Project Report: AWS Lambda JSON Storage with AngularJS Frontend

# Table of Contents

1. 1. Project Overview
2. 2. API and AWS Information
3. 3. Backend Implementation
4. 4. Frontend Implementation
5. 5. Development Thought Process
6. 6. Results and Screenshots

# 1. Project Overview

The task is to create a service where users can store and retrieve JSON data using AWS Lambda and S3, with an AngularJS frontend to interact with the API.

# 2. API and AWS Information

### AWS Setup:

- Create an S3 bucket\*\* to store JSON data.  
- Use API Gateway\*\* to route HTTP requests to the Lambda function.  
- Ensure that the Lambda function has the appropriate IAM roles to access the S3 bucket.

### IAM Policies:

The IAM role assigned to the Lambda function includes permissions to PutObject, GetObject, and ListBucket in S3.

# 3. Backend Implementation

The backend is built using AWS Lambda and Node.js. The Lambda function has two key endpoints:

* POST Endpoint\*\*: Accepts JSON data and stores it in S3.
* GET Endpoint\*\*: Retrieves all stored JSON data from S3.

### Example POST Implementation:

async function handlePostRequest(event) {  
 const body = JSON.parse(event.body);  
 const params = {  
 Bucket: BUCKET\_NAME,  
 Key: `${randomUUID()}.json`,  
 Body: JSON.stringify(body),  
 ContentType: 'application/json'  
 };  
 const result = await s3Client.send(new PutObjectCommand(params));  
 return {  
 statusCode: 200,  
 body: JSON.stringify({ e\_tag: result.ETag, url: `https://${BUCKET\_NAME}.s3.amazonaws.com/${filename}` })  
 };  
}

# 4. Frontend Implementation

The frontend is built with AngularJS and consists of two key components:

* DataFormComponent : Collects user input and sends it to the backend.
* DataListComponent : Fetches and displays stored data.

### Example Form Handling:

onSubmit() {  
 this.apiService.postData({ name: this.name, age: this.age }).subscribe(  
 (response) => {  
 this.message = 'Data submitted successfully!';  
 },  
 (error) => {  
 this.showError('Error submitting data.');  
 }  
 );  
}

# 5. Development Thought Process

**1. Backend Thought Process:**

The backend is built using AWS Lambda and is structured for flexibility and scalability using serverless architecture. The design choices primarily focus on leveraging AWS services to ensure the application is scalable, cost-efficient, and easy to maintain.

**Validation:**

User input is validated on both the **name** and **age** fields to ensure data integrity and avoid potential issues with malformed or incorrect data.

* **Name Validation**: The validation logic ensures that the name is provided and meets specific constraints:
  + **Required field**: The name cannot be empty.
  + **Length constraints**: The name must be between **2 and 50 characters**.
  + **Allowed characters**: Only alphabetic characters, spaces, and hyphens are permitted (to support names like "John Doe" or "Anne-Marie").
  + **Error messages**: If the input fails any of the checks, a specific error message is returned, making it easy to notify users of the issue.

Here's the name validation function:  
  
  
Code:

function validateName(name) {

if (typeof name !== 'string' || name.trim() === '') {

return { isValid: false, error: 'Name is required' };

}

const trimmedName = name.trim();

if (trimmedName.length < 2) {

return { isValid: false, error: 'Name must be at least 2 characters long' };

}

if (trimmedName.length > 50) {

return { isValid: false, error: 'Name must not exceed 50 characters' };

}

const nameRegex = /^[A-Za-z]+(?:[-\s][A-Za-z]+)\*$/;

if (!nameRegex.test(trimmedName)) {

return { isValid: false, error: 'Name can only contain alphabetic characters, spaces, and hyphens' };

}

return { isValid: true };

}

* **Age Validation**: The age validation ensures that the user provides a valid number within a realistic range:
  + **Required field**: Age is required.
  + **Whole number**: The age must be an integer.
  + **Range check**: The age must be between **1 and 120** to account for realistic age limits.

