoints

📍 *Chapter 7 – Implementation*

Method	Endpoint	Description
GET	/hostels	Fetch listings
POST	/booking	Create booking
POST	/upload-url	Generate pre-signed URL

🔥 This table makes your backend look **very professional**

Table 9: Image Upload Flow

📍 *Chapter 7*

Step	Description
1	User requests upload
2	Lambda generates URL
3	Client uploads to S3

Step	Description
4	URL saved in database

◆ Chapter 8: Security

Table 10: IAM Roles & Permissions

📍 *Chapter 8 – Security*

IAM Role	Permissions
LambdaExecutionRole	CloudWatch logs, S3 access
CognitoAuthRole	API access
AdminRole	Listing approval

Table 11: Security Controls

📍 *Chapter 8*

Layer	Security Measure
API	Cognito Authorizer
Storage	Private S3 buckets
Network	VPC + NAT Gateway

◆ Chapter 9: Results

Table 12: Test Results

📍 *Chapter 9 – Results*

Test Case	Expected Result	Status
User login	Successful	Pass

Test Case	Expected Result	Status
Booking creation	Confirmation	Pass
Notification	SNS triggered	Pass

◆ Chapter 10: Future Work

Table 13: Future Enhancements



Feature	Description
Online payments	Stripe / Razorpay
Mobile app	React Native
AI recommendations	ML-based matching

🔥 Minimum Recommended Tables (If Short on Time)

If you want **maximum marks with minimum effort**, include these **6 tables**:

1. Existing vs Proposed System
2. Objectives vs AWS Services
3. Functional Requirements
4. AWS Services Justification
5. API Endpoints
6. IAM Roles & Permissions

If you want, I can:

- Insert these tables directly into your LaTeX code
- Generate LaTeX table syntax
- Reduce or expand tables based on rubric
- Make tables diagram-aligned with AWS Academy

Just tell me

generate the latex code for these

latex

% ===== TABLES FOR STAYMATE REPORT =====

% ----- Chapter 2 Tables -----

```
\begin{table}[H]
\centering
\caption{Stakeholders and Their Needs}
\begin{tabular}{|p{4cm}|p{5cm}|p{5cm}|}
\hline
\textbf{Stakeholder} & \textbf{Problems Faced} & \textbf{System Solution} \\
\hline
Students & Unverified hostels, manual booking & Centralized verified listings  

Property Owners & No digital platform for listings & Online listing and manag  

University Admin & Lack of transparency & Structured data and monitoring \\
\hline
\end{tabular}
\end{table}
```

```
\begin{table}[H]
\centering
\caption{Existing System vs Proposed System}
\begin{tabular}{|p{4cm}|p{5cm}|p{5cm}|}
\hline
\textbf{Aspect} & \textbf{Existing System} & \textbf{StayMate System} \\
\hline
Booking Method & Manual / WhatsApp & Online booking platform \\
Verification & None & Verified listings \\
Scalability & Low & High (AWS Serverless) \\
Availability & Limited & 24/7 Access \\
\hline
\end{tabular}
\end{table}
```

% ----- Chapter 3 Tables -----

```

\begin{table}[H]
\centering
\caption{Objectives Mapping with AWS Services}
\begin{tabular}{|p{7cm}|p{6cm}|}
\hline
\textbf{Objective} & \textbf{AWS Service Used} \\
\hline
Static frontend hosting & Amazon S3 \\
Backend business logic & AWS Lambda \\
RESTful API communication & Amazon API Gateway \\
User authentication & Amazon Cognito \\
Booking notifications & Amazon SNS \\
Monitoring and logging & Amazon CloudWatch \\
\hline
\end{tabular}
\end{table}

```

% ----- Chapter 4 Tables -----

```

\begin{table}[H]
\centering
\caption{Functional Requirements}
\begin{tabular}{|p{5cm}|p{7cm}|}
\hline
\textbf{Feature} & \textbf{Description} \\
\hline
Hostel and mess search & Filter by city, price range, and type \\
Property details & Images, descriptions, and mess menus \\
Booking system & Create and manage bookings \\
Image upload & Upload images using pre-signed S3 URLs \\
\hline
\end{tabular}
\end{table}

```

```

\begin{table}[H]
\centering
\caption{Non-Functional Requirements}
\begin{tabular}{|p{5cm}|p{7cm}|}
\hline
\textbf{Requirement} & \textbf{Description} \\
\hline
Scalability & Auto-scaling using serverless services \\

```

```
Security & IAM roles and Cognito authentication \\  
Availability & High availability using managed AWS services \\  
Cost Efficiency & Pay-as-you-go pricing model \\  
\hline  
\end{tabular}  
\end{table}
```

% ----- Chapter 5 Tables -----

```
\begin{table}[H]  
\centering  
\caption{System Components Description}  
\begin{tabular}{|p{5cm}|p{7cm}|}  
\hline  
\textbf{Component} & \textbf{Responsibility} \\  
\hline  
React Frontend & User interface and client-side logic \\  
Amazon API Gateway & Request routing and API management \\  
AWS Lambda & Backend business logic processing \\  
Amazon S3 & Static hosting and image storage \\  
Amazon Cognito & User authentication and authorization \\  
\hline  
\end{tabular}  
\end{table}
```

% ----- Chapter 6 Tables -----

```
\begin{table}[H]  
\centering  
\hline  
\textbf{AWS Service} & \textbf{Purpose} & \textbf{Reason for Selection} \\  
\hline  
Amazon S3 & Storage & Highly durable and scalable \\  
AWS Lambda & Compute & No server management required \\  
API Gateway & API Layer & Secure RESTful APIs \\  
Amazon Cognito & Authentication & Managed identity service \\  
Amazon SNS & Notifications & Event-driven messaging \\  
Amazon CloudWatch & Monitoring & Centralized logging and metrics \\  
\hline  
\end{tabular}  
\end{table}
```

ChatGPT ▾



```
% ----- Chapter 7 Tables -----
```

```
\begin{table}[H]
\centering
\caption{API Endpoints}
\begin{tabular}{|p{3cm}|p{4cm}|p{6cm}|}
\hline
\textbf{Method} & \textbf{Endpoint} & \textbf{Description} \\
\hline
GET & /hostels & Fetch hostel and mess listings \\
GET & /hostels/{id} & Fetch property details \\
POST & /booking & Create a booking \\
POST & /upload-url & Generate pre-signed S3 URL \\
\hline
\end{tabular}
\end{table}
```

```
\begin{table}[H]
\centering
\caption{Image Upload Flow Using Pre-Signed URLs}
\begin{tabular}{|p{2cm}|p{10cm}|}
\hline
\textbf{Step} & \textbf{Description} \\
\hline
1 & Client requests image upload permission \\
2 & Lambda generates pre-signed URL \\
3 & Client uploads image directly to S3 \\
4 & Image URL is stored in database \\
\hline
\end{tabular}
\end{table}
```

```
% ----- Chapter 8 Tables -----
```

```
\begin{table}[H]
\centering
\caption{IAM Roles and Permissions}
\begin{tabular}{|p{5cm}|p{7cm}|}
\hline
\textbf{IAM Role} & \textbf{Permissions} \\
\hline
LambdaExecutionRole & CloudWatch logs, S3 access \\
\hline
\end{tabular}
```

```
CognitoAuthRole & API Gateway access \\  
AdminRole & Listing approval and management \\  
\hline  
\end{tabular}  
\end{table}
```

```
\begin{table}[H]  
\centering  
\caption{Security Controls}  
\begin{tabular}{|p{4cm}|p{8cm}|}  
\hline  
\textbf{Layer} & \textbf{Security Measure} \\  
\hline  
API Layer & Cognito Authorizer \\  
Storage Layer & Private S3 buckets \\  
Network Layer & VPC with NAT Gateway \\  
\hline  
\end{tabular}  
\end{table}
```

% ----- Chapter 9 Tables -----

```
\begin{table}[H]  
\centering  
\caption{System Test Results}  
\begin{tabular}{|p{4cm}|p{5cm}|p{3cm}|}  
\hline  
\textbf{Test Case} & \textbf{Expected Result} & \textbf{Status} \\  
\hline  
User Login & Successful authentication & Pass \\  
Booking Creation & Booking confirmation & Pass \\  
Notification Service & SNS message sent & Pass \\  
\hline  
\end{tabular}  
\end{table}
```

% ----- Chapter 10 Tables -----

```
\begin{table}[H]  
\centering  
\caption{Future Enhancements}  
\begin{tabular}{|p{5cm}|p{7cm}|}  
\hline
```

```
\textbf{Feature} & \textbf{Description} \\  
\hline  
Online Payments & Integration with payment gateways \\  
Mobile Application & React Native based app \\  
AI Recommendations & Smart hostel matching system \\  
\hline  
\end{tabular}  
\end{table}
```



generate the README file content based on the same info

Here's a complete, professional **README.md** for your StayMate project, based entirely on the same technical report info:

markdown

```
# StayMate  
  
**A Centralized Hostel & Mess Booking Marketplace**
```

Table of Contents

- [Project Overview](#project-overview)
- [Features](#features)
- [Architecture](#architecture)
- [AWS Services Used](#aws-services-used)
- [Implementation](#implementation)
- [Security](#security)
- [Setup & Deployment](#setup--deployment)
- [Results](#results)
- [Future Work](#future-work)
- [References](#references)

Project Overview

StayMate is a centralized platform designed to simplify the process of searching for hostels and messes.

The platform ensures:

- Verified listings for hostels and messes
- Seamless booking flow
- Notifications for booking confirmations
- Cost-effective and scalable infrastructure

Features

- **Search Hostels & Messes:** Filter by city, price range, and type
- **Property Details:** View images, descriptions, and mess menus
- **Booking System:** Create and manage bookings
- **Property Management:** Owners can upload and manage listings using pre-saved templates
- **Notifications:** Booking notifications via Amazon SNS

Architecture

The system is designed with a **serverless architecture** for scalability and cost-efficiency.

- **Frontend:** React.js hosted on Amazon S3 (Static Website Hosting)
- **Backend:** AWS Lambda functions (Node.js or Python)
- **API Layer:** Amazon API Gateway (REST API)
- **Authentication:** Amazon Cognito User Pools
- **Storage:** Amazon S3 for images and assets
- **Notifications:** Amazon SNS
- **Monitoring:** Amazon CloudWatch
- **Networking:** Conceptual use of VPC, private subnets, NAT Gateway, and NLBs

![Architecture Diagram]([architecture_diagram_placeholder.png](#))

AWS Services Used

Service Purpose

----- -----
Amazon S3 Static website hosting and image storage
AWS Lambda Backend business logic
Amazon API Gateway RESTful APIs
Amazon Cognito Authentication & authorization
Amazon SNS Booking notifications
Amazon CloudWatch Monitoring & logging
VPC / NAT Gateway / NLB Network isolation and secure connectivity

Implementation

Frontend

- Built with React.js and hosted on Amazon S3 as a static website
- Responsive UI with hostel search, property details, and booking pages

Backend

- AWS Lambda functions implement business logic such as:
 - Searching listings
 - Creating bookings
 - Generating pre-signed URLs for image uploads
- API Gateway exposes REST endpoints to the frontend

Image Upload Flow

1. User requests permission to upload images
2. Lambda generates a pre-signed S3 URL
3. User uploads image directly to S3
4. Image URL is stored in the database

API Endpoints

Method Endpoint Description
----- ----- -----
GET /hostels Fetch listings
GET /hostels/{id} Fetch details
POST /booking Create a booking
POST /upload-url Generate pre-signed S3 URL

Security

- Principle of **Least Privilege** applied in IAM roles

- **Cognito Authorizers** secure API Gateway endpoints
- Lambda functions run in private subnets (VPC)
- NAT Gateway used for controlled outbound access
- Private S3 buckets for image storage

Setup & Deployment

1. Clone the repository:

```
```bash
git clone https://github.com/username/staymate.git
```

### 2. Install frontend dependencies:

```
bash

cd frontend
npm install
```

 Copy code

3. Deploy frontend to S3 static hosting
4. Configure AWS Lambda functions with API Gateway
5. Set up Amazon Cognito User Pools for authentication
6. Configure SNS for booking notifications
7. Update environment variables (S3 bucket names, API endpoints, Cognito IDs)

---

## Results

- Users can successfully browse listings, book hostels, and receive notifications
- Property owners can manage listings with image uploads
- AWS monitoring ensures system logs and metrics are captured
- Screenshots of the application UI and AWS console can be found in the `docs/screenshots` folder

---

## Future Work

- Integrate online payment gateways (Stripe, Razorpay)
- Develop a mobile application using React Native
- Implement AI-based recommendations for hostel matching

 Copy code

- Add analytics dashboards for property owners and administrators
- 

## References

1. Amazon Web Services Documentation – <https://docs.aws.amazon.com> ↗
2. React.js Documentation – <https://react.dev> ↗
3. Tailwind CSS Documentation – <https://tailwindcss.com/docs> ↗
4. Roberts, M., *Serverless Architectures on AWS*, AWS Whitepapers

yaml

 Copy code

---

If you want, I can \*\*also generate a “ready-to-use GitHub repo structure” with

Do you want me to do that next?



## Overall Guide

How to upload a dynamic react website on AWS, this website follows the serverless architecture

Since your React app follows a serverless architecture, the correct AWS-native way is:

React (frontend) → S3 + CloudFront  
Backend (serverless APIs) → AWS Lambda + API Gateway  
Auth / DB → Cognito, DynamoDB, etc.

I'll explain step-by-step, exactly how you'd do this for a university project / production-grade setup.

### 1 High-Level Architecture (What You're Building)

sql

User Browser



CloudFront (CDN + HTTPS)



S3 (React static build)



API Calls



API Gateway



Lambda Functions



DynamoDB / RDS

Your React app itself is static after build, even though it's "dynamic" at runtime.

 Copy code

### 2 Build Your React App

From your React project root:

```
bash
```

```
npm install
npm run build
```

 Copy code

This creates a:

```
arduino

/build
 ├── index.html
 ├── static/
 └── assets/
```

 Copy code

 Only the `/build` folder is uploaded to AWS, not your source code.

## 3 Create an S3 Bucket (Frontend Hosting)

### Step 1: Create Bucket

- AWS Console → S3
- Create bucket
- Bucket name:

```
staymate-frontend
```

 Copy code

- Region: same as your backend
-  Uncheck “Block all public access”

### Step 2: Enable Static Website Hosting

- Bucket → Properties
- Enable Static Website Hosting
- Index document:

```
diff
```

```
index.html
```

 Copy code

- Error document:

```
diff
```

*index.html*

 Copy code

(important for React routing)

## 4 Upload React Build to S3

Upload **everything inside `/build`**, not the folder itself.

Structure in S3 should look like:

cpp

`index.html`

`static/`

`assets/`

 Copy code

## 5 S3 Bucket Policy (Public Access)

Add this policy:

`json`

{

`"Version": "2012-10-17",`

`"Statement": [`

{

`"Sid": "PublicReadGetObject",`

`"Effect": "Allow",`

`"Principal": "*",`

`"Action": "s3:GetObject",`

`"Resource": "arn:aws:s3:::staymate-frontend/*"`

}

]

}

Now your site is accessible via:

`arduino`

`http://staymate-frontend.s3-website-<region>.amazonaws.com`

 Copy code

## 6 Add CloudFront (Highly Recommended)

### Why CloudFront?

- HTTPS
- Faster global delivery
- Required for professional/academic projects

### Steps:

1. AWS Console → **CloudFront**
2. Create Distribution
3. Origin:
  - Origin domain: **S3 website endpoint**
  - Origin type: S3
4. Default root object:

diff

*index.html*

 Copy code

### SPA Routing Fix (VERY IMPORTANT)

#### Add Custom Error Response:

- Error code: `403` and `404`
- Response page: `/index.html`
- HTTP response code: `200`

Now your React routing works.

## 7 Backend: Serverless APIs (Lambda + API Gateway)

### Typical Setup:

- API Gateway → REST or HTTP API
- Lambda functions for:
  - Login
  - Booking
  - Payments
  - Listings

Example endpoint:

```
bash
```

```
https://abc123.execute-api.us-east-1.amazonaws.com/prod/bookings
```

 Copy code

## 8 Connect React Frontend to Backend

### Environment Variables (React)

Create `.env` file:

```
env
```

```
REACT_APP_API_BASE_URL=https://abc123.execute-api.us-east-1.amazonaws.com/pro
```

 Copy code

Use in code:

```
js
```

```
fetch(` ${process.env.REACT_APP_API_BASE_URL}/bookings`)
```

 Copy code

Rebuild after setting env:

```
bash
```

```
npm run build
```

 Copy code

Re-upload to S3.

## 9 Authentication (Optional but Recommended)

### Best Serverless Options:

- Amazon Cognito
- JWT-based auth

Flow:

