**Programming and Application**

**By Devi Sri Charan**

[Upgrade Claims Management System to Stateful application 1](#_Toc582823068)

[Create a New Feature Branch 2](#_Toc1793132124)

[Database System Selection 2](#_Toc1000360783)

[Database Schema Design and Implementation 3](#_Toc1430128813)

[Collections and Their Schemas 3](#_Toc633029542)

[1. Policyholders Collection 3](#_Toc381177357)

[2. Policies Collection 4](#_Toc1797485662)

[3. Claims Collection 4](#_Toc1905513420)

[Indexing Strategy 5](#_Toc1423001978)

[Integrating with Node.js 5](#_Toc849066143)

[Creating Mongoose Models 5](#_Toc2044613613)

[Application Modification 6](#_Toc1945102540)

[Step 1: Connect to MongoDB 6](#_Toc1042593160)

[Step 2: Update API Endpoints 7](#_Toc1654289618)

[Error Handling and Data Validation 10](#_Toc1025949053)

[Updated Schemas with Validation 10](#_Toc1871867232)

[Policy Schema 10](#_Toc43994706)

[Claim Schema 11](#_Toc1272195278)

[Updated API Endpoints with Validation and Error Handling 12](#_Toc81409338)

[Policyholder 12](#_Toc540065903)

[Policy 14](#_Toc777342511)

[Claim 15](#_Toc225750830)

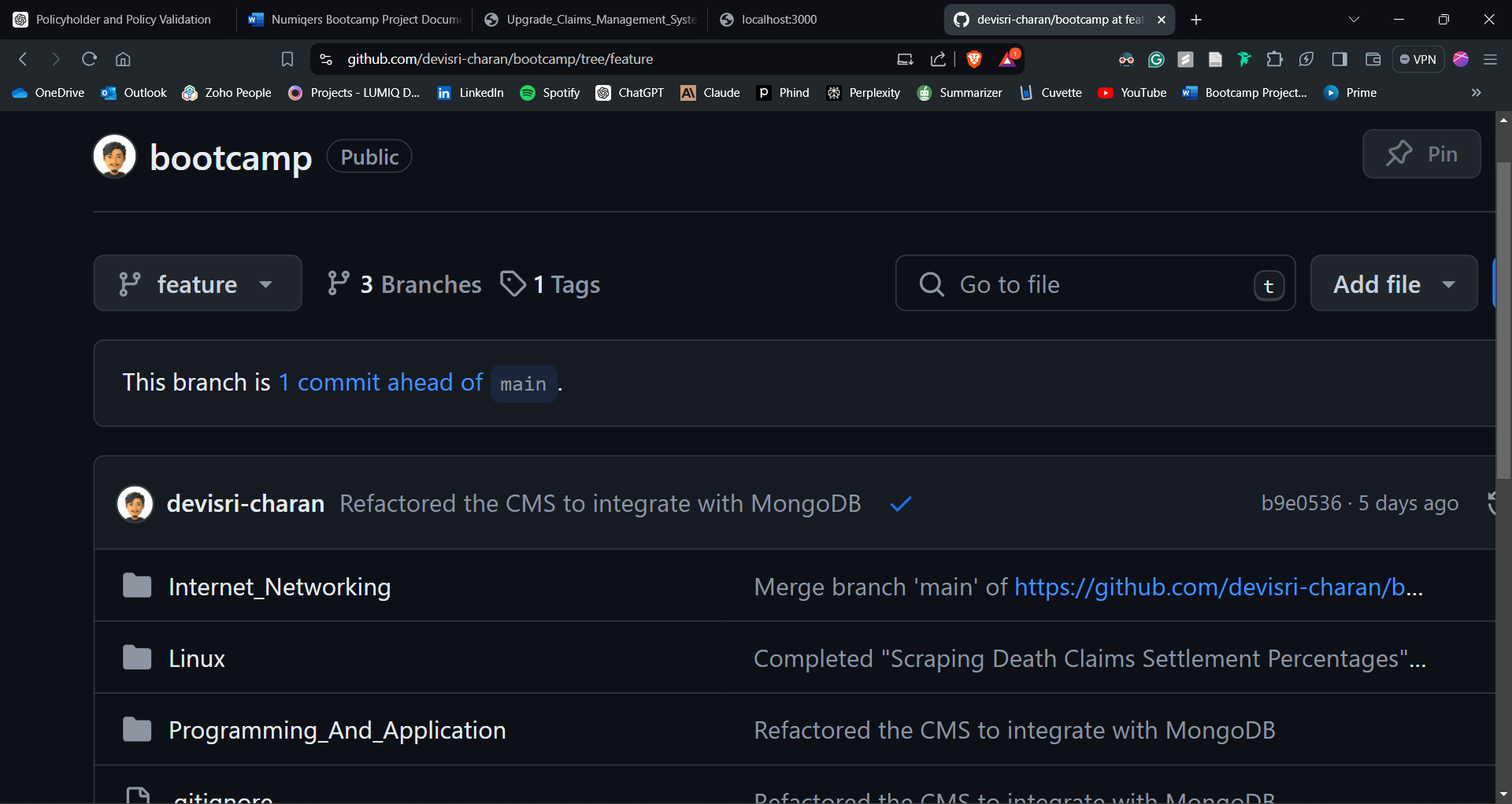
[Pull Request (PR) for Merging 16](#_Toc243365018)