Here's the age validation function:

function validateAge(age) {

if (age === undefined || age === null) {

return { isValid: false, error: 'Age is required' };

}

if (typeof age !== 'number' || !Number.isInteger(age)) {

return { isValid: false, error: 'Age must be a whole number' };

}

if (age < 1) {

return { isValid: false, error: 'Age must be at least 1' };

}

if (age > 120) {

return { isValid: false, error: 'Age must not exceed 120' };

}

return { isValid: true };

}

The logic ensures the application only accepts valid and meaningful user data, preventing potential issues such as storing incomplete or erroneous records. Validation checks like these also reduce the chances of incorrect or malicious data being stored in S3, making the backend more robust and secure.

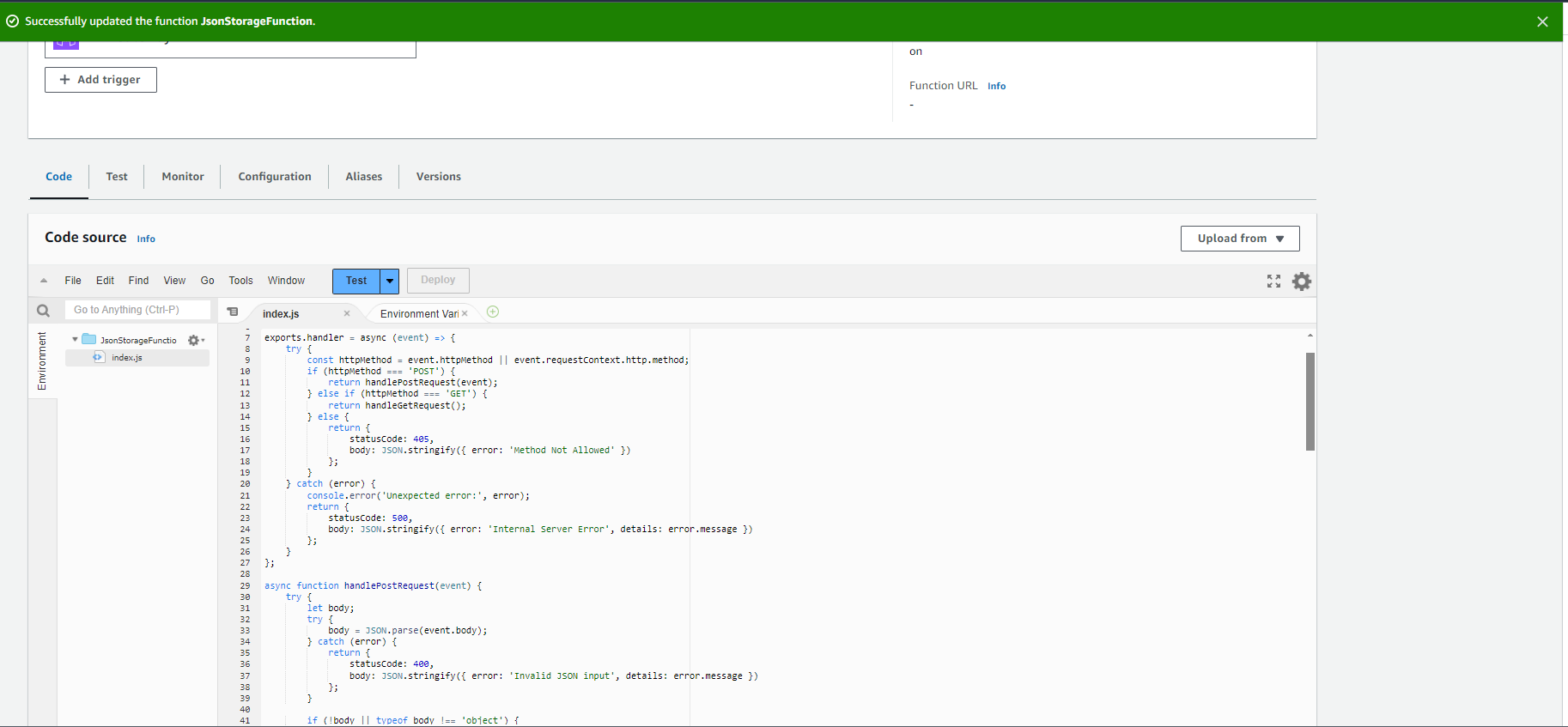
**2. Frontend Thought Process:**

The frontend interacts with the backend through an AngularJS framework. Since the backend requires validated input for both name and age, the frontend performs basic form validation before submitting the data. This reduces unnecessary API calls and provides instant feedback to the user if something is wrong.