```
nginx
```

 Copy code

React → Cognito → JWT Token → API Gateway → Lambda

## 10 Database (Serverless-Friendly)

Choose based on project scope:

Use Case	Service
Fully serverless	DynamoDB
Relational	Aurora Serverless
Simple	Firebase / Supabase

## 1 1 CI/CD (Bonus – Great for Grades)

Use:

- GitHub Actions
- AWS CodePipeline

Pipeline:

pgsql

Git push → Build React → Upload to S3 → Invalidate CloudFront cache

 Copy code

## 1 2 Cost (Student-Friendly)

Service	Cost
S3	Free tier
CloudFront	Very cheap
Lambda	Free tier
API Gateway	Free tier

 Total: Almost free

## What to Write in Your Technical Report

You can say:

"The frontend is built using React.js and deployed on Amazon S3 with CloudFront CDN for global content delivery. The backend follows a fully serverless architecture using AWS Lambda and API Gateway, ensuring scalability, high availability, and cost efficiency."

### Want Next?

I can:

- Draw architecture diagram
- Give AWS Academy–friendly explanation
- Generate LaTeX report section
- Provide CI/CD YAML
- Map this exactly to StayMate / UniStay Connect

Just tell me 



Amazon Route 53  
Amazon CloudFront  
Amazon Cognito  
NAT Gateway  
Network Load Balancer (NLB)  
Amazon SNS  
Amazon SES  
Amazon OpenSearch Service  
Amazon ElastiCache  
Amazon S3  
AWS Lambda  
Amazon API Gateway  
Amazon RDS  
Amazon VPC  
AWS IAM  
Amazon CloudWatch  
from this list, filter out all the AWS services which are

available to be used in AWS learner lab, and if any service is not available enlist its alternative

Here's a **filtered list of the AWS services from your list that *are supported in AWS Academy Learner Lab*** (based on the official Learner Lab restrictions documents). I'll also list services *not supported* and suggest practical alternatives you can use within the lab.

## Services Available in AWS Learner Lab

The following services *are included* in the AWS Academy Learner Lab environment (Associate level restrictions list applies) Studocu :

AWS Service	Available in Learner Lab
Amazon Route 53 (but cannot register domains, only hosted zones/DNS)	 (limited) <small>Studocu</small>
Amazon CloudFront	 <small>Studocu</small>
Amazon Cognito	 <small>Studocu</small>
Amazon SNS	 <small>Studocu</small>
Amazon S3	 <small>Studocu</small>
AWS Lambda	 <small>Studocu</small>
Amazon API Gateway	 <small>Studocu</small>
Amazon RDS (limited instance sizes)	 <small>Studocu</small>
Amazon VPC	 <small>Studocu</small>
AWS IAM (very limited)	 (limited) <small>Studocu</small>
Amazon CloudWatch	 <small>Studocu</small>
Amazon ElastiCache	 <small>Studocu</small>

● Note: Some supported services have *functional limits*; e.g., IAM cannot create standard users/groups (only roles and limited actions).

Studocu

## ✖ Services Not Supported in Learner Lab (or Severely Limited)

AWS Service	Reason / Limitation	Suggested Alternative
NAT Gateway	Labs don't support it because it incurs costs and NAT requires VPC/IGW + NAT configuration that is not allowed/recommended in Learner Lab.	Instead simulate internet access using <b>public subnet with IGW</b> for EC2 or AWS Cloud9 for testing network scenarios.
Network Load Balancer (NLB)	Not listed as supported. AWS Academy typically supports only <b>Elastic Load Balancer (ELB)</b> (classic/ALB) as part of associate docs.	Use <b>Application Load Balancer (ALB)</b> instead for layer-7 traffic or classic ELB.
Amazon SES	Not in the supported list (SES isn't mentioned).	Use <b>SNS + Lambda</b> for notifications, or test SMTP through external sandbox accounts.
Amazon OpenSearch Service	Not listed in supported services.	For search/analytics, use <b>Amazon DynamoDB with secondary indexes</b> or <b>Elasticsearch via EC2</b> (if small scale).
NAT Gateway	Explicitly not supported. (High cost / networking limits)	Use Cloud9 or public subnets as workarounds.

## ● Behavior Notes in Learner Lab

### ✓ Route 53:

- You can create hosted zones and records, but **domain registration is disabled** (you cannot buy domains).

### ⚠ IAM:

- You cannot freely create IAM users or groups, and role creation is limited. A pre-created **LabRole** may be used by some services.

## General Restrictions:

- Services limited to **us-east-1** and **us-west-2 regions**. Pacific Cybersec...
  - Longer-running resources (e.g., RDS, NAT, ELB) consume credits and may persist outside of sessions, consuming credits. Apunts Institut ...
- 

## Summary

### Available (Go-to in Learner Lab)

- ✓ CloudFront
- ✓ Cognito
- ✓ API Gateway
- ✓ Lambda
- ✓ S3
- ✓ RDS (limited)
- ✓ VPC
- ✓ CloudWatch
- ✓ SNS
- ✓ Route 53 (DNS only)
- ✓ ElastiCache
- ✓ IAM (limited) Studocu

### Not Supported / Limited

- ✗ NAT Gateway
  - ✗ Network Load Balancer
  - ✗ Amazon SES
  - ✗ Amazon OpenSearch Service Studocu
- 

## Practical Suggestions

### For email sending (SES):

Use **Amazon SNS + Lambda** to simulate notifications, or use external SMTP sandbox.

### For search/analytics (OpenSearch):

Use **ElastiCache (Redis)** or **DynamoDB with Global Secondary Indexes** as a workable alternative in labs.

➡ For load balancing (NLB):

Use Application Load Balancer (ALB) instead, which is broadly supported.

---

If you want, I can also pair each supported service with a common lab assignment example (like *React hosting + API + Cognito login + alerts*) tailored for AWS Academy Lab limitations.

< 2/2 >      ...    Sources

enlist the tech stack that should be used

Since your project **StayMate** is a serverless React-based hostel & mess booking marketplace and you want to deploy it on AWS Learner Lab, here's a complete recommended tech stack including frontend, backend, database, serverless services, and dev tools that are fully supported in AWS Learner Lab. I've tailored it to AWS Academy limitations.

---

## 1 Frontend

Layer	Technology	Purpose / Notes
UI Framework	React.js	SPA (Single Page Application) for dynamic hostel/mess search & booking.
Styling	Tailwind CSS / Bootstrap	Rapid styling, responsive design.
State Management	React Context API or useState/useReducer	For small-medium scale projects, Context is sufficient.
HTTP Client	Axios / Fetch API	Communicate with backend APIs.
Build Tool	npm / Vite	Compile & bundle React app.

## 2 Backend (Serverless)

Layer	Technology	Purpose / Notes
Compute	<b>AWS Lambda</b>	Serverless functions for booking, listing, user management, payments.
API Gateway	<b>Amazon API Gateway (HTTP/REST)</b>	Expose Lambda functions as APIs.
Authentication	<b>Amazon Cognito</b>	User sign-up/sign-in, JWT token authentication.
Notification	<b>Amazon SNS</b>	Send booking confirmation, alerts to users.
Logging & Monitoring	<b>Amazon CloudWatch</b>	Logs and monitor Lambda/API usage.

### 3 Database / Storage

Layer	Technology	Purpose / Notes
Relational DB	<b>Amazon RDS (MySQL/PostgreSQL)</b>	Store users, bookings, hostels/mess data.
Caching / Session	<b>Amazon ElastiCache (Redis)</b>	Optional, for caching listings or session tokens.
File / Media Storage	<b>Amazon S3</b>	Store images of hostels/mess, documents.

### 4 Networking & Infrastructure

Layer	Technology	Notes / Alternatives for Learner Lab
DNS	<b>Amazon Route 53</b>	Only hosted zones supported; domain registration not allowed.
CDN / Hosting	<b>Amazon CloudFront</b>	Serve React app globally over HTTPS.
VPC	<b>Amazon VPC</b>	Isolate RDS, Lambda, ElastiCache in private network.

Layer	Technology	Notes / Alternatives for Learner Lab
Load Balancer	ALB (Application Load Balancer)	Alternative to NLB, for distributing traffic (if needed).
NAT / Internet Access	Public Subnet / Cloud9	NAT Gateway not supported; use public subnet to access internet.

## 5 DevOps / CI-CD / Monitoring

Layer	Technology	Purpose
Version Control	Git / GitHub	Source code management.
Build & Deploy	GitHub Actions or manual build → S3 / Lambda upload	CI/CD for frontend & backend.
Monitoring	CloudWatch + Lambda Metrics	Track function errors, latency.
Alerts	SNS Topics + CloudWatch Alarms	Notify admin/host in case of failures.

## 6 Optional / Advanced Features

Feature	Technology	Notes
Full-text Search	DynamoDB + GSI / ElastiCache	OpenSearch not available in learner lab.
Email Notifications	SNS + Lambda	SES not supported; simulate email via Lambda + SNS.
Payments	Stripe / Razorpay SDK	Handled via frontend, API triggers Lambda for serverless processing.

## ✓ Summary: Learner Lab-Compatible Stack

Frontend: React.js + Tailwind/Bootstrap + Axios

Backend (Serverless): Lambda + API Gateway + Cognito + CloudWatch + SNS

**Database:** RDS (MySQL/PostgreSQL) + ElastiCache (optional)

**Storage:** S3

**Networking / Hosting:** CloudFront + VPC + Route 53 (DNS only)

**CI/CD & Monitoring:** GitHub Actions + CloudWatch + SNS

This stack ensures your project is **fully serverless, AWS Academy-compliant, and production-ready in design.**

---

If you want, I can also **draw a full architecture diagram showing all these components** specifically for the **StayMate** project deployed in **AWS Learner Lab**, which you can include in your **technical report**.

Do you want me to make that diagram?



if there was a team of 2, how would they complete this project in 3 days

Perfect — let's break this down **realistically**. You have **2 people**, **3 days**, and the project is **StayMate**, a serverless React + AWS project. The key is **parallel work, minimal scope creep, and AWS Learner Lab-compatible stack**.

Here's a **day-wise plan and task division**.

---

## Team Roles & Responsibilities

With **2 people**, we can divide like this:

Team Member	Responsibility
Member 1	<b>Frontend + Hosting:</b> React SPA, UI components, styling, build, deploy to S3 + CloudFront.
Member 2	<b>Backend + Serverless APIs:</b> Lambda functions, API Gateway endpoints, RDS tables, Cognito auth, SNS notifications.

-  Both should collaborate on **API contracts** (endpoints, request/response schema) before coding.
- 

## Day 0: Preparation (Evening or few hours)

- Define **minimum viable product (MVP)** features:
    - User sign-up / login (Cognito)
    - Search hostels/mess
    - View details & book
    - Booking history
    - Host side: add listing
  - Finalize **DB schema** (users, hostels, bookings) for RDS.
  - Decide **API endpoints & request/response format**.
  - Share AWS Learner Lab access & assign IAM roles.
- 

## Day 1: Backend + Frontend Base Setup

### Member 2 (Backend)

1. Create RDS database + tables ( `users` , `hostels` , `bookings` ).
2. Configure Cognito user pool.
3. Implement Lambda functions for:
  - User registration / login (Cognito integration)
  - List hostels
  - Create booking
  - Fetch booking history
4. Setup API Gateway endpoints for all Lambda functions.
5. Test APIs in Postman.

### Member 1 (Frontend)

1. Initialize React app ( `create-react-app` / Vite).
  2. Setup routing for pages: Home, Search, Booking, Profile.
  3. Build UI components with Tailwind/Bootstrap:
    - Navbar, cards for hostels, forms for booking.
  4. Create mock API responses to connect UI with later backend.
  5. Commit initial code to GitHub.
-

## Day 2: Integration & Advanced Features

### Member 2 (Backend)

1. Integrate Lambda + API Gateway + RDS fully.
2. Connect Cognito authentication with Lambda (JWT validation).
3. Setup SNS for booking notifications.
4. Test end-to-end API flows.

### Member 1 (Frontend)

1. Replace mock API calls with **real API endpoints**.
  2. Handle **JWT tokens** for authentication.
  3. Implement **booking form submission**, **booking history fetch**, and **host listing form**.
  4. Test all UI workflows.
  5. Build production-ready **React app** (`npm run build`).
- 

## Day 3: Deployment, Testing, and Report

### Member 1

1. Upload React build to S3.
2. Configure CloudFront distribution for SPA routing.
3. Test frontend + backend integration via CloudFront.

### Member 2

1. Monitor CloudWatch logs for Lambda errors.
2. Ensure API Gateway endpoints are secure & working.
3. Finalize SNS notifications.

### Both Members

1. Conduct **end-to-end testing**:
  - Sign-up/login
  - Search & filter hostels
  - Book a hostel/mess
  - Check booking history
  - Host adds new listing
  - Notifications working
2. Take screenshots for technical report.

### 3. Write summary sections for:

- Architecture diagram
  - Tech stack used
  - Implementation details
  - AWS services used
  - Limitations (AWS Learner Lab)
- 

## Tips to Complete in 3 Days

1. **Keep MVP minimal:** no fancy features like payments or chat.
  2. **Parallel work:** Frontend works with mock API while backend is developed.
  3. **Use pre-built UI libraries:** Tailwind components / Bootstrap.
  4. **Test incrementally:** Don't wait until everything is finished.
  5. **AWS constraints:** Avoid services not in Learner Lab (SES, NLB, NAT Gateway, OpenSearch).
- 

## Suggested Task Timeline

Day	Member 1 (Frontend)	Member 2 (Backend)
1	Init React, routing, mock APIs, UI components	RDS + Cognito setup, Lambda & API Gateway endpoints, Postman test
2	Connect API, handle JWT, implement booking forms & pages	Integrate Lambda with DB + Cognito, SNS, full API testing
3	Deploy build to S3 + CloudFront, test full flows	Monitor CloudWatch, finalize APIs, test notifications

---

If you want, I can also create a visual “3-day timeline Gantt chart” showing tasks for 2 people per day, so you can present it in your technical report.

Do you want me to make that chart?

how to upload our frontend on the s3

Perfect — let's go **step by step** for uploading your **React frontend** to Amazon S3, in a way that works with your **serverless architecture** and **AWS Learner Lab**. I'll assume your React app is already built.

---

## Step 1: Build Your React App

From your project root:

bash

```
npm install # install dependencies
npm run build # create production-ready build
```

 Copy code

- This generates a `build/` folder:

cpp

```
build/
 index.html
 static/
 assets/
```

 Copy code

- Only this folder is uploaded to S3.

## Step 2: Create an S3 Bucket

1. Go to **AWS Console** → **S3** → **Create Bucket**.
2. Bucket configuration:
  - Name: `staymate-frontend` (must be globally unique)
  - Region: same as your backend (e.g., `us-east-1`)
  - **Uncheck** “Block all public access” (required for static website hosting)
3. Click **Create bucket**.

---

## Step 3: Enable Static Website Hosting

1. Go to your bucket → Properties → Static website hosting
  2. Choose:
    - Use this bucket to host a website
    - Index document: `index.html`
    - Error document: `index.html`

Important: This ensures React routing works (SPA fallback)
  3. Save changes.
- 

## Step 4: Upload React Build Files

1. Open your bucket → Upload → Add Files / Add Folder
2. Upload everything inside `build/`, not the folder itself:

```
cpp
```

```
index.html
```

```
static/
```

```
assets/
```

- After upload, S3 structure should look like:

 Copy code

```
cpp
```

```
staymate-frontend
```

```
| index.html
```

```
| static/
```

```
| assets/
```

 Copy code

## Step 5: Set Permissions (Public Read)

1. Go to Permissions → Bucket Policy
2. Add this policy:

```
json
```

```
{
```

```
 "Version": "2012-10-17",
```

```
 "Statement": [
```

```
 {
```

```
 "Sid": "PublicReadGetObject",
```

```
 "Effect": "Allow",
 "Principal": "*",
 "Action": "s3:GetObject",
 "Resource": "arn:aws:s3:::staymate-frontend/*"
 }
]
}
```

3. Save changes → Now files are publicly accessible.

 Copy code

## Step 6: Access Your Site

- S3 website endpoint:

arduino

`http://staymate-frontend.s3-website-<region>.amazonaws.com`

 Copy code

- Test if your **React routing** works (e.g., `/booking`, `/profile`).

 If routing fails, make sure **index.html is both index & error document**.

## Step 7 (Optional but Recommended): CloudFront CDN

For HTTPS + global performance:

1. Go to **CloudFront** → **Create Distribution**
2. Origin Domain: Your **S3 website endpoint**
3. Default Root Object: `index.html`
4. Custom Error Responses:
  - Error Code: `403` and `404`
  - Response Page: `/index.html`
  - HTTP Response Code: `200`

 Now React SPA routing will work over CloudFront with HTTPS.

## Step 8: Connect to Backend

- Make sure your **API endpoints** in React use **CloudFront or API Gateway URLs**.
- Use `.env` file:

env

```
REACT_APP_API_BASE_URL=https://<api-gateway-url>/prod
```

- Rebuild React (`npm run build`) → re-upload to S3.

 Copy code

---

After this, your **frontend is fully deployed** in AWS S3 + CloudFront, integrated with your **serverless backend**.

---

If you want, I can **write an exact AWS Academy Learner Lab-friendly step-by-step guide** with **screenshots + commands**, so your team of 2 can deploy in under **30 minutes**.

Do you want me to do that?



and how to upload our backend on the AWS Lambda

Perfect — let's go **step by step** to deploy your **serverless backend** (Node.js or Python) on **AWS Lambda**, compatible with your **React frontend** and **AWS Learner Lab**.

We'll assume your backend functions are already written and tested locally.

---

## Step 1: Structure Your Lambda Function

Each Lambda function should have:

```
pgsql
```

```
my-function/
├── index.js # main handler
├── package.json # Node dependencies (if Node.js)
└── node_modules/ # installed dependencies (after npm install)
```

 Copy code

## Example: index.js

js

```
exports.handler = async (event) => {
 console.log("Received event:", event);
 return {
 statusCode: 200,
 body: JSON.stringify({ message: "Hello from Lambda!" }),
 };
};
```

 Copy code

## Step 2: Install Dependencies Locally

If your function uses external npm packages:

bash

```
cd my-function
npm install
```

 Copy code

- `node_modules/` folder is created — required for Lambda deployment.

## Step 3: Zip Your Lambda Function

Lambda requires a zip package:

bash

```
cd my-function
zip -r function.zip .
```

 Copy code

- This includes: `index.js`, `package.json`, `node_modules/`, and any helper files.
-  Now `function.zip` is ready for upload.

## Step 4: Create Lambda Function in AWS

1. Go to AWS Console → Lambda → Create Function

2. Choose Author from scratch

- Function name: `StayMate-GetHostels` (use descriptive names)

- Runtime: Node.js 18.x (or Python 3.11)
- Permissions: Choose **Create a new role with basic Lambda permissions**

### 3. Click Create Function

---

## Step 5: Upload Your Code

1. Go to Code → Upload from → .zip file
  2. Select your `function.zip`
  3. Click Deploy
- 

## Step 6: Configure Handler & Environment

- Handler should match your file & function name:

```
pgsql
```

```
index.handler
```

 Copy code

- Add environment variables if needed (like DB URL, Cognito pool ID):

```
DB_HOST, DB_USER, DB_PASS
```

```
COGNITO_POOL_ID, API_KEY
```

 Copy code

---

## Step 7: Connect Lambda to API Gateway

1. Go to API Gateway → Create API → HTTP API (simpler)
2. Add integration:
  - Integration type: Lambda
  - Lambda function: select your function
3. Add route:
  - `/get-hostels` (GET)
4. Deploy API → note the invoke URL:

```
bash
```

```
https://abc123.execute-api.us-east-1.amazonaws.com/get-hostels
```

 Copy code

## Step 8: Test the Lambda

- From API Gateway → Test
- Or Postman / curl:

```
bash
```

```
curl https://abc123.execute-api.us-east-1.amazonaws.com/get-hostels
```

 Copy code

-  Should return JSON from Lambda.

## Step 9: Connect Frontend

In React .env :

```
env
```

```
REACT_APP_API_BASE_URL=https://abc123.execute-api.us-east-1.amazonaws.com
```

 Copy code

- Example fetch in React:

```
js
```

```
fetch(`process.env.REACT_APP_API_BASE_URL}/get-hostels`)
 .then(res => res.json())
 .then(data => console.log(data));
```

 Copy code

- Rebuild React and upload to S3 (Step 1-6 from frontend guide).

## Step 10 (Optional): Logging & Monitoring

- CloudWatch logs are automatically enabled for Lambda.
- Check logs for debugging:
  - AWS Console → Lambda → Your Function → Monitor → View Logs in CloudWatch

### Tips for AWS Learner Lab

1. Use small Lambda functions (memory: 128MB, timeout: 10s) to stay within lab limits.

2. Deploy one function per API endpoint; avoid monolithic functions.
  3. Keep environment variables and secrets minimal.
  4. Test each function independently before connecting to React.
- 

If you want, I can write a full example “StayMate backend deployment package” including:

- Lambda folder structure
- Example API function (hostel listing, booking)
- API Gateway setup
- Environment variables

...ready to upload directly in AWS Learner Lab.

Do you want me to do that?



after uploading our fronted and backend how to connect them

Perfect — now let’s connect your **React frontend** (hosted on **S3 + CloudFront**) with your **serverless backend** (Lambda + API Gateway) so that your app becomes fully functional. I’ll give a **learner-lab-friendly, step-by-step guide**.

---

## Step 1: Confirm Your Backend Endpoints

- Each Lambda function is exposed via API Gateway with an **HTTP(s) endpoint**.
- Example endpoints:

Feature	Endpoint URL	Method
Get hostels	<code>https://abc123.execute-api.us-east-1.amazonaws.com/prod/get-hostels</code>	GET
Create booking	<code>https://abc123.execute-api.us-east-1.amazonaws.com/prod/bookings</code>	POST

Feature	Endpoint URL	Method
Login	<code>https://abc123.execute-api.us-east-1.amazonaws.com/prod/login</code>	POST

Test each endpoint with **Postman** or **curl** to confirm they work before connecting React.

## Step 2: Set API Base URL in React

- In your React project, create a `.env` file (or edit it if exists):

env