[Deploy on cloud 17](#_Toc77567336)

# Upgrade Claims Management System to Stateful application

## Create a New Feature Branch

I have created a new branch using `git branch feature` command



Database System Selection

For the Claims Management System, MongoDB was selected as the database system due to its flexibility, scalability, and robust handling of diverse data types and structures. MongoDB, being a NoSQL database, excels in scenarios where the data schema may evolve over time, as it inherently supports schema-less data storage. This feature is particularly beneficial in developing systems like the Claims Management System, where adjustments to data attributes (such as adding new types of claims or policy details) might be needed as the system grows and adapts to business requirements.

Unlike relational databases like PostgreSQL, MongoDB stores data in BSON (Binary JSON) format, which can more naturally capture the hierarchical relationships between policies, policyholders, and claims without the need for complex joins and relational mappings. This can lead to performance benefits in data retrieval and updates, as data that is logically related can be stored together in the same document.

MongoDB also offers powerful query capabilities and indexing features that make it well-suited for handling large volumes of data, which is typical in an insurance or claims management context. The ability to scale out horizontally using sharding is another advantage of MongoDB, providing the capability to distribute data across multiple servers as data volume and throughput grow. This scalability is a critical consideration for a Claims Management System that may need to handle increasing loads of claim processing as the user base expands.

Moreover, MongoDB's document model can be more developer-friendly due to its JSON-like format, making it easier to map application objects to database records without the overhead of object-relational impedance mismatch, which is commonly encountered in systems using traditional SQL databases. This can lead to faster development and fewer errors as developers can work with a more intuitive data representation.

In summary, the choice of MongoDB for the Claims Management System is driven by its flexibility in data schema design, performance efficiency in handling complex nested data, scalability to support growth, and ease of use for developers, making it an optimal choice for dynamic and evolving systems like insurance and claims management.

## Database Schema Design and Implementation

To implement a MongoDB database schema for the Claims Management System, we will design three collections: policyholders, policies, and claims.

### Collections and Their Schemas

#### 1. Policyholders Collection

This collection will store information about the policyholders. Here’s a schema design:

{

"policyholder\_id": "string",

"name": "string",

"date\_of\_birth": "date",

"address": "string",

"phone": "string"

}

**Explanation**:

* **policyholder\_id**: A unique identifier for the policyholder. This can be indexed for faster query performance.
* **name**: Name of the policyholder.
* **date\_of\_birth**: The birth date of the policyholder. Stored as a date object.
* **address**: Address of the policyholder.
* **phone**: Contact number of the policyholder.

#### 2. Policies Collection

This collection holds details about the policies.