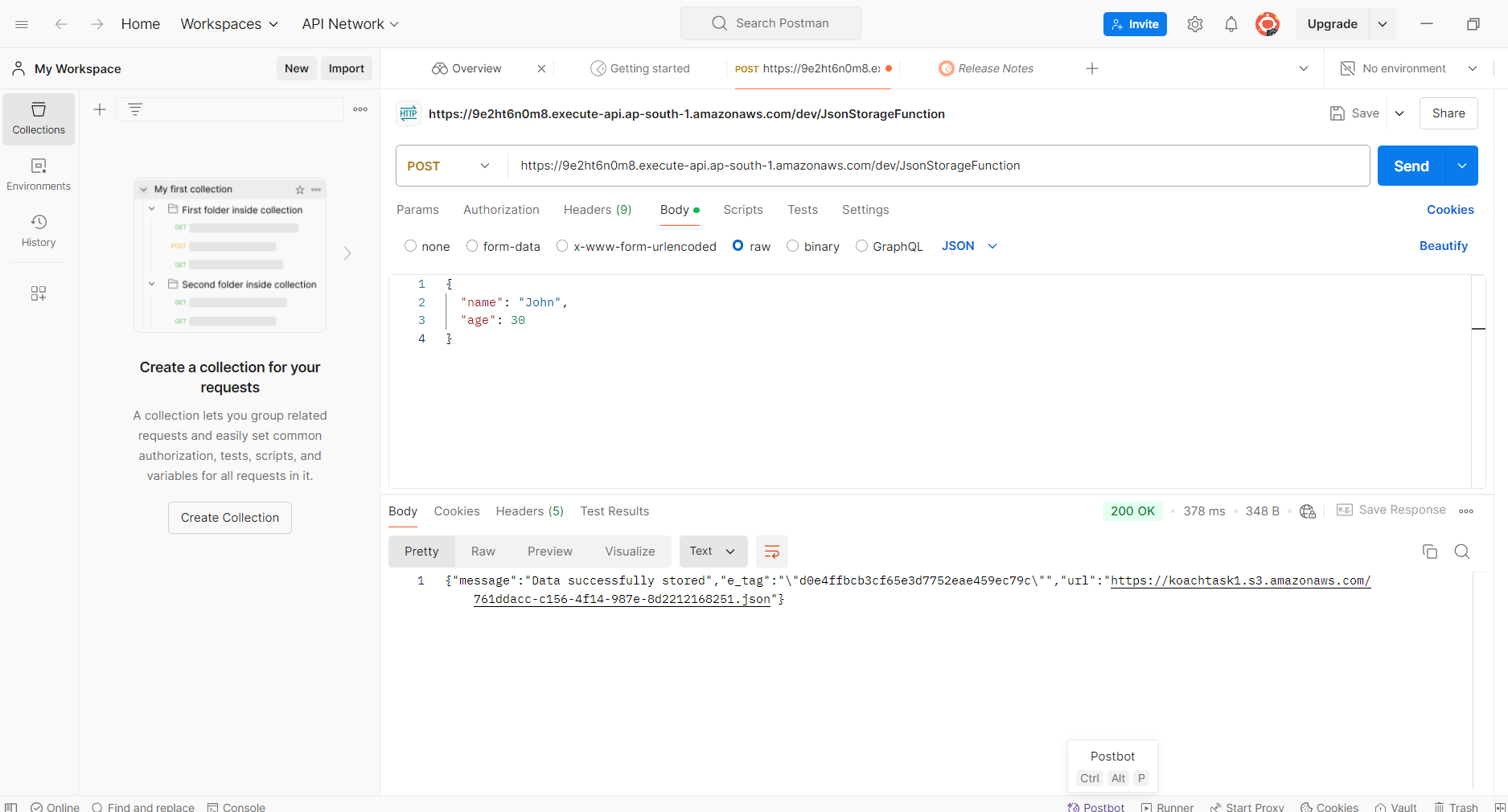
# 6. Results and Screenshots

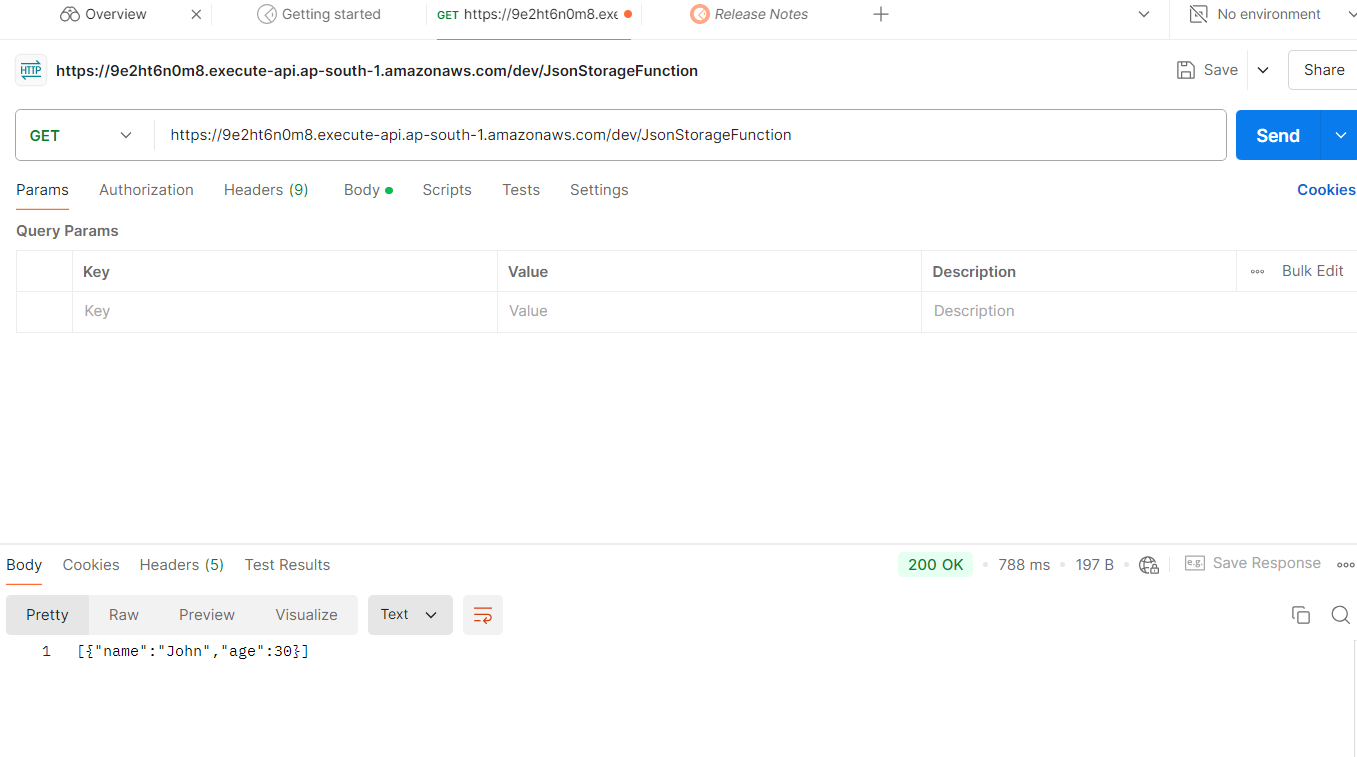
Screenshots of the frontend interface showing form submission and data retrieval will be included here.