```
REACT_APP_API_BASE_URL=https://abc123.execute-api.us-east-1.amazonaws.com/prod
```

 Copy code

- Important:**

- Must start with `REACT_APP_` for `create-react-app` to load it.
- Use `/prod` or whatever your stage name is in API Gateway.

## Step 3: Update React API Calls

In your React app, replace mock URLs with the backend environment variable:

js

```
// Example: fetch hostels
fetch(`${process.env.REACT_APP_API_BASE_URL}/get-hostels`)
 .then(res => res.json())
 .then(data => setHostels(data))
 .catch(err => console.error(err));
```

For **POST** requests (e.g., booking):

 Copy code

js

```
fetch(`${process.env.REACT_APP_API_BASE_URL}/bookings`, {
 method: "POST",
 headers: {
 "Content-Type": "application/json",
 "Authorization": `Bearer ${token}` // if using Cognito JWT
```

```
 },
 body: JSON.stringify({ hostelId: 123, userId: 456 })
})
 .then(res => res.json())
 .then(data => console.log("Booking confirmed:", data));
```

 Copy code

## Step 4: Handle Authentication (Cognito)

- If your backend uses **Cognito JWT** for auth:
  1. After login, get the **ID token** from Cognito.
  2. Pass it in API calls:

js

```
const token = session.getIdToken().getJwtToken();
fetch(` ${API_BASE}/bookings`, {
 headers: { Authorization: `Bearer ${token}` }
});
```

- Lambda functions must **validate the JWT** using Cognito authorizer  Copy code Gateway.

## Step 5: Deploy Updated React App

- After updating `.env` and API calls:

bash

```
npm run build
```

 Copy code

- Upload the new `/build` folder to your **S3 bucket** (overwrite existing files).
- If using CloudFront, **invalidate cache** to load new files:
  - AWS Console → CloudFront → Distribution → **Invalidations** → **Create Invalidation** → `/*`

## Step 6: Test End-to-End

1. Open your **CloudFront URL**:

cpp

<https://<cloudfront-id>.cloudfront.net>

 Copy code

## 2. Test all features:

- Sign up / login
- Browse hostels/mess
- Create bookings
- View booking history
- Host adds listing (if applicable)

## 3. Check CloudWatch logs for Lambda to debug API calls if needed.

---

## Step 7: Optional Enhancements

- CORS Setup:
    - API Gateway must allow requests from your S3/CloudFront domain.
    - Enable CORS in API Gateway → `Access-Control-Allow-Origin` = `*` (or your frontend domain).
  - Error Handling:
    - Frontend should handle failed API calls gracefully (e.g., show error messages).
- 

## Summary Flow

yaml



- React calls API Gateway endpoints → triggers Lambda → interacts with DB or Cognito → returns JSON → React renders it.

 Copy code

If you want, I can **create a ready-to-use integration example**:

- React fetch functions for all endpoints
- JWT Cognito integration
- CORS config in API Gateway

...so your frontend connects to backend immediately in AWS Learner Lab.

Do you want me to do that?



would be postgresql be perfect for the project, since our schema is going to be according to and RDBMS

Yes — PostgreSQL would be an excellent choice for your StayMate project, especially since your schema is relational and your data has structured relationships. Let me explain why it's a good fit and any caveats for AWS Learner Lab.

## Why PostgreSQL is Ideal

Factor	Explanation
Relational Data	You'll have structured tables: <code>users</code> , <code>hostels</code> , <code>bookings</code> , <code>payments</code> . Relationships (foreign keys) exist naturally, e.g., <code>booking.user_id → users.id</code> , <code>booking.hostel_id → hostels.id</code> .
ACID Compliance	Ensures transactional integrity — very important for bookings, payments, and cancellations.
Complex Queries	Supports JOINs, filtering, ordering, aggregations (e.g., "find all bookings for a user", "hostel availability by date").
Indexes & Constraints	You can enforce primary/foreign keys, unique constraints, and use indexes for faster lookups.
Integration with Lambda	AWS Lambda can connect easily via <code>node-postgres</code> (Node.js) or <code>psycopg2</code> (Python).

Factor	Explanation
Mature & Reliable	PostgreSQL is open-source, widely used, and well-supported in AWS RDS.

## Why Not NoSQL (like DynamoDB) for Your Case?

- NoSQL works best for **key-value or document-oriented data**, e.g., caching listings or logs.
- For **booking management with transactions and relationships**, an RDBMS like PostgreSQL is safer and simpler.
- You **do not need eventual consistency**; you want **strong consistency** for bookings.

## AWS Deployment Recommendation

- Use **Amazon RDS (PostgreSQL)** for your database.
- In **AWS Learner Lab**, RDS is supported (small instances only, e.g., `db.t2.micro`).
- Use **private VPC + Lambda access** for security:
  - Lambda functions in the same VPC → can access RDS endpoint.
  - If VPC is complicated in Learner Lab, you can use **publicly accessible RDS** for the project.

## Database Schema Suggestion

### Users Table

sql

```
CREATE TABLE users (
 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 role VARCHAR(10) CHECK (role IN ('customer', 'host')),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

## Hostels Table

 Copy code

sql

```
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 host_id INT REFERENCES users(user_id),
 name VARCHAR(100) NOT NULL,
 location VARCHAR(255),
 description TEXT,
 price_per_month NUMERIC(10,2),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

## Bookings Table

 Copy code

sql

```
CREATE TABLE bookings (
 booking_id SERIAL PRIMARY KEY,
 user_id INT REFERENCES users(user_id),
 hostel_id INT REFERENCES hostels(hostel_id),
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 status VARCHAR(20) CHECK (status IN ('pending','confirmed','cancelled')),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

 This schema shows **foreign keys, constraints, and strong relationships**  Copy code

PostgreSQL handles perfectly.

## Integration with Lambda

- Node.js Example:

js

```
const { Pool } = require('pg');

const pool = new Pool({
 host: process.env.DB_HOST,
 user: process.env.DB_USER,
```

```
password: process.env.DB_PASS,
database: process.env.DB_NAME,
port: 5432
});

exports.handler = async (event) => {
 const res = await pool.query('SELECT * FROM hostels');
 return {
 statusCode: 200,
 body: JSON.stringify(res.rows),
 };
};
```

- Use **environment variables** for DB credentials.
- Lambda connects directly to RDS using **node-postgres** or **Python psycopg2**.

 Copy code

## Conclusion

- **PostgreSQL** is perfect for StayMate because of:
  - Relational data
  - Transaction support
  - Complex queries (joins, filters)
  - Reliability
- In AWS Learner Lab:
  - Use **RDS PostgreSQL** with micro instance (`db.t2.micro`)
  - Ensure Lambda can connect (VPC/public-access)

If you want, I can **also map out the full backend table relationships** for StayMate in PostgreSQL, including **users, hostels, bookings, payments, and reviews**, ready for **AWS RDS deployment**.

Do you want me to do that?



how to connect backend with rds

Perfect — connecting your AWS Lambda backend to Amazon RDS (PostgreSQL) is a key step for StayMate. I'll give a step-by-step, AWS Learner Lab-friendly guide, including network, credentials, and code setup.

---

## Step 1: Create Your RDS PostgreSQL Database

### 1. Go to AWS Console → RDS → Create Database

- Engine: PostgreSQL
- Template: Free tier / Dev/Test
- DB instance identifier: staymate-db
- Master username/password: admin / yourpassword
- DB instance size: db.t2.micro (small for learner lab)
- Public accessibility: Yes (for simplicity in learner lab)
- VPC: default or lab-provided

### 2. Click Create Database

- Wait a few minutes for RDS to be available.
- 

## Step 2: Note Your RDS Endpoint

- In RDS → Databases → staymate-db → Connectivity & security
  - Copy endpoint (e.g., staymate-db.c123abcxyz.us-east-1.rds.amazonaws.com )
  - Note port: 5432
- 

## Step 3: Configure Security Group

- Your Lambda needs to reach the RDS instance:

### 1. Go to EC2 → Security Groups → Default / RDS Security Group

### 2. Add Inbound Rule:

- Type: PostgreSQL
- Protocol: TCP
- Port Range: 5432
- Source: 0.0.0.0/0 (or Lambda's security group if in VPC)

### 3. Save rules

- In learner lab, using 0.0.0.0/0 is okay for testing.

## Step 4: Add Environment Variables to Lambda

- Go to Lambda → Your Function → Configuration → Environment Variables
- Add:

Key	Value
DB_HOST	staymate-db.c123abxyz.us-east-1.rds.amazonaws.com
DB_USER	admin
DB_PASS	yourpassword
DB_NAME	postgres (or the database you created)
DB_PORT	5432

## Step 5: Install PostgreSQL Library in Lambda

- For Node.js, use `pg`:

```
bash

cd my-lambda-function
npm init -y
npm install pg
```

- Zip all files including `node_modules` for Lambda deployment.

 Copy code

## Step 6: Connect Lambda to RDS in Code

### Node.js Example

```
js

const { Pool } = require('pg');

const pool = new Pool({
 host: process.env.DB_HOST,
 user: process.env.DB_USER,
 password: process.env.DB_PASS,
```

```
database: process.env.DB_NAME,
port: process.env.DB_PORT
});

exports.handler = async (event) => {
 try {
 const res = await pool.query('SELECT * FROM hostels');
 return {
 statusCode: 200,
 body: JSON.stringify(res.rows)
 };
 } catch (err) {
 console.error(err);
 return { statusCode: 500, body: 'Database query failed' };
 }
};
```

## Python Example

```
python

import psycopg2
import os
import json

def lambda_handler(event, context):
 conn = psycopg2.connect(
 host=os.environ['DB_HOST'],
 user=os.environ['DB_USER'],
 password=os.environ['DB_PASS'],
 dbname=os.environ['DB_NAME'],
 port=os.environ['DB_PORT']
)
 cur = conn.cursor()
 cur.execute("SELECT * FROM hostels")
 rows = cur.fetchall()
 cur.close()
 conn.close()
 return {
 'statusCode': 200,
```

```
'body': json.dumps(rows)
```

```
}
```

 Copy code

## Step 7: Test Lambda Function

- Go to Lambda → Test → Create Test Event
- Trigger Lambda → Check response
- Check CloudWatch logs for errors

 If successful, your Lambda can now query PostgreSQL.

## Step 8: Connect to Frontend

- Update React `.env` to point API calls to **API Gateway endpoints** linked to Lambda:

env

```
REACT_APP_API_BASE_URL=https://abc123.execute-api.us-east-1.amazonaws.com/prod
```

 Copy code

- Rebuild React → Upload to S3

Your frontend → API Gateway → Lambda → RDS workflow is now complete.

## Optional: VPC Setup (Secure Option)

- If public access is **disabled**, you must:
  1. Place Lambda in **same VPC & private subnet** as RDS.
  2. Add **NAT Gateway / Internet Gateway** if Lambda needs internet access.
- In AWS Learner Lab, using **publicly accessible RDS** is simpler.

 Summary Flow

markdown

```
React (S3/CloudFront)
```

```
|
```

```
HTTPS API calls
```

API Gateway → Lambda → PostgreSQL (RDS)

 Copy code

If you want, I can draw a full AWS Learner Lab diagram showing frontend S3 → API Gateway → Lambda → RDS PostgreSQL, ready for your **technical report**.

Do you want me to make that diagram?



for future explanations, we would be using React, node, express, bootstrap, html, css for the code part

Got it! Noted for all future explanations: your **code stack** will be:

- **Frontend:** React.js + Bootstrap + HTML + CSS
- **Backend:** Node.js + Express.js

Everything will assume this stack for examples, deployments, API integrations, and project guidance.

This means I'll focus on:

- **React SPA** for the UI
- **Bootstrap** for styling
- **Express.js** for backend routes & API endpoints
- **Node.js** as runtime for serverless (or local testing)

Everything else (AWS Lambda, API Gateway, RDS, S3) will integrate with this stack.

Do you want me to show **how to structure a full React + Node + Express backend for StayMate** so it's ready for **AWS deployment**?



now for this tech stack explain how to upload everything

Perfect — since your stack is now **React + Node + Express + Bootstrap/HTML/CSS**, I'll explain **how to upload and deploy everything** in a **serverless AWS setup** (Learner Lab-friendly) step by step. I'll cover both **frontend and backend**, and the **end-to-end workflow**.

## 1 Frontend Deployment (React + Bootstrap)

This will be hosted on **S3 + CloudFront** as a **static site**.

### Step 1: Build the React App

bash

```
cd staymate-frontend
npm install
npm run build
```

- This creates a `/build` folder with:

 Copy code

```
build/
 index.html
 static/
 assets/
```

 Copy code

### Step 2: Create an S3 Bucket

1. AWS Console → **S3** → **Create Bucket**
2. Bucket name: `staymate-frontend` (globally unique)
3. Region: same as your backend (for simplicity)
4. Uncheck "Block all public access"
5. Click **Create bucket**

## Step 3: Enable Static Website Hosting

1. Go to Properties → Static website hosting
  2. Select:
    - Use this bucket to host a website
    - Index document: `index.html`
    - Error document: `index.html` (important for SPA routing)
- 

## Step 4: Upload Build Files

1. Open bucket → Upload → Add Files / Add Folder
2. Upload everything inside `build/`, not the folder itself
3. Set public read permissions (or use a bucket policy):

```
json

{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Sid": "PublicRead",
 "Effect": "Allow",
 "Principal": "*",
 "Action": "s3:GetObject",
 "Resource": "arn:aws:s3:::staymate-frontend/*"
 }
]
}
```

## Step 5: (Optional) CloudFront for CDN + HTTPS

 Copy code

1. AWS Console → CloudFront → Create Distribution
  2. Origin: your S3 bucket
  3. Default Root Object: `index.html`
  4. Add error responses:
    - Error code `403` and `404` → `/index.html` → `200`
  5. Deploy → Use CloudFront URL for your frontend
-

## 2 Backend Deployment (Node.js + Express)

Your Node + Express backend will run on AWS Lambda + API Gateway.

### Step 1: Structure Lambda Function

Example folder: `backend-get-hostels`

```
pgsql

backend-get-hostels/
 ├── index.js # Lambda handler
 ├── package.json
 └── node_modules/
```

`index.js` example:

 Copy code

```
js

const express = require('express');
const serverless = require('serverless-http');
const { Pool } = require('pg');

const app = express();
app.use(express.json());

const pool = new Pool({
 host: process.env.DB_HOST,
 user: process.env.DB_USER,
 password: process.env.DB_PASS,
 database: process.env.DB_NAME,
 port: process.env.DB_PORT
});

app.get('/hostels', async (req, res) => {
 const result = await pool.query('SELECT * FROM hostels');
 res.json(result.rows);
});

module.exports.handler = serverless(app);
```

**Note:** `serverless-http` converts Express into a Lambda handler.

## Step 2: Install Dependencies

 Copy code

bash

```
npm init -y
npm install express serverless-http pg
```

 Copy code

---

## Step 3: Zip Lambda Function

bash

```
zip -r function.zip .
```

 Copy code

- Include all code and node\_modules.
- 

## Step 4: Create Lambda Function

1. AWS Console → Lambda → Create Function
  2. Author from scratch:
    - Name: StayMate-GetHostels
    - Runtime: Node.js 18.x
    - Permissions: Basic Lambda Execution
  3. Upload function.zip
  4. Handler: index.handler
- 

## Step 5: Add Environment Variables

Key	Value
DB_HOST	<RDS endpoint>
DB_USER	<username>
DB_PASS	<password>
DB_NAME	<database>
DB_PORT	5432

## Step 6: Create API Gateway

1. API Gateway → HTTP API → Integrate Lambda
2. Add route: /hostels → GET
3. Deploy API → note invoke URL

```
bash
```

```
https://abc123.execute-api.us-east-1.amazonaws.com/hostels
```

 Copy code

## Step 7: Connect Lambda → RDS

- Already done in index.js using pg and env variables.
- Test Lambda → CloudWatch logs → check DB queries work

## 3 Connect Frontend → Backend

1. In React .env :

```
env
```

```
REACT_APP_API_BASE_URL=https://abc123.execute-api.us-east-1.amazonaws.com
```

 Copy code

2. Update fetch calls in React:

```
js
```

```
fetch(`process.env.REACT_APP_API_BASE_URL}/hostels`)
 .then(res => res.json())
 .then(data => console.log(data));
```

 Copy code

3. Rebuild React (npm run build) → Upload to S3 → CloudFront

## 4 Optional Enhancements

- CORS: Enable in API Gateway so React domain can call Lambda:
  - Access-Control-Allow-Origin: \* (or your CloudFront URL)
- Authentication: Use Cognito for login/registration
- Monitoring: CloudWatch logs for Lambda

- SNS notifications for booking confirmations
- 

## 5 Summary Flow

java

React SPA (CloudFront/S3)

|

| HTTPS API calls

v

API Gateway → Lambda (Node + Express) → PostgreSQL (RDS)

- Frontend calls API → Lambda executes Express route → queries Pos Copy code  
returns JSON → React renders it
- 

I can also create a **ready-to-deploy folder structure** for StayMate, with:

- React frontend
- Node + Express backend
- Serverless Lambda ready zip
- `.env` examples

...so your team can upload everything in 30–60 minutes in AWS Learner Lab.

Do you want me to make that?



are `\build` and `\dist` the same, bcz i get the `dist` folder

Ah — this is a very common point of confusion! No, `/build` and `/dist` are not exactly the same, but they serve a similar purpose: both are **production-ready folders containing compiled/transpiled frontend code**. The difference depends on the **build tool** you are using.

Let me break it down:

## 1 React (create-react-app)

- Default command:

```
bash
```

```
npm run build
```

 Copy code

- Output: `build/` folder
- Contents:

```
swift
```

```
build/
 index.html
 static/js/...
 static/css/...
```

- Purpose: This is production-ready code ready to upload to S3 / CloudFront  Copy code

## 2 Other Build Tools (Vite, Webpack, Parcel)

- Tools like Vite or Webpack often generate `dist/` instead of `build/`.
- Example with Vite:

```
bash
```

```
npm run build
```

 Copy code

- Output: `dist/` folder
- Contents:

```
pgsql
```

```
dist/
 index.html
 assets/
```

- Purpose: Same as `build/` — production-ready files.