{

"policy\_id": "string",

"policyholder\_id": "string",

"start\_date": "date",

"end\_date": "date",

"premium": "number",

"coverage": "number",

"payments": "array",

}

**Explanation**:

* **policy\_id**: Unique identifier for each policy.
* **policyholder\_id**: Reference to the policyholder\_id in the policyholders collection.
* **start\_date**: When the policy starts, stored as a date.
* **end\_date**: When the policy ends, stored as a date.
* **premium**: Amount to be paid for the policy, stored as a number.
* **coverage:** Amount covered by the policy, stored as number.
* **payments:** Array containing payment ID, amount paid and date.

#### 3. Claims Collection

This collection will store details about the claims made against policies.

{

"claim\_id": "string",

"policy\_id": "string",  
 "policyholder\_id": "string",

"date\_of\_claim": "date",

"claim\_amount": "number",

"status": "string",  
 "reason\_of\_claim": "string",

}

**Explanation**:

* **claim\_id**: Unique identifier for the claim.
* **policy\_id**: Link to the policy this claim is against (reference to policy\_id in the policies collection).
* **date\_of\_claim**: The date when the claim was made.
* **claim\_amount**: The amount claimed.
* **status**: The current status of the claim (e.g., "Pending", "Approved", "Denied").
* **reason\_of\_claim:** Reason submitted for the claim

### Indexing Strategy

For optimal performance, especially on read operations, I considerd indexing the following fields:

* policyholder\_id in both the policyholders and policies collections for quick lookups.
* policy\_id in both the policies and claims collections to efficiently join data and process claims.
* claim\_id in the claims collection if direct access to claim records is frequently required.

### Integrating with Node.js

To integrate this schema with your Node.js application, I used the Mongoose library or the native MongoDB driver for Node.js. Mongoose, in particular, allows you to define models with schemas that can enforce the structure and validation we've described here.

### Creating Mongoose Models

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const claimSchema = new Schema({

claim\_id: { type: String, required: true, unique: true, index: true },

policy\_id: { type: String, required: true, index: true },

date\_of\_claim: { type: Date, required: true },

claim\_amount: { type: Number, required: true },

status: { type: String, default: "Pending"},

reason\_of\_claim: { type: String, required: true}

});

module.exports = mongoose.model("Claim", claimSchema);

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const policySchema = new Schema({

policy\_id: { type: String, required: true, unique: true, index: true },

policyholder\_id: { type: String, required: true, index: true },

start\_date: { type: Date, required: true },

end\_date: { type: Date, required: true },

coverage: { type: Number, required: true },

payments: [{

payment\_id: { type: String, required: true, unique: true, default: () => new mongoose.Types.ObjectId().toString() },

date: { type: Date, default: Date.now },

amount: { type: Number, required: true },

}],

premium: { type: Number, required: true }

});

module.exports = mongoose.model("Policy", policySchema);

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const policyholderSchema = new Schema({

policyholder\_id: { type: String, required: true, unique: true, index: true },

name: { type: String, required: true },

date\_of\_birth: { type: Date, required: true },

address: { type: String, required: true },

phone: { type: String, required: true }

});

module.exports = mongoose.model("Policyholder", policyholderSchema);

## Application Modification

To refactor the Node.js application to interact with a MongoDB database using Mongoose, we'll make changes to use the database for all data operations, replacing the in-memory storage used previously. This involves setting up Mongoose models, connecting to MongoDB, and modifying API endpoints to perform CRUD operations via Mongoose.

### Step 1: Connect to MongoDB

Add a connection setup to MongoDB in your application, usually at the start of your main server file (app.js).

const mongoose = require("mongoose")

const dotenv = require("dotenv")

dotenv.config();

mongoose

.connect(process.env.MONGODB\_URL)

.then(() => console.log("Connected to MongoDB"))

.catch((err) => console.log('Could not connect to MongoDB', err))

### Step 2: Update API Endpoints

Modified the API endpoints to use these models for database operations instead of manipulating in-memory objects.