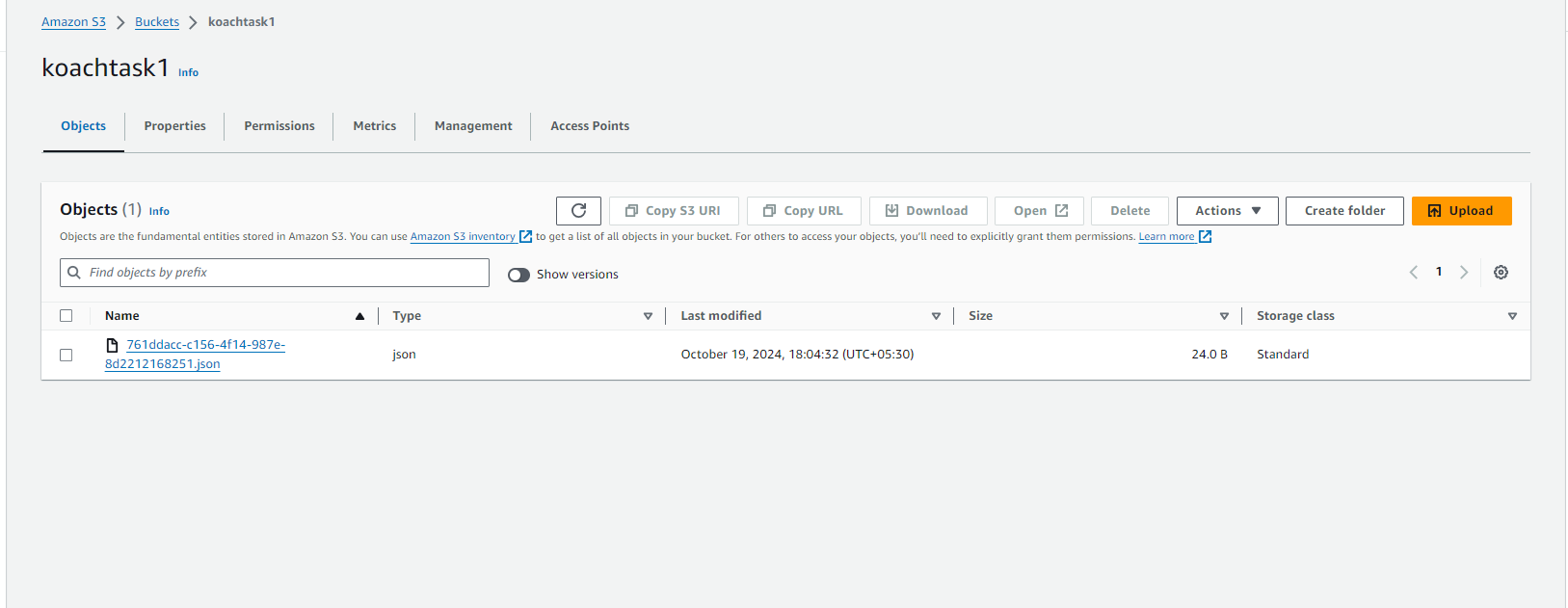
1. AWS Backend Images:

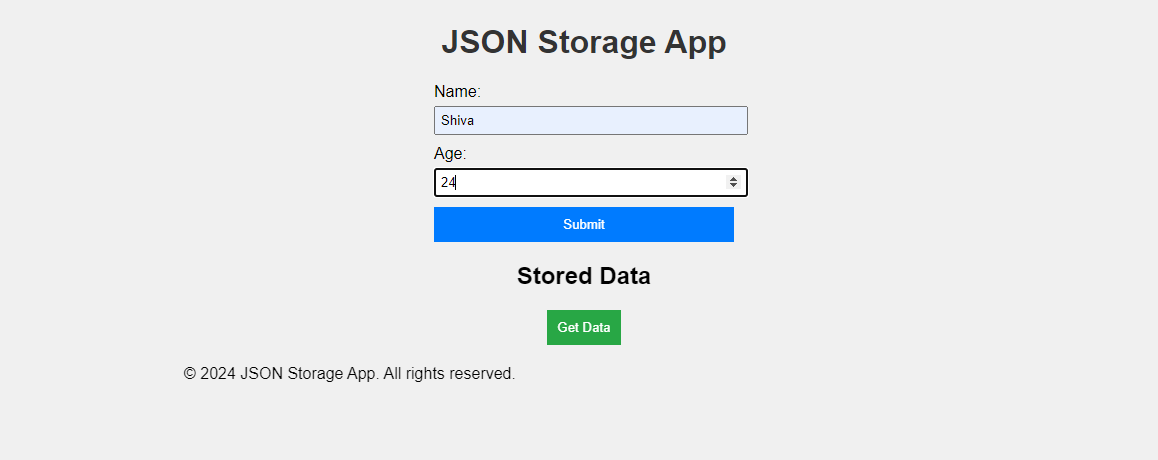
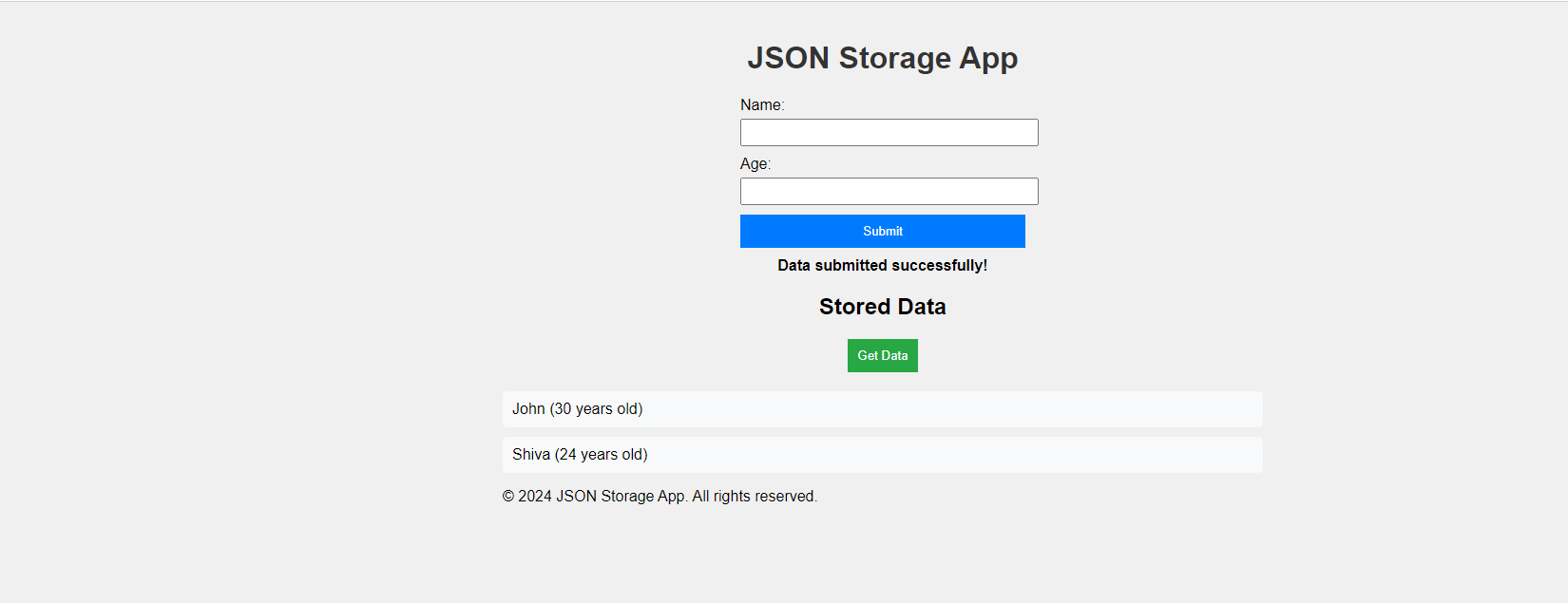
Part 1: Lambda Code Deployed Successfully  
  


Part 2: API endpoint for POST and GET Requests -   
API endpoint: [**https://9e2ht6n0m8.execute-api.ap-south-1.amazonaws.com/dev/JsonStorageFunction**](https://9e2ht6n0m8.execute-api.ap-south-1.amazonaws.com/dev/JsonStorageFunction)

Case1: Post Request Send Through Postman - {"name": "John", "age": 30}  
  


Case 2: GET Request Send through Postman   
  


Case 3: S3 Image  
  


1. Frontend Images  
     
   Case 1 Storing Data  
     
     
     
     
     
     
     
     
     
     
     
   Case 2 Getting Data  
     
     
     
   Case 3 Validation check and error handling  
     
   