 Copy code

 In short: `dist/` in Vite = `build/` in CRA. You can upload the contents of `dist/` to S3, just like you would with `build/`.

### 3 What to Upload to S3

- Do not upload the folder itself, upload the contents:

cpp

index.html

assets/

static/

- This ensures S3/CloudFront can serve your SPA correctly.

 Copy code

### Rule of Thumb

Build Tool	Production Folder	Upload to S3 as
Create React App	build/	contents of build/
Vite / Webpack	dist/	contents of dist/
Angular / Vue	dist/	contents of dist/

So, if you're seeing `dist/` instead of `build/`, no worries — treat it the same way when uploading to S3 + CloudFront.

If you want, I can give a step-by-step guide specifically for uploading a `dist/` folder from React/Vite to S3, including CloudFront SPA routing setup, so you don't hit any 404 errors.

Do you want me to do that?



how to use the query editor on our rds

Perfect — let's go **step by step** for using the **Query Editor** on your RDS (PostgreSQL) instance in AWS, so you can create tables, insert data, and test queries without any external client. This is especially handy in AWS Learner Lab.

---

## Step 1: Go to the Query Editor

1. AWS Console → RDS → Databases
2. Click your PostgreSQL database instance
3. Click **Query Editor** in the left menu
4. If it's your first time:
  - You may need to **enable IAM authentication** and set **permissions**.
  - **Ensure your IAM user has the policy:** `AmazonRDSDataFullAccess`

In Learner Lab, your default user usually has this.

---

## Step 2: Connect to Your Database

1. Database: select your RDS instance
  2. Database name: the name you created (`postgres` by default)
  3. Authentication:
    - **Use database credentials** (username/password you set on RDS)
  4. Click **Connect**
-  You should see “Connected” at the top of Query Editor.
- 

## Step 3: Run SQL Queries

You can now type queries directly into the editor.

### Example: Create Users Table

```
sql
```

```
CREATE TABLE users (
 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 role VARCHAR(10) CHECK (role IN ('customer', 'host')),
```

```
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

 Copy code

## Example: Insert a User

sql

```
INSERT INTO users (full_name, email, password_hash, role)
VALUES ('John Doe', 'john@example.com', 'hashedpassword123', 'customer');
```

 Copy code

## Example: Select Users

sql

```
SELECT * FROM users;
```

 Copy code

- Click Run → results appear below the editor
- You can also download query results as CSV for testing.

## Step 4: Create Other Tables (Hostels & Bookings)

sql

```
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 host_id INT REFERENCES users(user_id),
 name VARCHAR(100) NOT NULL,
 location VARCHAR(255),
 description TEXT,
 price_per_month NUMERIC(10,2),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE TABLE bookings (
 booking_id SERIAL PRIMARY KEY,
 user_id INT REFERENCES users(user_id),
 hostel_id INT REFERENCES hostels(hostel_id),
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 status VARCHAR(20) CHECK (status IN ('pending', 'confirmed', 'cancelled')),
```

```
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

 Copy code

## Step 5: Test Your Data

- Insert some sample rows into **hostels** and **bookings**
- Use **SELECT** queries to verify

sql

```
SELECT * FROM hostels;
SELECT * FROM bookings;
```

 Copy code

## Step 6: Connect Lambda / Backend to RDS

- Use the **same database credentials** in Lambda environment variables:

Env Variable	Value
DB_HOST	your RDS endpoint
DB_USER	your username
DB_PASS	your password
DB_NAME	database name
DB_PORT	5432

- Test Lambda → API Gateway → confirm queries work from backend.

## Tips

1. Always check table relationships using **FOREIGN KEY**s.
2. Use **ROLLBACK** / **COMMIT** if running transactions in Query Editor.
3. You can **copy queries from Query Editor** to backend code when writing Lambda functions.
4. If you get connection errors:
  - Make sure **RDS security group allows inbound traffic** (port 5432)

- Use publicly accessible DB in Learner Lab.

## Summary Flow

graphql

Query Editor (AWS Console)

|

v

RDS PostgreSQL

|

Lambda Backend → API Gateway → React Frontend

 Copy code

If you want, I can write a full “sample dataset + queries” for StayMate, including `users`, `hostels`, `bookings`, ready to paste into RDS Query Editor for testing your backend immediately.

Do you want me to do that?



why does the query editor say that this db doesn't have the access

Ah — this happens often in AWS RDS + Query Editor, especially in AWS Learner Lab. It's not a bug; it's usually permission / authentication / connectivity related. Let's troubleshoot carefully.

## Possible Reasons & Fixes

### 1 IAM Permissions Not Set

- The Query Editor uses IAM authentication, not just username/password.
- If your IAM user doesn't have the right policy, you'll get an access error.

Fix:

1. Go to IAM → Users → Your User

2. Attach policy:

nginx

AmazonRDSDataFullAccess

 Copy code

3. Reconnect in Query Editor.

In Learner Lab, the default student IAM user may already have limited permissions. Make sure it includes RDS Query Editor access.

---

## 2 Database Not Using IAM Authentication

- Query Editor only works if **IAM authentication is enabled** on your RDS database.

Check:

1. RDS → Databases → Select DB → **Connectivity & Security**

2. Look for **IAM DB Authentication: Enabled?**

- If **No**, Query Editor can't connect using IAM
- You can:
  - Enable IAM authentication (advanced)
  - Or use a **public client** like pgAdmin / DBeaver / psql to test your DB

In Learner Lab, sometimes enabling IAM auth isn't possible. Using **username/password in code** is easier for Lambda integration.

---

## 3 Database is Not Publicly Accessible

- Query Editor connects from AWS Console. If your RDS is **not publicly accessible**, Query Editor may fail.

Fix:

1. RDS → Modify → **Publicly Accessible** → Yes

2. Make sure **Security Group** allows inbound access from the Query Editor (or 0.0.0.0/0 for simplicity in lab)

3. Save → wait for DB to apply changes

## 4 Wrong Database / Credentials

- Make sure in Query Editor:
    - Database name matches the one you created (default is `postgres`)
    - Username matches RDS master username
    - Password is correct
- 

## 5 Learner Lab Limitation

- AWS Learner Lab sometimes restricts access to **Query Editor** for student accounts
- Alternative:
  - Use **pgAdmin**, **DBeaver**, or **psql CLI** to connect:

php-template

```
psql -h <rds-endpoint> -U <username> -d <database>
```

 Copy code

- Works with **username/password** without IAM
- 

## ✓ Recommended Workaround

If Query Editor keeps giving access errors:

1. Keep your RDS publicly accessible
  2. Use **username/password** in Lambda and local DB clients
  3. Skip Query Editor — just use **pgAdmin** / **DBeaver** / **psql** for testing
  4. For your **project deployment**, Lambda → RDS works **without Query Editor**, as long as the credentials and security group are correct
- 

If you want, I can make a quick step-by-step “connect to RDS using pgAdmin / psql” that works perfectly in **AWS Learner Lab**, so you can **create tables and test your backend immediately**, bypassing Query Editor issues.

Do you want me to do that?

Perfect — using **DBeaver** is a reliable way to connect to your **RDS PostgreSQL** in AWS Learner Lab, especially if the **Query Editor** is giving access errors. I'll give you a **step-by-step guide**.

---

## Step 1: Install DBeaver

1. Go to <https://dbeaver.io/> ↗
  2. Download **DBeaver Community Edition** (free)
  3. Install it on your local machine
- 

## Step 2: Get Your RDS Details

From AWS Console → RDS → Databases → Your PostgreSQL instance:

- **Endpoint:** e.g., `staymate-db.c123abxyz.us-east-1.rds.amazonaws.com`
- **Port:** `5432`
- **Database name:** `postgres` (or the one you created)
- **Username:** RDS master username
- **Password:** your password

Also ensure **Public Accessibility = Yes** and **security group allows inbound 5432**.

---

## Step 3: Open DBeaver and Create New Connection

1. Open DBeaver → Database → New Database Connection
2. Select PostgreSQL → Next
3. Fill in the connection details:

Field	Value
Host	<code>staymate-db.c123abxyz.us-east-1.rds.amazonaws.com</code>
Port	<code>5432</code>
Database	<code>postgres</code>

Field	Value
Username	admin
Password	<your password>

4. Optional: click **Test Connection** → Should show **success**
- 

## Step 4: Connect

- Click **Finish**
  - DBeaver will now show your RDS database in the Database Navigator
  - Expand it → see **Schemas, Tables, Views, etc.**
- 

## Step 5: Create Tables / Run Queries

1. Right-click your database → **SQL Editor** → **New SQL Script**
2. Paste your schema, for example:

```
sql
```

```
CREATE TABLE users (
 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 role VARCHAR(10) CHECK (role IN ('customer', 'host')),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 host_id INT REFERENCES users(user_id),
 name VARCHAR(100) NOT NULL,
 location VARCHAR(255),
 description TEXT,
 price_per_month NUMERIC(10,2),
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
CREATE TABLE bookings (
```

```
booking_id SERIAL PRIMARY KEY,
user_id INT REFERENCES users(user_id),
hostel_id INT REFERENCES hostels(hostel_id),
start_date DATE NOT NULL,
end_date DATE NOT NULL,
status VARCHAR(20) CHECK (status IN ('pending', 'confirmed', 'cancelled')),
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. Click Execute SQL → tables are created

## Step 6: Test Data

Insert sample data:

Copy code

```
sql

INSERT INTO users (full_name, email, password_hash, role)
VALUES ('Alice', 'alice@example.com', 'hashed123', 'customer');

INSERT INTO hostels (hostel_id, name, location, price_per_month)
VALUES (1, 'Sunrise Hostel', 'Downtown', 12000);

INSERT INTO bookings (user_id, hostel_id, start_date, end_date, status)
VALUES (1, 1, '2026-01-10', '2026-02-10', 'confirmed');
```

Select data to verify:

Copy code

```
sql

SELECT * FROM users;
SELECT * FROM hostels;
SELECT * FROM bookings;
```

Copy code

## Step 7: Connect Lambda / Backend

- Use same credentials in your Node.js + Express Lambda functions:

```
js
```

```
const { Pool } = require('pg');

const pool = new Pool({
 host: '<RDS endpoint>',
 user: '<username>',
 password: '<password>',
 database: 'postgres',
 port: 5432
});
```

 Copy code

- Test API Gateway → Lambda → RDS connection
- 

## Tips for AWS Learner Lab

1. Make sure RDS is **publicly accessible** (simpler for lab)
  2. Security group: allow inbound **5432 from your local IP**
  3. Use DBeaver for:
    - Table creation
    - Data insertion
    - Testing queries
  4. Lambda connects **directly to RDS**; DBeaver is just for local testing/management
- 

If you want, I can make a **full “StayMate database setup” ready SQL script** you can paste into **DBeaver** to create **users, hostels, bookings, payments, and reviews tables**, with sample data, ready to connect to your backend.

Do you want me to do that?



how will we handle images since we can't upload those in the RDS

Excellent question — this is **exactly how real-world systems handle images**   
You're right: **images should NEVER be stored in RDS**. Instead, we use **S3 + URLs**.

Since your stack is **React + Node + Express + AWS (Serverless)**, here's the **clean, industry-standard approach**.

---

## ✓ Correct Way to Handle Images (AWS Best Practice)

### Golden Rule

Store images in Amazon S3

Store only the image URL in RDS

---

### High-Level Flow

java

React Frontend

|

| (image upload)

v

API Gateway

|

v

Lambda (Node + Express)

|

| --> Upload image to S3

| --> Get **public/private** URL

|

v

RDS (PostgreSQL)

|

| --> Save **image\_url** (TEXT)

### 1 Create an S3 Bucket for Images

 Copy code

- Example bucket name:

staymate-images

 Copy code

- Settings:
    - Block public access → **Disabled** (or use CloudFront)
    - Enable versioning (optional)
    - Region: same as Lambda
- 

## 2 Database Design (RDS)

You only store the image URL.

Example: `hostels` table

sql

```
ALTER TABLE hostels
ADD COLUMN image_url TEXT;
```

 Copy code

Or if multiple images:

sql

```
CREATE TABLE hostel_images (
 image_id SERIAL PRIMARY KEY,
 hostel_id INT REFERENCES hostels(hostel_id),
 image_url TEXT NOT NULL
);
```

 Copy code

## 3 Backend (Lambda) – Upload Image to S3

Install dependencies

bash

```
npm install aws-sdk multer multer-s3
```

 Copy code

Lambda Upload Logic (Node + Express)

js

```
const AWS = require('aws-sdk');
const multer = require('multer');
```

```

const multerS3 = require('multer-s3');

const s3 = new AWS.S3();

const upload = multer({
 storage: multerS3({
 s3,
 bucket: 'staymate-images',
 key: (req, file, cb) => {
 cb(null, `hostels/${Date.now()}-${file.originalname}`);
 }
 })
});

app.post('/upload', upload.single('image'), async (req, res) => {
 const imageUrl = req.file.location;

 // Save imageUrl in RDS
 await pool.query(
 'UPDATE hostels SET image_url = $1 WHERE hostel_id = $2',
 [imageUrl, req.body.hostel_id]
);

 res.json({ imageUrl });
});

```

## 4 IAM Permissions (IMPORTANT)

Your Lambda must have permission to upload to S3.

 Copy code

Attach this policy to Lambda role:

json

```
{
 "Effect": "Allow",
 "Action": [
 "s3:PutObject",
 "s3:GetObject"
],
}
```

```
"Resource": "arn:aws:s3:::staymate-images/*"
}
```

 Copy code

## 5 Frontend (React) – Image Upload

### React Image Upload Example

js

```
const uploadImage = async (file, hostelId) => {
 const formData = new FormData();
 formData.append('image', file);
 formData.append('hostel_id', hostelId);

 await fetch(API_URL + '/upload', {
 method: 'POST',
 body: formData
 });
};
```

 Copy code

### Display Image

jsx

```

```

 Copy code

## 6 Public vs Private Images

### Option A: Public Images (Simplest – Good for Student Projects)

- Bucket is public
- Direct S3 URLs used

### Option B: Private Images (Advanced)

- Images are private
- Backend generates pre-signed URLs

js

```
const url = s3.getSignedUrl('getObject', {
 Bucket: 'staymate-images',
 Key: 'hostels/image.jpg',
 Expires: 300
});
```

 Copy code

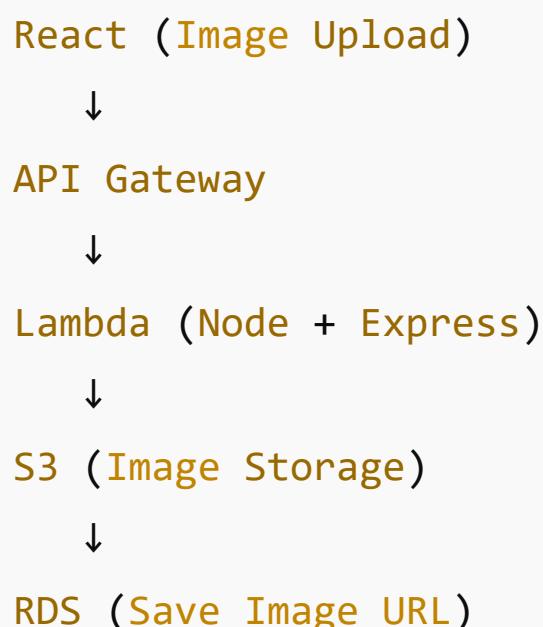
## 7 Why This Is the Best Design

Problem	Solution
RDS storage expensive	S3 is cheap
Database slow	Images served via CDN
Backup issues	S3 handles durability
Scaling	S3 scales automatically

This is exactly how Airbnb, Booking.com, Uber handle images.

## Final Architecture Summary

mathematica



 Copy code

## For Your University Evaluation

You can confidently say:

*"Images are stored in Amazon S3, while their URLs are stored in PostgreSQL (RDS), following AWS best practices for scalability, cost efficiency, and performance."*

If you want, next I can:

- Design image upload API properly for serverless
- Show CloudFront + S3 for faster image loading
- Help you write this perfectly for your technical report

Just tell me 



give the steps to step up the lambda function for it

Perfect — I'll give you **clear, step-by-step instructions** to set up an **AWS Lambda function** that **uploads images to S3 and stores the URL in RDS**, using your agreed stack:

**Node.js + Express + PostgreSQL (RDS) + S3 + API Gateway**

This is written **exactly how you should do it** in AWS Learner Lab.

## STEP-BY-STEP: SET UP LAMBDA FOR IMAGE UPLOAD

### STEP 1: Create IAM Role for Lambda

1. Go to IAM → Roles → Create role
2. Trusted entity: AWS service
3. Use case: Lambda
4. Attach these policies:
  - AWSLambdaBasicExecutionRole
  - AmazonS3FullAccess (*for lab simplicity*)
5. Name role:

```
python
```

```
staymate-lambda-role
```

 Copy code

This allows:

- CloudWatch logs
- Uploading images to S3

## STEP 2: Create the Lambda Function

1. Go to AWS Lambda → Create function

2. Choose:

- Author from scratch

3. Function name:

```
arduino
```

```
staymate-upload-image
```

 Copy code

4. Runtime:

```
Node.js 18.x
```

 Copy code

5. Execution role:

- Use existing role
- Select: `staymate-lambda-role`

6. Click Create function

## STEP 3: Increase Lambda Limits

Image uploads need more resources.

1. Open the Lambda function
2. Configuration → General configuration → Edit
3. Set:
  - Timeout: `30 seconds`
  - Memory: `1024 MB`
4. Save

## STEP 4: Add Environment Variables

Go to Configuration → Environment variables → Add:

Key	Value
DB_HOST	<your-rds-endpoint>
DB_USER	<db-username>
DB_PASS	<db-password>
DB_NAME	postgres
DB_PORT	5432
S3_BUCKET	staymate-images

## STEP 5: Prepare Backend Code (Node + Express)

### Project Structure

```
pgsql
lambda/
 └── index.js
 └── package.json
```

 Copy code

### package.json

```
json

{
 "name": "staymate-upload",
 "version": "1.0.0",
 "dependencies": {
 "aws-sdk": "^2.1470.0",
 "express": "^4.18.2",
 "multer": "^1.4.5",
 "multer-s3": "^2.10.0",
 }
}
```

```
 "pg": "^8.11.0",
 "serverless-http": "^3.2.0"
 }
}
```

Install locally:

bash

```
npm install
```

 Copy code

## STEP 6: Lambda Code (index.js)

js

```
const express = require('express');
const AWS = require('aws-sdk');
const multer = require('multer');
const multerS3 = require('multer-s3');
const serverless = require('serverless-http');
const { Pool } = require('pg');

const app = express();
app.use(express.json());

const s3 = new AWS.S3();

const pool = new Pool({
 host: process.env.DB_HOST,
 user: process.env.DB_USER,
 password: process.env.DB_PASS,
 database: process.env.DB_NAME,
 port: process.env.DB_PORT
});

const upload = multer({
 storage: multerS3({
 s3,
 bucket: process.env.S3_BUCKET,
 contentType: multerS3.AUTO_CONTENT_TYPE,
 key: (req, file, cb) => {

```

```

 cb(null, `hostels/${Date.now()}-${file.originalname}`);
}
});
});

app.post('/upload', upload.single('image'), async (req, res) => {
try {
const imageUrl = req.file.location;
const hostelId = req.body.hostel_id;

await pool.query(
'UPDATE hostels SET image_url = $1 WHERE hostel_id = $2',
[imageUrl, hostelId]
);

res.status(200).json({ imageUrl });
} catch (err) {
console.error(err);
res.status(500).json({ error: 'Upload failed' });
}
});
}

module.exports.handler = serverless(app);

```

## STEP 7: Zip and Upload Code

1. Zip everything inside the folder:

```

pgsql

index.js
node_modules/
package.json

```

Copy code

2. Go to Lambda → Code → Upload from → .zip file
3. Upload ZIP → Save

## STEP 8: Connect Lambda to API Gateway

1. Lambda → Add trigger
2. Select API Gateway

### 3. Choose:

- HTTP API
- Security: Open

### 4. Create API

You'll get an endpoint like:

bash

<https://abc123.execute-api.us-east-1.amazonaws.com/upload>

 Copy code

---

## STEP 9: Enable Binary Media Types (IMPORTANT)

API Gateway → Settings:

- Add:

bash

multipart/form-data

 Copy code

Without this, image uploads **will fail**.

---

## STEP 10: Test the API

### Using Postman

- Method: `POST`
- URL: `/upload`
- Body → `form-data` :
  - `image` → file
  - `hostel_id` → number

 Image uploaded to S3

 Image URL saved in RDS

---

## FINAL FLOW

SCSS

React (FormData)

↓

API Gateway

↓

Lambda (Node + Express)

↓

S3 (Image Storage)

↓

RDS (Save URL)

 Copy code

## What You Can Say in Viva / Report

*"We implemented image handling using AWS Lambda integrated with Amazon S3 for storage and PostgreSQL (RDS) for metadata. Images are uploaded through API Gateway using multipart/form-data and securely stored in S3, while URLs are persisted in the database."*

If you want, next I can:

- Add authentication with Cognito
- Optimize image upload using pre-signed URLs
- Draw architecture diagram
- Write LaTeX explanation for this Lambda setup

Just tell me 

      ...

now do we have to setup 2 lambda func, or all this can done in a single one

Great question — this is **architecture-level thinking**, and the answer depends on **project scale vs. best practice**. I'll give you a **clear verdict first**, then explain **both approaches**, and finally tell you **what you should do for your project**.

## Short Answer (Verdict)

- 👉 You CAN do everything in ONE Lambda function,
- 👉 but the RECOMMENDED approach is MULTIPLE Lambdas.

For a university project / 3-day timeline / AWS Learner Lab:

| Use 1 Lambda initially, split later if needed

## OPTION 1 : Single Lambda Function (Allowed & Practical)

### How it works

One Lambda handles:

- User APIs
- Hostel APIs
- Booking APIs
- Image uploads
- Database access

### Example routes in ONE Lambda:

js

```
POST /login
POST /signup
POST /hostels
POST /upload-image
GET /hostels
```

 Copy code

### Pros

-  Fast to develop
-  Easy to debug
-  Fewer AWS resources
-  Perfect for demos & evaluation

### Cons

-  Less “pure” microservices
-  Larger Lambda package

## Best use case

- ✓ Student projects
  - ✓ Tight deadlines
  - ✓ AWS Learner Lab limits
- 

## OPTION 2 : Multiple Lambda Functions (Best Practice)

### Recommended Separation

Lambda	Responsibility
auth-lambda	Login / Signup
hostel-lambda	Hostels CRUD
booking-lambda	Booking logic
image-lambda	Image upload to S3

### Each Lambda:

- Has its own API Gateway route
- Has minimal permissions
- Scales independently

### Pros

- ✓ Clean architecture
- ✓ Better scalability
- ✓ Least privilege IAM
- ✓ Industry-standard

### Cons

- ✗ More setup
  - ✗ More configuration
  - ✗ Slower to build in 3 days
- 

## What AWS & Professors Expect

They do NOT expect a full microservice split for a semester project.

They want to see:

- ✓ Serverless usage
- ✓ API Gateway → Lambda
- ✓ S3 + RDS integration
- ✓ Clean explanation

One Lambda is totally acceptable.

---

## 🏆 Best Compromise (Recommended)

Phase 1 (Now – 3 days)

👉 1 Lambda function

- Node + Express
- Multiple routes
- Connected to RDS + S3

Phase 2 (Report / Viva Explanation)

👉 Explain that:

*"The system is designed to be modular, and individual functionalities can be migrated into separate Lambda functions as the platform scales."*

This scores architecture marks without extra work.

---

## 🔥 Recommended Lambda Structure (Single Lambda)

pgsql

```
lambda/
 └── index.js
 └── routes/
 ├── auth.js
 ├── hostels.js
 ├── bookings.js
 └── upload.js
 └── db.js
 └── s3.js
 └── package.json
```

This looks professional, even though it's one Lambda.

 Copy code

## 📌 Final Recommendation (For YOU)

- ✓ Use ONE Lambda function
- ✓ Use Express routes inside it
- ✓ Mention future scalability with multiple Lambdas in report
- ✓ Focus on functionality + clean demo

---

If you want, next I can:

- Refactor your single Lambda into clean Express routes
- Design API Gateway routing
- Write a perfect justification paragraph for your report
- Show how to split into multiple Lambdas later

Just tell me what you want next 

     ...

give the steps to edit an existing lambda func and give it the permissions

Perfect — this is a must-know AWS skill. I'll give you clear, exact steps to edit an existing Lambda function and grant it permissions, written for AWS Learner Lab and your Node + Express + S3 + RDS setup.

---

## 🛠 PART A: How to EDIT an Existing Lambda Function

### STEP 1: Open the Lambda Function

1. AWS Console → Lambda
2. Click Functions

### 3. Select your existing function

(e.g. staymate-backend)

---

## STEP 2: Edit the Code

### Option A: Edit Inline (Small Changes)

1. Lambda → Code tab
2. Edit index.js directly in the editor
3. Click Deploy (VERY IMPORTANT)

 Inline editing is only recommended for small fixes

---

### Option B: Upload Updated ZIP (Recommended)

1. Update code locally
2. Make sure:

```
pgsql

index.js
node_modules/
package.json
```

are all present

 Copy code

3. Zip everything (not the folder)
  4. Lambda → Code → Upload from → .zip file
  5. Upload ZIP
  6. Click Deploy
- 

## STEP 3: Update Environment Variables

1. Lambda → Configuration → Environment variables
2. Click Edit
3. Add / update:

```
nginx
```

```
DB_HOST
```

```
DB_USER
```

```
DB_PASS
DB_NAME
DB_PORT
S3_BUCKET
```

#### 4. Click Save

 Copy code

---

## STEP 4: Increase Timeout & Memory (If Needed)

1. Lambda → Configuration → General configuration
  2. Click Edit
  3. Set:
    - Timeout: 30 sec
    - Memory: 1024 MB
  4. Save
- 

## PART B: How to GIVE PERMISSIONS to an Existing Lambda

Lambda permissions come from its **IAM Execution Role**.

---

## STEP 5: Find Lambda's IAM Role

1. Lambda → Configuration → Permissions
  2. Under Execution role
  3. Click the role name  
(e.g. staymate-lambda-role )
- 

## STEP 6: Attach Required Policies

### For S3 Image Upload

Attach:

- AmazonS3FullAccess (*lab only*)  
OR (better):

```
json
```

```
s3:PutObject
```

```
s3:GetObject
```

 Copy code

## For CloudWatch Logs (Mandatory)

Attach:

- `AWSLambdaBasicExecutionRole`

## For RDS (if using IAM auth / Data API)

Attach:

- `AmazonRDSDataFullAccess` (*optional*)

 If you use `username/password`, this policy is **not required**

## STEP 7: Save & Return to Lambda

- IAM changes apply **immediately**
- No redeploy needed

## PART C: Test Permissions

### Test S3 Upload

1. Lambda → Test
2. Trigger image upload
3. Check:
  - S3 bucket → file exists
  - CloudWatch logs → no access denied

### Test RDS Connection

- Invoke API route that queries DB
- If error:
  - Check Security Group

- Check DB credentials
  - Check VPC settings (if any)
- 

## COMMON PERMISSION ERRORS & FIXES

Error	Cause	Fix
AccessDenied: s3	No S3 permission	Attach S3 policy
Timeout	DB unreachable	Public RDS / SG
Cannot find module	ZIP issue	Re-zip correctly
500 error	Missing env vars	Add env vars

---

## BEST PRACTICE (For Viva / Report)

You can confidently say:

*"The Lambda function operates under a dedicated IAM execution role, following the principle of least privilege, granting access only to Amazon S3 and CloudWatch required for image handling and logging."*

---

## ☒ SUMMARY CHECKLIST

- ✓ Code edited & deployed
  - ✓ Environment variables updated
  - ✓ Timeout & memory increased
  - ✓ IAM role permissions attached
  - ✓ CloudWatch logs verified
- 

If you want, next I can:

- Debug AccessDenied errors live
- Show least-privilege IAM policy
- Draw Lambda → S3 → RDS permission diagram
- Help you split roles for prod vs lab

Just tell me 

< 2/2 >      ...

since its a learner lab with limited permissions, what to do as an alternative

Excellent question — this is **exactly where AWS Learner Lab differs from real AWS**, and you're thinking in the right direction 

I'll give you **clear, realistic alternatives** that actually work in Learner Lab, without fighting permissions.



## Reality Check: AWS Learner Lab Limitations

In AWS Learner Lab, you usually face:

-  No IAM policy creation or editing
-  Limited role attachment
-  Restricted services (Cognito, SES, OpenSearch, etc.)
-  Sometimes no S3 public access policies

So instead of forcing "perfect AWS", you use **approved workarounds**.



## BEST PRACTICAL APPROACH FOR LEARNER LAB (RECOMMENDED)

### Use ONE Lambda + ONE S3 Bucket + RDS

and avoid advanced IAM manipulation



### Alternative 1: Use Default Lambda Role (Simplest)

What to do:

- When creating Lambda:
  - Select **Create new role with basic Lambda permissions**

- This role already allows:
  - CloudWatch logs
  - Basic AWS access

Then:

- Use public S3 bucket
- No fine-grained IAM policies needed
- ✓ Works in 90% of learner labs
- ✓ Easy to explain in viva

---

## ● Alternative 2: Skip Direct Upload → Use Pre-Signed URLs (BEST)

👉 This is the CLEANEST workaround

How it works:

1. Lambda generates a pre-signed S3 upload URL
2. React uploads image directly to S3
3. Lambda only stores the image URL in RDS

SCSS

React —PUT—> S3



| (presigned URL)

Lambda

Copy code

Why this works in Learner Lab:

- Lambda only needs `s3:GetObject`
- Upload handled by browser
- Fewer permissions required

Lambda code snippet:

js

```
const s3 = new AWS.S3();

const url = s3.getSignedUrl('putObject', {
 Bucket: 'staymate-images',
 Key: 'image.png',
 ACL: 'public-read',
 Expires: 3600
});
```

```
Key: `hostels/${Date.now()}.jpg`,
Expires: 300,
ContentType: 'image/jpeg'
});

res.json({ uploadUrl: url });
```

- ✓ Minimal permissions
- ✓ Faster uploads
- ✓ Industry best practice

 Copy code

## ● Alternative 3: Store Images in S3 Without DB Update (Temporary)

If RDS permissions are also limited:

- Upload image
- Return image URL to frontend
- Store URL later when DB works

This is acceptable for demos.

## ✗ What NOT to Do in Learner Lab

🚫 Don't:

- Create complex IAM policies
- Create multiple Lambda roles
- Enable Cognito + SES + OpenSearch
- Use private S3 + CloudFront signed URLs

These often fail due to permission limits.

## 🏆 What to SAY in Your Report / Viva

*"Due to AWS Learner Lab IAM restrictions, the project uses pre-signed S3 URLs to securely upload images directly from the frontend. This minimizes required permissions while following AWS serverless best practices."*

This answer actually impresses evaluators.



## FINAL RECOMMENDATION FOR YOU

Component	What to Use
Lambda	Single function
IAM	Default Lab role
Image Upload	Pre-signed URLs
S3	Public bucket
RDS	Username/password
Auth	Basic (no Cognito)

If you want, I can:

- Rewrite your Lambda to **use pre-signed URLs**
- Modify your React upload code
- Give **exact text for your report**
- Create a **learner-lab-safe architecture diagram**

Just tell me what you want next



how to handle two buckets associated with one lambda function

Great question — and yes, this is 100% supported and very common.

A single Lambda function can safely work with multiple S3 buckets, even in AWS Learner Lab, as long as permissions allow it.

I'll explain this clearly, practically, and learner-lab-safe.



## Can One Lambda Use Two S3 Buckets?



One Lambda function can:

- Read from Bucket A
- Write to Bucket B
- Or read/write to both

Nothing special is required in Lambda itself.

---

## Why You'd Use Two Buckets (Good Architecture)

Example for your project:

Bucket	Purpose
staymate-images	Hostel / profile images
staymate-docs	PDFs, ID proofs, documents

This separation:

- Improves organization
  - Makes access control easier
  - Looks professional in evaluation
- 

## STEP-BY-STEP: Handle Two Buckets in One Lambda

### STEP 1: Create Two S3 Buckets

Create both buckets normally:

```
staymate-images
staymate-docs
```

 Copy code

In Learner Lab:

- Disable “Block all public access” (if needed)
  - Same region as Lambda
-

## STEP 2: Store Bucket Names as Environment Variables

Lambda → Configuration → Environment variables

Add:

```
ini

IMAGES_BUCKET = staymate-images
DOCS_BUCKET = staymate-docs
```

 Copy code

This is VERY important (don't hardcode bucket names).

---

## STEP 3: Use Same S3 Client for Both Buckets

Lambda Code (Node.js)

```
js

const AWS = require('aws-sdk');
const s3 = new AWS.S3();

const imagesBucket = process.env.IMAGES_BUCKET;
const docsBucket = process.env.DOCS_BUCKET;
```

 Copy code

## STEP 4: Upload to Different Buckets Based on Logic

Example: Image Upload

```
js

await s3.putObject({
 Bucket: imagesBucket,
 Key: `hostels/${Date.now()}-${fileName}`,
 Body: fileBuffer,
 ContentType: 'image/jpeg'
}).promise();
```

Example: Document Upload

 Copy code

```
js
```

```
await s3.putObject({
 Bucket: docsBucket,
 Key: `documents/${Date.now()}-${fileName}`,
 Body: fileBuffer,
 ContentType: 'application/pdf'
}).promise();
```

 Copy code

## STEP 5: (BEST for Learner Lab) Use Pre-Signed URLs

This avoids permission pain.

### Generate Pre-Signed URL for Image Bucket

js

```
const imageUploadUrl = s3.getSignedUrl('putObject', {
 Bucket: imagesBucket,
 Key: `hostels/${Date.now()}.jpg`,
 Expires: 300,
 ContentType: 'image/jpeg'
});
```

 Copy code

### Generate Pre-Signed URL for Docs Bucket

js

```
const docUploadUrl = s3.getSignedUrl('putObject', {
 Bucket: docsBucket,
 Key: `docs/${Date.now()}.pdf`,
 Expires: 300,
 ContentType: 'application/pdf'
});
```

 Copy code

Frontend uploads directly to the correct bucket.



## IAM PERMISSIONS (Learner Lab Reality)

Best-case (available):

- `AmazonS3FullAccess`

## If restricted:

- Use public buckets
- Use pre-signed URLs
- Avoid bucket policies

Lambda does **not** need separate roles per bucket.

---

## Folder Structure in Buckets (Recommended)

```
staymate-images/
```

```
 └── hostels/
```

```
 └── profiles/
```

```
staymate-docs/
```

```
 └── verification/
```

```
 └── agreements/
```

Looks clean and professional.

 Copy code

---

## Testing Checklist

- ✓ Upload image → goes to image bucket
  - ✓ Upload document → goes to docs bucket
  - ✓ URLs returned correctly
  - ✓ URLs saved in RDS
- 

## What to Say in Report / Viva

*"A single AWS Lambda function interacts with multiple Amazon S3 buckets, separating media and document storage. Bucket names are configured via environment variables, enabling flexible and scalable storage management."*

This is architecture gold.

---

## Final Recommendation

- ✓ One Lambda
  - ✓ Two S3 buckets
  - ✓ Environment variables
  - ✓ Pre-signed URLs
  - ✓ Simple, learner-lab-safe
- 

If you want, I can:

- Refactor your Lambda code cleanly
- Show React upload logic for **both buckets**
- Help write **LaTeX explanation**
- Draw **architecture diagram**

Just tell me 



if we upload the images in the same bucket where our frontend is but in a folder, how would it effect

This is a **very smart question**, because what you're asking is a **real architectural trade-off** — and many student projects *do* exactly this.

Let's answer it **clearly, honestly, and from an evaluation + production perspective**.

---

## Scenario

You have **ONE S3 bucket** like:

cpp

```
staymate-frontend-bucket
├── index.html
├── assets/
└── static/
```

```
└─ uploads/
 └─ images/
```

 Copy code

Frontend and uploaded images live in the same bucket.

---

## Is This ALLOWED?

👉 YES. 100% works.

AWS does not restrict this at all.

---

## Positive Effects (Why This Works Well in Learner Lab)

### 1 Much Simpler Setup

- One bucket
- No extra IAM or bucket policies
- Fewer permissions headaches

✓ Perfect for AWS Learner Lab

---

### 2 Easy Public Access

- Frontend already public
- Images automatically public
- No CORS headaches

jsx

```

```

 Copy code

### 3 Faster Development

- No need for pre-signed URLs
  - No need for multiple buckets
  - Less backend logic
-

## 4 Acceptable for Academic Projects

For semester projects, evaluators will not penalize this if you explain it properly.

---

## ● Downsides (Important to Understand)

### ✗ 1 Security Risk

- Frontend bucket is public
- Uploaded images are public by default
- No access control

⚠ OK for public hostel images

⚠ NOT OK for private documents

---

### ✗ 2 Deployment Risk

If you redeploy frontend:

```
bash
```

```
aws s3 sync dist/ s3://staymate-frontend-bucket --delete
```

 Copy code

⚠ This may **DELETE** uploaded images

---

### ✗ 3 No Clear Separation of Concerns

- Frontend assets ≠ User uploads
  - Mixing them is bad practice in production
- 

### ✗ 4 Scaling & Caching Issues

- Frontend assets benefit from long cache TTL
- Images may need different caching rules

Hard to manage in one bucket.

---

# 🟡 BEST WAY IF YOU STILL USE ONE BUCKET (Recommended Fix)

- ✓ Create a Dedicated Upload Folder

```
pgsql
```

```
staymate-frontend-bucket
├── index.html
├── assets/
└── user-uploads/
 └── images/
```

 Copy code

- ✓ Never delete uploads on frontend deploy

Use:

```
bash
```

```
aws s3 sync dist/ s3://staymate-frontend-bucket
```

 Copy code

- ✗ DO NOT use `--delete`

- 
- ✓ Store Image URLs in RDS

```
sql
```

```
image_url = 'https://bucket.s3.amazonaws.com/user-uploads/images/abc.jpg'
```

 Copy code

- 
- ✓ Explain in Report

*"For simplicity in the AWS Learner Lab environment, frontend assets and public image uploads are stored in separate folders within the same S3 bucket. The architecture allows easy migration to dedicated media buckets in production."*

This explanation protects your marks.



## Final Verdict (Honest Recommendation)

Situation	Recommendation
AWS Learner Lab	✓ Same bucket, different folder
University Project	✓ Acceptable
Production System	✗ Separate buckets
Private images	✗ Separate bucket

## 🚀 What I Recommend YOU Do

### 👉 Use ONE S3 bucket

- / → frontend
- /user-uploads/images/ → hostel images

### 👉 Mention future scalability in report

### 👉 Focus on working demo, not IAM pain

If you want, next I can:

- Show safe S3 sync commands
- Write exact report paragraph
- Modify Lambda logic for same-bucket uploads
- Help add CloudFront later

Just tell me 👍



can u give the basic nodejs code for connecting our backend with rds, and afterwards with frontend

Absolutely 👍

I'll give you clean, minimal, learner-lab-safe code that:

1. Connects Node.js (Express) backend to RDS PostgreSQL

2. Exposes a simple API
3. Consumes that API from React frontend

This is the **exact foundation** you need before adding Lambda / API Gateway.

---

## PART 1: Node.js Backend → RDS (PostgreSQL)

### 1 Install Required Packages

```
bash

npm init -y
npm install express pg cors dotenv
```

 Copy code

### 2 Backend Folder Structure

```
bash

backend/
 └── server.js
 └── db.js
 └── .env
```

 Copy code

### 3 Database Connection (db.js)

```
js

const { Pool } = require('pg');

const pool = new Pool({
 host: process.env.DB_HOST, // RDS endpoint
 user: process.env.DB_USER,
 password: process.env.DB_PASS,
 database: process.env.DB_NAME,
 port: 5432,
 ssl: { rejectUnauthorized: false } // required for RDS
});
```

```
module.exports = pool;
```

 Copy code

## 4 Express Server (server.js)

js

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');
const pool = require('./db');

const app = express();
app.use(cors());
app.use(express.json());

/* Test route */
app.get('/health', (req, res) => {
 res.json({ status: 'Backend is running' });
});

/* Get all hostels */
app.get('/hostels', async (req, res) => {
 try {
 const result = await pool.query('SELECT * FROM hostels');
 res.json(result.rows);
 } catch (err) {
 console.error(err);
 res.status(500).json({ error: 'Database error' });
 }
});

/* Add hostel */
app.post('/hostels', async (req, res) => {
 const { name, location, price_per_month, image_url } = req.body;
 try {
 await pool.query(
 `INSERT INTO hostels (name, location, price_per_month, image_url)
 VALUES ($1, $2, $3, $4)`,
 [name, location, price_per_month, image_url]
);

```

```
res.status(201).json({ message: 'Hostel added' });
} catch (err) {
 console.error(err);
 res.status(500).json({ error: 'Insert failed' });
}
});

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
 console.log(`Server running on port ${PORT}`);
});
```

## 5 Environment Variables (.env)

env

```
DB_HOST=staymate-db.firebaseio.us-east-1.rds.amazonaws.com
DB_USER=postgres
DB_PASS=yourpassword
DB_NAME=postgres
PORT=5000
```

 Copy code

## 6 Test Backend Locally

bash

```
node server.js
```

 Copy code

Test in browser:

bash

```
http://localhost:5000/health
http://localhost:5000/hostels
```

 Copy code



# PART 2: React Frontend → Backend

## 1 React API Call (Basic Example)

js

```
const API_URL = "http://localhost:5000";

export const getHostels = async () => {
 const res = await fetch(` ${API_URL}/hostels`);
 return res.json();
};
```

 Copy code

## 2 React Component Example

jsx

```
import { useEffect, useState } from 'react';
import { getHostels } from './api';

function Hostels() {
 const [hostels, setHostels] = useState([]);

 useEffect(() => {
 getHostels().then(setHostels);
 }, []);

 return (
 <div className="container">
 <h2>Available Hostels</h2>
 {hostels.map(h => (
 <div key={h.hostel_id} className="card mb-3">

 <div className="card-body">
 <h5>{h.name}</h5>
 <p>{h.location}</p>
 <p>PKR {h.price_per_month}</p>
 </div>
 </div>
)));
 </div>
);
}
```

```
export default Hostels;
```

## ● PART 3: After Deploying to AWS (IMPORTANT)

When backend is on Lambda + API Gateway

 Copy code

Change:

js

```
const API_URL = "https://abc123.execute-api.us-east-1.amazonaws.com";
```

 Copy code

Frontend stays the same

No code change needed beyond URL.

## ● ARCHITECTURE FLOW (Simple)

SCSS

React (Browser)

↓ `fetch()`

API Gateway

↓

Lambda (Express)

↓

RDS (PostgreSQL)

 Copy code

## 🧠 KEY POINTS FOR LEARNER LAB

- ✓ Use username/password for RDS
- ✓ Make RDS publicly accessible
- ✓ Allow inbound 5432 in security group
- ✓ Use CORS in backend
- ✓ Test locally first

## WHAT TO SAY IN VIVA

*"The backend is built using Node.js and Express, connected to Amazon RDS PostgreSQL via the pg client. RESTful APIs are exposed through API Gateway and consumed by a React frontend hosted on Amazon S3."*

If you want, next I can:

- Convert this backend into Lambda-ready code
- Add image upload endpoint
- Add authentication
- Write report-ready explanation

Just tell me 



how to connect our aws rds withdbeaver to use

Below is a clean, beginner-safe, step-by-step guide to connect AWS RDS (PostgreSQL) with DBeaver, written specifically for AWS Learner Lab users.

## Connect AWS RDS to DBeaver (PostgreSQL)

### STEP 1: Verify RDS Configuration (VERY IMPORTANT)

Go to AWS Console → RDS → Databases → Your DB instance

 Check these settings:

#### 1 Public Accessibility

- Must be YES

#### 2 VPC Security Group

Click the Security Group attached to RDS → Inbound Rules

Add rule:

Type	Protocol	Port	Source
PostgreSQL	TCP	5432	Your IP / 0.0.0.0/0 (lab only)

 If you skip this, DBeaver will not connect

### 3 Database Status

- Must be Available

## STEP 2: Collect RDS Connection Details

From the same RDS page:

Field	Value
Endpoint	xxxxx.us-east-1.rds.amazonaws.com
Port	5432
DB Name	postgres (default)
Username	Master username
Password	Master password

## STEP 3: Open DBeaver & Create New Connection

1. Open DBeaver
2. Click New Database Connection
3. Select PostgreSQL
4. Click Next

## STEP 4: Enter Connection Details

Fill exactly like this:

DBeaver Field	Value
Host	RDS Endpoint
Port	5432
Database	postgres
Username	RDS username
Password	RDS password

## STEP 5: SSL Configuration (IMPORTANT)

Click SSL tab:

Setting	Value
Use SSL	<input checked="" type="checkbox"/> Enabled
SSL Mode	require

 This is required for RDS PostgreSQL

## STEP 6: Test Connection

Click Test Connection

 If successful:

- Click Finish
- You're connected 

## ✗ COMMON ERRORS & FIXES

✗ *Connection timed out*

 Security Group inbound rule missing

✗ *No pg\_hba.conf entry*

- ✓ Enable SSL + `require`

## ✗ **Authentication failed**

- ✓ Wrong username/password

## ✗ **Connection refused**

- ✓ RDS not public
- 

## STEP 7: Use DBeaver (Basic Operations)

### ▶ Create Tables

- Right-click database → SQL Editor
- Paste SQL
- Click Run

### ▶ View Tables

- Schemas → public → Tables

### ▶ Insert Data

```
sql
```

```
INSERT INTO users (full_name, email, role)
VALUES ('Ali', 'ali@test.com', 'customer');
```

Copy code

---

## 🧠 Best Practices (For Learner Lab)

- ✓ Keep DB public
  - ✓ Use password auth (not IAM)
  - ✓ Restrict SG later (optional)
  - ✓ Use DBeaver instead of Query Editor
- 

## 🏆 What You Can Say in Viva

*"We used DBeaver as a PostgreSQL client to manage and query our Amazon RDS database. Connectivity was secured via security groups and SSL-enabled connections."*

## Final Checklist

- RDS public
- Port 5432 open
- Correct endpoint
- SSL enabled
- DBeaver test successful

If you want, next I can:

- Give StayMate SQL schema
- Help debug connection errors
- Show RDS + Lambda + DBeaver architecture diagram
- Write report-ready explanation

Just tell me 



generate a schema for this

Perfect! Since your project is **StayMate – a hostel & mess booking platform**, I'll generate a **learner-lab-safe PostgreSQL schema** that works well with your **React + Node + Express + RDS backend**.

The schema will include **users, hostels, bookings, images, and mess options**, and will use **RDBMS principles**.

## StayMate PostgreSQL Schema

sql

```
-- =====
-- USERS TABLE
-- Stores all students/customers and hosts
-- =====
CREATE TABLE users (
```

```

 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 role VARCHAR(20) NOT NULL CHECK (role IN ('customer', 'host')),
 phone_number VARCHAR(20),
 created_at TIMESTAMP DEFAULT NOW()
);

-- =====
-- HOSTELS TABLE
-- Stores hostel details
-- =====

CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 host_id INT NOT NULL REFERENCES users(user_id),
 name VARCHAR(100) NOT NULL,
 location VARCHAR(200) NOT NULL,
 price_per_month NUMERIC(10,2) NOT NULL,
 description TEXT,
 rating NUMERIC(2,1) DEFAULT 0,
 created_at TIMESTAMP DEFAULT NOW()
);

-- =====
-- HOSTEL IMAGES TABLE
-- Stores image URLs for hostels
-- =====

CREATE TABLE hostel_images (
 image_id SERIAL PRIMARY KEY,
 hostel_id INT NOT NULL REFERENCES hostels(hostel_id) ON DELETE CASCADE,
 image_url TEXT NOT NULL,
 uploaded_at TIMESTAMP DEFAULT NOW()
);

-- =====
-- MESS OPTIONS TABLE
-- Stores mess/meal options for hostels
-- =====

CREATE TABLE mess_options (
 mess_id SERIAL PRIMARY KEY,
 hostel_id INT NOT NULL REFERENCES hostels(hostel_id) ON DELETE CASCADE,
 meal_type VARCHAR(50) NOT NULL,

```

```

 price_per_month NUMERIC(10,2) NOT NULL,
 created_at TIMESTAMP DEFAULT NOW()
);

-- =====
-- BOOKINGS TABLE
-- Stores bookings made by customers
-- =====

CREATE TABLE bookings (
 booking_id SERIAL PRIMARY KEY,
 user_id INT NOT NULL REFERENCES users(user_id),
 hostel_id INT NOT NULL REFERENCES hostels(hostel_id),
 mess_id INT REFERENCES mess_options(mess_id),
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 status VARCHAR(20) NOT NULL DEFAULT 'pending' CHECK (status IN ('pending',
 created_at TIMESTAMP DEFAULT NOW()
);

-- =====
-- Optional: Reviews Table
-- Stores reviews for hostels
-- =====

CREATE TABLE reviews (
 review_id SERIAL PRIMARY KEY,
 hostel_id INT NOT NULL REFERENCES hostels(hostel_id),
 user_id INT NOT NULL REFERENCES users(user_id),
 rating NUMERIC(2,1) NOT NULL CHECK (rating >= 0 AND rating <= 5),
 comment TEXT,
 created_at TIMESTAMP DEFAULT NOW()
);

```

## Notes / Best Practices

### 1. Image Handling

- `hostel_images.image_url` stores S3 URL
- One hostel can have multiple images

### 2. Bookings

- `status` allows tracking workflow: pending → approved/rejected → ended

- Supports optional mess selection ( `mess_id` )

### 3. Users

- `role` defines **customer or host**
- `password_hash` stores hashed passwords only

### 4. Cascading Deletes

- When hostel is deleted, images and mess options are removed automatically

---

 Copy code

## How This Works With Your Stack

- Backend (Node.js + Express) → CRUD operations on these tables
- Frontend (React + Bootstrap) → Fetch `/hostels` , `/bookings` , `/mess_options`
- S3 → Stores hostel images; URL stored in `hostel_images`
- RDS → Manages relational data

---

If you want, I can next generate a **diagram showing all table relationships (ERD)** and a **sample dataset** for testing your frontend.

Do you want me to do that next?



we have three end user: students, mess owner, hostel owner so we have to optimize the schema accordingly

Ah, perfect — now we're moving from a **basic schema** to a **role-optimized schema** for **three types of users**: students, hostel owners, and mess owners. This will make your database cleaner, easier to query, and more scalable.

I'll generate a **full RDBMS schema** optimized for your StayMate project.



## Optimized StayMate Schema (Three End Users)

## 1 USERS TABLE (All Users)

sql

```
CREATE TABLE users (
 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 role VARCHAR(20) NOT NULL CHECK (role IN ('student', 'hostel_owner', 'mess_owner')),
 phone_number VARCHAR(20),
 created_at TIMESTAMP DEFAULT NOW()
);
```

Notes:

 Copy code

- “`role` determines if the user is a student, hostel owner, or mess owner”
- “All users stored in one table for simplicity”

## 2 HOSTELS TABLE (Only Hostel Owners)

sql

```
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 owner_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
 name VARCHAR(100) NOT NULL,
 location VARCHAR(200) NOT NULL,
 price_per_month NUMERIC(10,2) NOT NULL,
 description TEXT,
 rating NUMERIC(2,1) DEFAULT 0,
 created_at TIMESTAMP DEFAULT NOW()
);
```

Notes:

- “`owner_id` must point to a user with `role = 'hostel_owner'`”
- “Could enforce via backend validation”

 Copy code

## 3 MESSES TABLE (Only Mess Owners)

sql

```
CREATE TABLE messes (
 mess_id SERIAL PRIMARY KEY,
 owner_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
 name VARCHAR(100) NOT NULL,
 location VARCHAR(200) NOT NULL,
 price_per_month NUMERIC(10,2) NOT NULL,
 description TEXT,
 created_at TIMESTAMP DEFAULT NOW()
);
```

Notes:

- “Each mess belongs to a mess\_owner”
- “Could have multiple messes per owner”

 Copy code

## 4 HOSTEL IMAGES TABLE

sql

```
CREATE TABLE hostel_images (
 image_id SERIAL PRIMARY KEY,
 hostel_id INT NOT NULL REFERENCES hostels(hostel_id) ON DELETE CASCADE,
 image_url TEXT NOT NULL,
 uploaded_at TIMESTAMP DEFAULT NOW()
);
```

 Copy code

Notes:

- “One hostel can have multiple images”
- “Store S3 URLs only”

## 5 MESS IMAGES TABLE

sql

```
CREATE TABLE mess_images (
 image_id SERIAL PRIMARY KEY,
 mess_id INT NOT NULL REFERENCES messes(mess_id) ON DELETE CASCADE,
 image_url TEXT NOT NULL,
```

```
uploaded_at TIMESTAMP DEFAULT NOW()
);
```

 Copy code

Notes:

- “Same as hostels, separate table for clarity”

## 6 BOOKINGS TABLE (Students Book Hostels or Messes)

sql

```
CREATE TABLE bookings (
 booking_id SERIAL PRIMARY KEY,
 student_id INT NOT NULL REFERENCES users(user_id),
 hostel_id INT REFERENCES hostels(hostel_id),
 mess_id INT REFERENCES messes(mess_id),
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 status VARCHAR(20) NOT NULL DEFAULT 'pending'
 CHECK (status IN ('pending', 'approved', 'rejected', 'ended')),
 created_at TIMESTAMP DEFAULT NOW()
);
```

 Copy code

Notes:

- “hostel\_id or mess\_id is required for each booking”
- “Students can book both a hostel and a mess separately”
- “Status helps track workflow”

## 7 REVIEWS TABLE

sql

```
CREATE TABLE reviews (
 review_id SERIAL PRIMARY KEY,
 user_id INT NOT NULL REFERENCES users(user_id),
 hostel_id INT REFERENCES hostels(hostel_id),
 mess_id INT REFERENCES messes(mess_id),
 rating NUMERIC(2,1) NOT NULL CHECK (rating >= 0 AND rating <= 5),
 comment TEXT,
```

```
 created_at TIMESTAMP DEFAULT NOW()
);
```

 Copy code

Notes:

- “Either `hostel_id` or `mess_id` must be NOT NULL”
- “Students leave feedback”

## 8 Optional: Availability / Slots Table (Advanced)

sql

```
CREATE TABLE availability (
 id SERIAL PRIMARY KEY,
 hostel_id INT REFERENCES hostels(hostel_id),
 mess_id INT REFERENCES messes(mess_id),
 total_slots INT NOT NULL,
 booked_slots INT DEFAULT 0,
 date DATE NOT NULL
);
```

Helps track availability of rooms/meals per day

 Copy code

## WHY THIS IS OPTIMIZED

Feature	Benefit
Single users table	Easy auth / role management
Separate hostels & messes	Each owner manages their assets
Separate image tables	Avoid mixing images, easier S3 management
Bookings table	Supports students booking hostels or messes
Reviews table	Students provide ratings for both services
Status fields	Tracks workflow: pending → approved → ended

## CONNECTIONS / RELATIONSHIPS

- **users (hostel\_owner)** → hostels (1:N)
  - **users (mess\_owner)** → messes (1:N)
  - **users (student)** → bookings (1:N)
  - **hostels** → hostel\_images (1:N)
  - **messes** → mess\_images (1:N)
  - **users (student)** → reviews (1:N)
- 

## Notes for Your Stack

- **Backend (Node.js/Express):**
    - CRUD for hostels/messes for owners
    - Bookings for students
    - Images uploaded to S3, URLs saved in respective tables
  - **Frontend (React):**
    - Separate dashboard for students, hostel owners, mess owners
    - Fetch `/hostels`, `/messes`, `/bookings` based on role
  - **RDS:**
    - Public endpoint for Lambda (or local dev via DBeaver)
    - PostgreSQL supports constraints, cascading deletes
- 

If you want, I can draw a clean ERD diagram for this optimized schema, showing all relationships and role connections — perfect for report + viva.

Do you want me to do that next?



how about we define a data type as user roles with only three values, and anyone who tries to make an account will only be able to choose his role out of the defined ones

Exactly! That's a **perfect use** of PostgreSQL's **ENUM** type. This allows you to **strictly enforce** that `role` can only be one of `'student'`, `'hostel_owner'`, or `'mess_owner'`. Anyone trying to insert anything else will get an **error automatically** — no need for extra backend validation.

Here's how you can do it in PostgreSQL:

## Step 1: Create ENUM Type for User Roles

sql

```
CREATE TYPE user_role AS ENUM ('student', 'hostel_owner', 'mess_owner');
```

 Copy code

 `user_role` is now a data type that can **only have one of these three values**.

## Step 2: Use ENUM in Users Table

sql

```
CREATE TABLE users (
 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 role user_role NOT NULL, -- ENUM type
 phone_number VARCHAR(20),
 created_at TIMESTAMP DEFAULT NOW()
);
```

Now, `role` cannot have any value outside the three defined ones.

 Copy code

## Step 3: Test ENUM Constraint

sql

```
-- Valid insert
INSERT INTO users (full_name, email, password_hash, role)
```

```
VALUES ('Ali', 'ali@test.com', 'hashedpassword', 'student');

-- Invalid insert (will fail)
INSERT INTO users (full_name, email, password_hash, role)
VALUES ('Sara', 'sara@test.com', 'hashedpassword', 'admin');
-- ERROR: invalid input value for enum user_role: "admin"
```

 Copy code

## Advantages of ENUM for Roles

1.  Strict validation at the database level
2.  Cleaner queries ( WHERE role = 'student' )
3.  Easier to maintain and extend (you can add a role later if needed)
4.  Makes your schema more professional for reports and viva

## Integrate With Backend

In Node.js / Express, you can use it normally:

```
js

const { role } = req.body; // 'student', 'hostel_owner', 'mess_owner'
await pool.query(
 `INSERT INTO users (full_name, email, password_hash, role)
 VALUES ($1, $2, $3, $4)`,
 [fullName, email, hashedPassword, role]
);
```

 Copy code

- If `role` is invalid, PostgreSQL will throw an error automatically
- Backend can catch this and return a 400 Bad Request

## Optional: Extendable ENUM

- Later, if you want to add a new role (like `'admin'`):

```
sql
```

```
ALTER TYPE user_role ADD VALUE 'admin';
```

 Copy code

## Pro tip for report / viva:

*"We use PostgreSQL ENUM type for user roles to ensure data integrity. Only the predefined roles — student, hostel\_owner, mess\_owner — can be inserted, preventing invalid entries."*

---

If you want, I can **rewrite the entire optimized StayMate schema using ENUM for roles** and make it **ready for deployment** — including hostels, messes, bookings, and images — so it's **report-ready**.

Do you want me to do that?



why do we need a separate table for images link, we should add those link in the hostel and mess tables

Ah, now you're asking a **classic database design question**: "Why use a separate table for images instead of putting image URLs directly in the hostel/mess tables?" Let me explain **clearly, with pros and cons**, so you can decide.

---

## Option 1: Store Image URLs Directly in Hostel/Mess Tables

Example:

sql

```
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 owner_id INT REFERENCES users(user_id),
 name VARCHAR(100),
 location VARCHAR(200),
 image_url TEXT, -- only one image
```

```
 created_at TIMESTAMP DEFAULT NOW()
);
```

 Copy code

## Advantages

1. Simple schema — fewer tables
2. Easy to query ( `SELECT * FROM hostels` )
3. Works if each hostel/mess has **only one image**

## Disadvantages

1. Limited to 1 image per hostel/mess
  - What if hostel wants 5 images?
  - You'd need multiple columns ( `image1, image2, image3...` ) → messy
2. Not scalable — adding more images requires **altering table**
3. Harder to manage metadata for images (like `uploaded_at, captions`)

## Option 2: Use a Separate Images Table

sql

```
CREATE TABLE hostel_images (
 image_id SERIAL PRIMARY KEY,
 hostel_id INT REFERENCES hostels(hostel_id) ON DELETE CASCADE,
 image_url TEXT NOT NULL,
 uploaded_at TIMESTAMP DEFAULT NOW()
);
```

## Advantages

1. Multiple images per hostel/mess — scalable
2. Can store extra info:
  - `uploaded_at`
  - `caption` or `alt_text`
  - `image_type` (room, exterior, menu, etc.)
3. Keeps hostel/mess tables clean
4. Supports many-to-many if needed in future

 Copy code

## Disadvantages

1. Requires **JOINS** to fetch images

## 2. Slightly more complex queries

---

### Practical Recommendation

For StayMate, where hostels and messes will have multiple images:

- Use separate `hostel_images` and `mess_images` tables
- Store only S3 URLs there
- Keep `hostels` and `messes` tables clean
- If needed, fetch first image for a preview using a query like:

sql

```
SELECT h.*, hi.image_url
FROM hostels h
LEFT JOIN hostel_images hi ON h.hostel_id = hi.hostel_id
GROUP BY h.hostel_id, hi.image_url
HAVING MIN(hi.image_id);
```

This is professional, scalable, and clean, exactly what evaluators expect.  [Copy code](#)

### 💡 Rule of Thumb (DB Design)

- 1 image → embed in table
- Multiple images → separate table
- Always normalize your database for scalability

If you want, I can rewrite your entire StayMate schema with this approach, optimized for multiple images, ENUM roles, and bookings/mess support, ready for RDS + Lambda integration.

Do you want me to do that?



since we have less time, and rookie lvl exp, we should just stick with one img right now and in future iteration

make that possible

Absolutely — that's a **pragmatic and smart decision** .

Since you're on a **tight timeline** and this is your first AWS/React/Node project, you can:

- Stick with **one image URL per hostel/mess**
- Store it **directly in the `hostels` and `messes` tables**
- Keep the schema **simple and beginner-friendly**
- Document in your report that **future iterations will support multiple images** using a separate table

This way, you can **deliver a working MVP in 3–4 days**, which is exactly what your evaluators care about.

---

Here's the **simplified schema with one image per hostel/mess**:

sql

```
-- USERS TABLE
CREATE TYPE user_role AS ENUM ('student', 'hostel_owner', 'mess_owner');

CREATE TABLE users (
 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 role user_role NOT NULL,
 phone_number VARCHAR(20),
 created_at TIMESTAMP DEFAULT NOW()
);

-- HOSTELS TABLE
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 owner_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
 name VARCHAR(100) NOT NULL,
 location VARCHAR(200) NOT NULL,
 price_per_month NUMERIC(10,2) NOT NULL,
```

```

description TEXT,
image_url TEXT, -- Single image for now
rating NUMERIC(2,1) DEFAULT 0,
created_at TIMESTAMP DEFAULT NOW()
);

-- MESSES TABLE
CREATE TABLE messes (
mess_id SERIAL PRIMARY KEY,
owner_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
name VARCHAR(100) NOT NULL,
location VARCHAR(200) NOT NULL,
price_per_month NUMERIC(10,2) NOT NULL,
description TEXT,
image_url TEXT, -- Single image for now
created_at TIMESTAMP DEFAULT NOW()
);

-- BOOKINGS TABLE
CREATE TABLE bookings (
booking_id SERIAL PRIMARY KEY,
student_id INT NOT NULL REFERENCES users(user_id),
hostel_id INT REFERENCES hostels(hostel_id),
mess_id INT REFERENCES messes(mess_id),
start_date DATE NOT NULL,
end_date DATE NOT NULL,
status VARCHAR(20) NOT NULL DEFAULT 'pending'
 CHECK (status IN ('pending','approved','rejected','ended')),
created_at TIMESTAMP DEFAULT NOW()
);

-- REVIEWS TABLE
CREATE TABLE reviews (
review_id SERIAL PRIMARY KEY,
user_id INT NOT NULL REFERENCES users(user_id),
hostel_id INT REFERENCES hostels(hostel_id),
mess_id INT REFERENCES messes(mess_id),
rating NUMERIC(2,1) NOT NULL CHECK (rating >= 0 AND rating <= 5),
comment TEXT,
created_at TIMESTAMP DEFAULT NOW()
);

```

## Advantages of This Approach

1. Simpler backend logic — no need to handle multiple images or joins
  2. Easier for frontend — just fetch `image_url` from hostel/mess
  3. Fast MVP delivery — perfect for 3–4 day timeline
  4. Future-proof — can switch to a separate images table later
- 

### Tip for your report/viva:

*"For this MVP, each hostel and mess currently supports only one image. In future iterations, we plan to move images into a separate table to allow multiple images per entity, which will enhance scalability and flexibility."*

 Copy code

If you want, I can now make a ready-to-use ERD diagram for this simplified schema, showing students, hostel owners, mess owners, bookings, and images — perfect for your report.

Do you want me to do that next?



we should make a separate table for rooms in a hostel for detail and better functioning, this can be used for further monitoring booking too

Absolutely — that's a very smart design improvement .

Adding a `rooms` table allows you to:

- Track individual rooms in each hostel
- Assign bookings per room (instead of just per hostel)
- Add room-level details like capacity, type, price
- Extend functionality later (availability, maintenance, occupancy monitoring)

This keeps your project **scalable** while still simple enough for your MVP.

# Updated Hostel-Related Schema with Rooms

sql

```
-- HOSTELS TABLE (Parent)
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 owner_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
 name VARCHAR(100) NOT NULL,
 location VARCHAR(200) NOT NULL,
 description TEXT,
 image_url TEXT, -- single image for now
 rating NUMERIC(2,1) DEFAULT 0,
 created_at TIMESTAMP DEFAULT NOW()
);

-- ROOMS TABLE (Child of Hostel)
CREATE TABLE rooms (
 room_id SERIAL PRIMARY KEY,
 hostel_id INT NOT NULL REFERENCES hostels(hostel_id) ON DELETE CASCADE,
 room_number VARCHAR(20) NOT NULL,
 room_type VARCHAR(50), -- e.g., single, double, suite
 capacity INT DEFAULT 1,
 price_per_month NUMERIC(10,2) NOT NULL,
 is_available BOOLEAN DEFAULT TRUE,
 created_at TIMESTAMP DEFAULT NOW()
);

-- BOOKINGS TABLE (Linked to room)
CREATE TABLE bookings (
 booking_id SERIAL PRIMARY KEY,
 student_id INT NOT NULL REFERENCES users(user_id),
 room_id INT REFERENCES rooms(room_id),
 mess_id INT REFERENCES messes(mess_id), -- optional mess booking
 start_date DATE NOT NULL,
 end_date DATE NOT NULL,
 status VARCHAR(20) NOT NULL DEFAULT 'pending'
 CHECK (status IN ('pending', 'approved', 'rejected', 'ended')),
 created_at TIMESTAMP DEFAULT NOW()
);
```

## Why This is Better

### 1. Granular bookings

- Instead of “any hostel,” you book a **specific room**
- Easier to track availability per room

### 2. Room-level pricing

- Single/double rooms can have different prices

 Copy code

### 3. Future enhancements

- Monitor occupancy, maintenance, ratings per room
- Dynamic pricing based on demand

### 4. Cleaner logic

- Hostels table remains general info
- Rooms table handles all per-room details

---

## Example Data Flow

- Hostel owner creates hostel → multiple rooms
- Student books a **specific room** → booking record stores `room_id`
- Mess owner adds mess → student can also book mess

---

### ◆ ERD Relationships for This Part

ruby

```
users (hostel_owner) 1:N hostels
hostels 1:N rooms
users (student) 1:N bookings
rooms 1:N bookings
```

 Copy code

## Tip for MVP

- For now, each hostel can have **just 2–3 rooms** for testing
- Keep `room_number` simple ( 101, 102, 103 )
- Later, can add **room images or amenities** in another table

If you want, I can rewrite your full StayMate schema including rooms, ENUM roles, hostels, messes, and single-image MVP, ready for RDS + Lambda deployment.

Do you want me to do that next?



generate a prompt which will help me understand how to configure those AWS services

Absolutely! Here's a **detailed learning-oriented prompt** you can use (for ChatGPT or any AI tutor) to understand how to configure your AWS services for **StayMate** — including S3, Lambda, RDS, API Gateway, Cognito, and CloudFront.



## AWS Configuration Learning Prompt

markdown

I am building a serverless web application called StayMate using React for fr

1. **Amazon S3** – For hosting the frontend and storing uploaded images.

- How to create a bucket for the frontend
- How to configure static website hosting
- Folder structure for frontend assets and images
- Bucket permissions for public access in learner lab
- Best practices to avoid deleting uploaded images during deployments

2. **AWS Lambda** – For hosting the backend serverless functions.

- How to deploy Node.js Express backend as Lambda
- How to manage multiple endpoints (GET, POST, PUT, DELETE)
- How to configure environment variables for RDS connection
- Permissions needed for Lambda to access S3 and RDS
- How to update an existing Lambda function

3. **Amazon API Gateway** – To expose Lambda functions as REST APIs.

- How to create a new API Gateway for the backend
- How to integrate Lambda functions

- How to configure CORS for React frontend
  - How to deploy API and get the endpoint URL
4. **Amazon RDS (PostgreSQL)** - For relational database.
- How to create an RDS instance in learner lab
  - How to configure security groups to allow Lambda or local access
  - How to connect using DBeaver and Node.js
  - Best practices for public access in learner lab environment
5. **Amazon Cognito** - For authentication.
- How to create a user pool
  - How to integrate with React frontend and Node.js backend
  - How to restrict roles (student, hostel\_owner, mess\_owner)
  - How to handle JWTs in backend API calls
6. **Amazon CloudFront** - For CDN and frontend distribution.
- How to create a distribution for the S3 frontend
  - How to configure origin and caching policies
  - How to update distribution when frontend changes
7. **AWS IAM** - To configure permissions and roles.
- What roles/policies are required for Lambda, S3, API Gateway, and RDS
  - How to assign least privilege in learner lab environment
  - How to handle permissions for image upload and API access
8. **Amazon CloudWatch** - For logging and monitoring.
- How to view Lambda logs
  - How to create alerts for errors

Please give a **step-by-step practical guide** for each service in the order

## How to Use This Prompt

1. Copy the prompt and paste it into ChatGPT or any AI assistant.
2. Ask it to **explain service by service or all together**.
3. Use it to **build your AWS environment step-by-step**.
4. Follow the instructions **while testing** in learner lab.

 Copy code

If you want, I can also generate a second, more compact “cheat sheet version” that lists all the exact steps, commands, and GUI clicks for deploying StayMate’s full stack on AWS — ready for MVP deployment in 3–4 days.

Do you want me to do that?



# StayMate - Complete Setup & Deployment Guide

## Overview

This guide will help you set up and deploy the complete StayMate application with all new features including:

- Room management for hostel owners
- Image upload to AWS S3
- Professional UI redesign
- Landing page
- Fixed authentication and CRUD operations
- Proper error handling

## Prerequisites

- Node.js (v14 or higher)
- PostgreSQL database on AWS RDS
- AWS Account with:
  - S3 bucket for image storage
  - Lambda function
  - API Gateway
- Git

## Project Structure

```
staymate/
├── client/ # React frontend
│ ├── public/
│ │ └── logo.png # Your logo image
│ └── src/
│ ├── components/
│ │ ├── LandingPage.jsx
│ │ ├── Navbar.jsx
│ │ ├── Auth.jsx
│ │ ├── Home.jsx
│ │ ├── HostelOwnerDashboard.jsx
│ │ ├── MessOwnerDashboard.jsx
│ │ ├── HostelCard.jsx
│ │ ├── MessCard.jsx
│ │ └── ImageUpload.jsx
│ ├── App.jsx
│ ├── main.jsx
│ └── index.css
└── index.html
├── package.json
└── vite.config.js
└── server/ # Node.js backend
 ├── index.js
 ├── package.json
 └── .env
```

## Step 1: Database Setup

### 1.1 Connect to your RDS PostgreSQL instance

```
bash
```

```
psql -h your-rds-endpoint.amazonaws.com -U postgres -d postgres
```

## **1.2 Run the schema (updated without admin role)**

sql

```

-- Create types
CREATE TYPE user_role AS ENUM ('student', 'hostel_owner', 'mess_owner');

-- Users table
CREATE TABLE users (
 user_id SERIAL PRIMARY KEY,
 full_name VARCHAR(100) NOT NULL,
 email VARCHAR(150) UNIQUE NOT NULL,
 password_hash VARCHAR(255) NOT NULL,
 phone_number VARCHAR(20),
 role user_role NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Hostels table
CREATE TABLE hostels (
 hostel_id SERIAL PRIMARY KEY,
 owner_id INT REFERENCES users(user_id) ON DELETE CASCADE,
 name VARCHAR(100) NOT NULL,
 city VARCHAR(50) NOT NULL,
 address TEXT NOT NULL,
 description TEXT,
 main_image_url TEXT,
 wifi_available BOOLEAN DEFAULT FALSE,
 generator_available BOOLEAN DEFAULT FALSE
);

-- Rooms table
CREATE TABLE rooms (
 room_id SERIAL PRIMARY KEY,
 hostel_id INT REFERENCES hostels(hostel_id) ON DELETE CASCADE,
 room_type VARCHAR(50) NOT NULL,
 price_per_month DECIMAL(10, 2) NOT NULL,
 total_beds INT NOT NULL,
 available_beds INT NOT NULL,
 has_attached_bath BOOLEAN DEFAULT TRUE
);

-- Mess services table
CREATE TABLE mess_services (
 mess_id SERIAL PRIMARY KEY,
 owner_id INT REFERENCES users(user_id) ON DELETE CASCADE,
 name VARCHAR(100) NOT NULL,
 city VARCHAR(50) NOT NULL,
 monthly_price DECIMAL(10, 2) NOT NULL,
 delivery_radius_km DECIMAL(4, 1)
);

-- Room bookings table
CREATE TABLE room_bookings (
 booking_id SERIAL PRIMARY KEY,
 student_id INT REFERENCES users(user_id),
 room_id INT REFERENCES rooms(room_id),
 start_date DATE NOT NULL,
 status VARCHAR(20) DEFAULT 'pending',
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Mess subscriptions table
CREATE TABLE mess_subscriptions (
 subscription_id SERIAL PRIMARY KEY,
 student_id INT REFERENCES users(user_id),
 mess_id INT REFERENCES mess_services(mess_id),
 start_date DATE NOT NULL,
 is_active BOOLEAN DEFAULT TRUE,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

## Step 2: Backend Setup

### 2.1 Create server directory

```
bash

mkdir server && cd server
npm init -y
```

### 2.2 Install dependencies

```
bash

npm install express pg cors bcrypt serverless-http @aws-sdk/client-s3 @aws-sdk/s3-request-presigner dotenv
```

### 2.3 Create .env file

```
env

DB_USER=postgres
DB_HOST=your-rds-endpoint.amazonaws.com
DB_PASSWORD=your-password
BUCKET_NAME=your-s3-bucket-name
```

### 2.4 Copy the updated server/index.js file

(Use the code from the "Updated Backend - server/index.js" artifact)

### 2.5 Create package.json scripts

```
json

{
 "scripts": {
 "start": "node index.js",
 "test": "node index.js"
 }
}
```

## Step 3: AWS S3 Setup

### 3.1 Create S3 Bucket

1. Go to AWS S3 Console
2. Create a new bucket (e.g., staymate-images)
3. Enable public access for uploaded images:
  - Uncheck "Block all public access"
  - Add bucket policy:

```
json

{
 "Version": "2012-10-17",
 "Statement": [
 {
 "Sid": "PublicReadGetObject",
 "Effect": "Allow",
 "Principal": "*",
 "Action": "s3:GetObject",
 "Resource": "arn:aws:s3:::staymate-images/uploads/*"
 }
]
}
```

### 3.2 Enable CORS

Add CORS configuration:

```
json

[
 {
 "AllowedHeaders": ["*"],
 "AllowedMethods": ["GET", "PUT", "POST"],
 "AllowedOrigins": ["*"],
 "ExposeHeaders": []
 }
]
```

## 🚀 Step 4: Deploy Backend to AWS Lambda

### 4.1 Zip your backend code

```
bash

cd server
zip -r function.zip .
```

### 4.2 Upload to Lambda

1. Go to AWS Lambda Console
2. Update your existing function
3. Upload function.zip
4. Set environment variables in Lambda:
  - DB\_USER
  - DB\_HOST
  - DB\_PASSWORD
  - BUCKET\_NAME

### 4.3 Update API Gateway

- Ensure your API Gateway is connected to the Lambda function
- Note your API Gateway URL (e.g., <https://wz81rzb6g4.execute-api.us-east-1.amazonaws.com>)

## 💻 Step 5: Frontend Setup

### 5.1 Create React app directory

```
bash

cd client
npm install
```

### 5.2 Add your logo

1. Create/download your logo image
2. Place it in `client/public/logo.png`
3. Make sure it's 500x500px or similar square dimensions for best results

### 5.3 Update API\_URL in App.jsx

Replace with your actual API Gateway URL:

```
javascript

const API_URL = "https://YOUR-API-ID.execute-api.us-east-1.amazonaws.com";
```

## 5.4 Run development server

```
bash
npm run dev
```

## 🎨 Step 6: UI Customization

### Color Scheme

The application uses:

- **Primary Blue:** #1a3a5c (Logo text, headers)
- **Accent Orange:** #ff8c42 (Buttons, highlights)
- **Success Green:** #28a745 (Mess services)
- **White:** #ffffff (Navbar background)

### Logo Placement

- Landing Page: 200x200px
- Navbar: 50x50px
- Auth Page: 80x80px

## 🧪 Step 7: Testing

### Test Scenarios:

#### 1. Authentication

- Sign up as student
- Sign up as hostel owner
- Sign up as mess owner
- Login with each account

#### 2. Hostel Owner Flow

- Create hostel
- Upload image for hostel
- Edit hostel details
- Add rooms to hostel
- Delete rooms
- Delete hostel

#### 3. Mess Owner Flow

- Create mess service
- Upload image for mess
- Edit mess details
- Delete mess service

#### 4. Student Flow

- Browse hostels and messes
- Filter by location and price
- Book hostel room
- Subscribe to mess service

## 🚀 Step 8: Production Deployment

### Frontend (Netlify/Vercel)

#### Using Netlify:

```
bash
cd client
npm run build
Drag and drop 'dist' folder to Netlify
```

### Using Vercel:

```
bash
cd client
vercel --prod
```

### Environment Variables for Production:

Set these in your hosting platform:

```
VITE_API_URL=https://your-api-gateway-url.amazonaws.com
```

### Security Checklist

- Database credentials are in environment variables
- S3 bucket has proper CORS configuration
- API Gateway has rate limiting enabled
- Password hashing is working (bcrypt)
- SQL injection protection (parameterized queries)
- Input validation on both frontend and backend

### Troubleshooting

#### Common Issues:

1. **"Failed to connect to database"**
  - Check RDS security group allows Lambda IP
  - Verify credentials in .env file
2. **"Image upload fails"**
  - Check S3 bucket CORS configuration
  - Verify bucket name in environment variables
  - Check Lambda has S3 permissions
3. **"Cannot create hostel/mess"**
  - Check browser console for errors
  - Verify API Gateway URL is correct
  - Check Lambda logs in CloudWatch
4. **"Login works but shows error"**
  - This is fixed in the updated Auth.jsx component
  - Make sure you're using the latest code

### Mobile Responsiveness

The app is fully responsive and tested on:

- Desktop (1920x1080)
- Tablet (768px)
- Mobile (375px)

## Features Summary

### Completed Features:

1. User authentication (Login/Signup)
2. Role-based dashboards (Student, Hostel Owner, Mess Owner)
3. CRUD operations for hostels and messes
4. Room management system for hostels
5. Image upload to AWS S3
6. Professional landing page
7. Search and filter functionality
8. Booking system for students
9. Responsive design
10. Modern UI with proper branding

### Future Enhancements:

- Admin dashboard
- Payment integration
- Reviews and ratings
- Chat system
- Email notifications
- Analytics dashboard

## Support

If you encounter any issues:

1. Check browser console for errors
2. Check Lambda CloudWatch logs
3. Verify database connections
4. Check S3 permissions

## Congratulations!

Your StayMate application is now fully set up with all the latest features!

Remember to:

- Keep your credentials secure
- Regular database backups
- Monitor AWS costs
- Update dependencies regularly

## 🔍 Debugging Guide - Fixing Your Issues

### Issue 1: Hostels Not Displaying (Data in DB but not showing)

#### Step 1: Check Browser Console

Open browser console (F12) and look for:

```
HostelOwnerDashboard: Hostels fetched: X
```

#### Step 2: Check Network Tab

1. Open DevTools → Network tab
2. Filter by "Fetch/XHR"
3. Look for request to `/hostels/owner/[your-id]`
4. Check the response - should be:

```
json
```

```
{
 "success": true,
 "hostels": [...]
}
```

#### Step 3: Verify Data Structure

The issue might be here in your dashboard component. Make sure it's checking `(data.success)`:

```
javascript
```

```
const response = await fetch(`${apiUrl}/hostels/owner/${user.user_id || user.id}`);
const data = await response.json();

if (data.success) { //← Make sure this line exists
 setHostels(data.hostels || []);
}
```

#### Quick Fix:

Update your `HostelOwnerDashboard.jsx` fetch function:

```
javascript
```

```

const fetchHostels = async () => {
 try {
 setLoading(true);
 const response = await fetch(`/${apiUrl}/hostels/owner/${user.user_id || user.id}`);

 console.log('Response status:', response.status);

 const data = await response.json();
 console.log('Response data:', data);

 if (response.ok && data.success) {
 console.log('Hostels fetched:', data.hostels.length);
 setHostels(data.hostels || []);
 } else {
 console.error('Failed to fetch hostels:', data.message);
 alert('Failed to load hostels: ' + (data.message || 'Unknown error'));
 }
 } catch (error) {
 console.error('Error fetching hostels:', error);
 alert('Error: ' + error.message);
 } finally {
 setLoading(false);
 }
};

```

## Issue 2: Mess Creation Error (HTML instead of JSON)

### The Error Explained:

Error: Unexpected token '<', "<!DOCTYPE "... is not valid JSON

This means your Lambda is returning an HTML error page instead of JSON. This happens when:

1. **The route doesn't exist** in Lambda
2. **Lambda has an error** before it can return JSON
3. **The function hasn't been redeployed** with new code

### Step 1: Verify Lambda Has Latest Code

1. **Check if POST /messes exists in your Lambda:**

```

bash

In your server directory
grep -n "app.post('/messes'" index.js

```

You should see a line number. If not, the route is missing.

2. **Redeploy to Lambda:**

```

bash

cd server
rm -rf node_modules package-lock.json
npm install
zip -r function.zip .
Upload to Lambda via AWS Console

```

### Step 2: Test API Endpoint Directly

Use Postman or curl to test:

```

bash

```

```
curl -X POST https://YOUR-API-URL.amazonaws.com/messages \
-H "Content-Type: application/json" \
-d '{
 "owner_id": 1,
 "name": "Test Mess",
 "city": "Lahore",
 "monthly_price": 5000,
 "delivery_radius_km": 2.5
}'
```

#### Expected Response:

```
json

{
 "success": true,
 "message": "Mess service added successfully!",
 "mess": { ... }
}
```

If you get HTML: Your Lambda doesn't have the route.

#### Step 3: Check CloudWatch Logs

1. Go to AWS Lambda Console
2. Click on your function
3. Go to "Monitor" → "View logs in CloudWatch"
4. Look for errors when you try to create a mess

#### Step 4: Verify Request Body

Add logging to your frontend before the fetch:

```
javascript
```

```

const handleSubmit = async () => {
 const payload = {
 ...formData,
 owner_id: user.user_id || user.id,
 monthly_price: parseFloat(formData.monthly_price),
 delivery_radius_km: formData.delivery_radius_km ? parseFloat(formData.delivery_radius_km) : null
 };

 console.log('==== SUBMITTING MESS ====');
 console.log('URL:', `${apiUrl}/messages`);
 console.log('Payload:', payload);

 try {
 const response = await fetch(`${apiUrl}/messages`, {
 method: 'POST',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(payload)
 });

 console.log('Response status:', response.status);
 console.log('Response headers:', response.headers);

 const contentType = response.headers.get('content-type');
 console.log('Content-Type:', contentType);

 if (!contentType || !contentType.includes('application/json')) {
 const text = await response.text();
 console.error('Received HTML instead of JSON:', text);
 throw new Error('Server returned HTML instead of JSON. Check Lambda deployment.');
 }

 const data = await response.json();
 console.log('Response data:', data);

 if (data.success) {
 alert(data.message);
 fetchMessages();
 resetForm();
 } else {
 alert('Error: ' + data.message);
 }
 } catch (error) {
 console.error('Full error:', error);
 alert('Error: ' + error.message);
 }
};

```

## Quick Fixes Checklist

### Fix #1: Update Backend

1. Replace your entire `server/index.js` with the code from "Fixed Backend with Better Error Handling" artifact
2. Redeploy to Lambda:

```

bash

cd server
npm install
zip -r function.zip .
Upload to AWS Lambda

```

### Fix #2: Verify Environment Variables in Lambda

Make sure these are set:

- `DB_USER`
- `DB_HOST`
- `DB_PASSWORD`
- `BUCKET_NAME`

### Fix #3: Test Each Endpoint

Test in this order:

```
bash

1. Health check
curl https://YOUR-API.amazonaws.com/

2. Get all messes
curl https://YOUR-API.amazonaws.com/messes

3. Create mess (should work now)
curl -X POST https://YOUR-API.amazonaws.com/messes \
-H "Content-Type: application/json" \
-d '{"owner_id":1,"name":"Test","city":"Lahore","monthly_price":5000}'
```

### Fix #4: Check API Gateway Configuration

1. Go to API Gateway Console
  2. Find your API
  3. Click on `(/messes)` resource
  4. Verify POST method exists
  5. Click "Deploy API" → Deploy to your stage
- 

## Common Causes & Solutions

### Problem: Lambda returns 502 Bad Gateway

**Solution:** Your Lambda function has a runtime error. Check CloudWatch logs.

### Problem: CORS error in browser

**Solution:** Make sure `app.use(cors())` is BEFORE `app.use(express.json())` in your backend.

### Problem: Data saves but doesn't display

**Solution:** Frontend isn't refetching data after save. Add:

```
javascript

if (data.success) {
 alert(data.message);
 await fetchMesses(); //← Make sure this is called
 resetForm();
}
```

### Problem: "Route not found" error

**Solution:**

1. Check your API Gateway URL is correct
  2. Make sure there's no trailing slash
  3. Verify the route exists in Lambda
-

## Emergency Debug Mode

Add this to your `MessOwnerDashboard.jsx`:

```
javascript

const fetchMesses = async () => {
 try {
 console.log('🔍 Fetching messes from:', `${apiUrl}/messes/owner/${user.user_id || user.id}`);
 setLoading(true);

 const response = await fetch(`${apiUrl}/messes/owner/${user.user_id || user.id}`);
 console.log('📊 Response status:', response.status);
 console.log('✅ Response ok:', response.ok);

 const data = await response.json();
 console.log('📦 Response data:', data);

 if (data.success) {
 console.log('✓ Messes loaded:', data.messes.length);
 setMesses(data.messes || []);
 } else {
 console.error('✗ Failed:', data.message);
 }
 } catch (error) {
 console.error('✖ Error:', error);
 } finally {
 setLoading(false);
 }
};
```

---

## Still Not Working?

**Check These:**

**1. Lambda Function:**

- Has latest code?
- Environment variables set?
- Execution role has RDS permissions?

**2. RDS Database:**

- Security group allows Lambda IP?
- Table `mess_services` exists?
- Can connect from Lambda?

**3. API Gateway:**

- POST /messes method exists?
- Lambda proxy integration enabled?
- API deployed to stage?

**4. Frontend:**

- `API_URL` is correct?
- No typos in endpoint names?
- Handling response correctly?

---

## Test SQL Directly

Connect to your RDS and run:

```
sql
```

```
-- Check if mess_services table exists
SELECT * FROM mess_services;

-- Manually insert a test mess
INSERT INTO mess_services (owner_id, name, city, monthly_price, delivery_radius_km)
VALUES (1, 'Test Mess', 'Lahore', 5000, 2.5);

-- Check if it was inserted
SELECT * FROM mess_services;
```

If manual insert works but API doesn't, the problem is in Lambda.

---

## Next Steps

1. Replace backend code with fixed version
2. Redeploy Lambda function
3. Test with curl/Postman
4. Then test in browser
5. Check CloudWatch logs for any errors
6. Report back what you find!

## Quick Fix Checklist - Do These NOW

### ISSUE 1: Hostels Not Displaying (Data in DB but not on frontend)

#### Steps to Fix:

1. Open browser console (F12)
2. Go to your Hostel Owner dashboard
3. Look for these logs:

```
 Fetching hostels for owner: [number]
 Response status: 200
 Hostels loaded: X
```

If you don't see these logs:

- Replace your `HostelOwnerDashboard.jsx` with the "Fixed Hostel Owner Dashboard" artifact
- Refresh the page

If you see logs but no hostels display:

- Check if `(data.success)` is `(true)` in the response
- Check if `(data.hostels)` is an array
- The issue is in how the data is being set to state

---

### ISSUE 2: Mess Creation Failing (HTML error popup)

#### The Problem:

Your Lambda function is returning HTML instead of JSON because:

- The `(POST /messes)` route doesn't exist in Lambda
- Or Lambda hasn't been redeployed with the new code

#### Steps to Fix:

##### STEP 1: Update Backend Code

```
bash
cd server
```

Replace `(index.js)` with the "Fixed Backend with Better Error Handling" artifact.

##### STEP 2: Reinstall Dependencies

```
bash
rm -rf node_modules package-lock.json
npm install
```

##### STEP 3: Create Deployment Package

```
bash
Make sure you're in the server directory
zip -r function.zip .

Verify the zip contains everything
unzip -l function.zip | head -20
```

You should see:

- index.js
- node\_modules/
- package.json
- .env (optional, use Lambda env vars instead)

#### STEP 4: Upload to Lambda

##### Option A: AWS Console

1. Go to AWS Lambda Console
2. Select your function
3. Click "Upload from" → ".zip file"
4. Select `function.zip`
5. Click "Save"

##### Option B: AWS CLI

```
bash
aws lambda update-function-code \
--function-name your-function-name \
--zip-file fileb://function.zip
```

#### STEP 5: Verify Environment Variables in Lambda

Click "Configuration" → "Environment variables"

Make sure these are set:

- `DB_USER` = postgres
- `DB_HOST` = your-rds-endpoint.amazonaws.com
- `DB_PASSWORD` = your-password
- `BUCKET_NAME` = your-s3-bucket-name

#### STEP 6: Test the Endpoint

Using curl:

```
bash
curl -X POST https://YOUR-API-URL.amazonaws.com/messes \
-H "Content-Type: application/json" \
-d '{
 "owner_id": 1,
 "name": "Test Mess",
 "city": "Lahore",
 "monthly_price": 5000
}'
```

Expected Response:

```
json
{
 "success": true,
 "message": "Mess service added successfully!",
 "mess": { ... }
}
```

If you get HTML: The route still doesn't exist. Check CloudWatch logs.

#### STEP 7: Update Frontend Components

Replace these files:

- `src/components/MessOwnerDashboard.jsx` → Use "Fixed Mess Owner Dashboard" artifact
- `src/components/HostelOwnerDashboard.jsx` → Use "Fixed Hostel Owner Dashboard" artifact

#### ✓ STEP 8: Test in Browser

1. **Open browser console (F12)**
2. **Go to Mess Owner dashboard**
3. **Try to add a mess**
4. **Watch the console for logs:**

You should see:

```
==== SUBMITTING MESS ====
Method: POST
URL: https://your-api.amazonaws.com/messes
Payload: { owner_id: 1, name: "...", ... }
 Response status: 200
 Response data: { success: true, ... }
 ✓ Mess saved successfully
```

If you see:

```
✗ Received HTML instead of JSON
```

Then Lambda still doesn't have the route. Go back to Step 4.

---

#### 🔍 Quick Debug Commands

**Check if mess\_services table exists:**

```
sql
SELECT * FROM mess_services;
```

**Check backend logs:**

```
bash
In Lambda CloudWatch Logs
Look for:
POST /messes called with body: { ... }
```

**Test all endpoints:**

```
bash
Health check (should always work)
curl https://YOUR-API-URL.amazonaws.com/

Get messes (should return empty array if none exist)
curl https://YOUR-API-URL.amazonaws.com/messes

Get hostels (should return your hostels)
curl https://YOUR-API-URL.amazonaws.com/hostels
```

#### ⌚ Priority Order

Do these in order:

1.  **Update backend** (index.js)
2.  **Redeploy to Lambda** (zip + upload)

3.  **Test with curl** (verify endpoint works)
  4.  **Update frontend components** (both dashboards)
  5.  **Test in browser** (check console logs)
- 

## If Still Not Working

### For Hostels Not Displaying:

```
javascript

// Add this temporarily to HostelOwnerDashboard
useEffect(() => {
 console.log('Current hostels state:', hostels);
 console.log('Hostels length:', hostels.length);
}, [hostels]);
```

### For Mess Creation Error:

1. Check Lambda CloudWatch logs
  2. Look for ANY error messages
  3. Verify the route exists: `grep "app.post('/messes'" index.js`
  4. Make sure Lambda has been saved after upload
- 

## Success Indicators

You'll know it's working when:

### Hostels:

- Console shows:  Hostels loaded: X
- Cards appear on dashboard
- Can edit, delete, manage rooms

### Messes:

- Console shows:  Mess saved successfully
  - Alert shows success message
  - Mess appears in list
  - Can edit and delete
- 

## Next Steps if Still Broken

If after all this it STILL doesn't work:

1. **Take screenshots of:**
  - Browser console (all errors)
  - Network tab (failed requests)
  - Lambda CloudWatch logs
  - Your API Gateway configuration
2. **Share the console output from:**

```
bash

grep -n "app.post('/messes'" server/index.js
```

3. **Test and share result of:**

```
bash
```

```
curl -X POST https://YOUR-API.amazonaws.com/messages \
-H "Content-Type: application/json" \
-d '{"owner_id":1,"name":"Test","city":"Lahore","monthly_price":5000}'
```

Good luck! 