// Create a new policyholder

app.post('/policyholders', async (req, res) => {

try {

const policyholder = new Policyholder(req.body);

await policyholder.save();

res.status(201).send(policyholder);

} catch (error) {

res.status(400).send(error.message);

}

});

// Retrieve a specific policyholder

app.get('/policyholders/:policyholder\_id', async (req, res) => {

try {

const policyholder = await Policyholder.findOne({ policyholder\_id: req.params.policyholder\_id });

if (!policyholder) return res.status(404).send('Policyholder not found');

res.send(policyholder);

} catch (error) {

res.status(400).send(error.message);

}

});

// Update a policyholder

app.put('/policyholders/:policyholder\_id', async (req, res) => {

try {

const policyholder = await Policyholder.findOneAndUpdate({ policyholder\_id: req.params.policyholder\_id }, req.body, { new: true });

if (!policyholder) return res.status(404).send('Policyholder not found');

res.send(policyholder);

} catch (error) {

res.status(400).send(error.message);

}

});

// Delete a policyholder

app.delete('/policyholders/:policyholder\_id', async (req, res) => {

try {

const policyholder = await Policyholder.findOneAndDelete({ policyholder\_id: req.params.policyholder\_id });

if (!policyholder) return res.status(404).send('Policyholder not found');

res.send({ message: 'Policyholder deleted successfully' });

} catch (error) {

res.status(400).send(error.message);

}

});

// Create a new policy

app.post('/policies', async (req, res) => {

try {

const policy = new Policy(req.body);

await policy.save();

res.status(201).send(policy);

} catch (error) {

res.status(400).send(error.message);

}

});

// Retrieve a specific policy

app.get('/policies/:policy\_id', async (req, res) => {

try {

const policy = await Policy.findOne({ policy\_id: req.params.policy\_id });

if (!policy) return res.status(404).send('Policy not found');

res.send(policy);

} catch (error) {

res.status(400).send(error.message);

}

});

// Update a policy

app.put('/policies/:policy\_id', async (req, res) => {

try {

const policy = await Policy.findOneAndUpdate({ policy\_id: req.params.policy\_id }, req.body, { new: true });

if (!policy) return res.status(404).send('Policy not found');

res.send(policy);

} catch (error) {

res.status(400).send(error.message);

}

});

// Delete a policy

app.delete('/policies/:policy\_id', async (req, res) => {

try {

const policy = await Policy.findOneAndDelete({ policy\_id: req.params.policy\_id });

if (!policy) return res.status(404).send('Policy not found');

res.send({ message: 'Policy deleted successfully' });

} catch (error) {

res.status(400).send(error.message);

}

});

// Create a new claim

app.post('/claims', async (req, res) => {

try {

const claim = new Claim(req.body);

await claim.save();

res.status(201).send(claim);

} catch (error) {

res.status(400).send(error.message);

}

});

// Retrieve a specific claim

app.get('/claims/:claim\_id', async (req, res) => {

try {

const claim = await Claim.findOne({ claim\_id: req.params.claim\_id });

if (!claim) return res.status(404).send('Claim not found');

res.send(claim);

} catch (error) {

res.status(400).send(error.message);

}

});

// Update a claim

app.put('/claims/:claim\_id', async (req, res) => {

try {

const claim = await Claim.findOneAndUpdate({ claim\_id: req.params.claim\_id }, req.body, { new: true });

if (!claim) return res.status(404).send('Claim not found');

res.send(claim);

} catch (error) {

res.status(400).send(error.message);

}

});

// Delete a claim

app.delete('/claims/:claim\_id', async (req, res) => {

try {

const claim = await Claim.findOneAndDelete({ claim\_id: req.params.claim\_id });

if (!claim) return res.status(404).send('Claim not found');

res.send({ message: 'Claim deleted successfully' });

} catch (error) {

res.status(400).send(error.message);

}

});

## Error Handling and Data Validation

To implement error handling mechanisms, and ensure data validation within the Node.js application, I used Mongoose's built-in validation features along with custom validation logic and express-validator for validating the incoming data.

#### Updated Schemas with Validation

First, I updated the Mongoose schemas to include validation rules.

#### Policy Schema

const mongoose = require('mongoose');

const Policyholder = require('./Policyholder');

const Schema = mongoose.Schema;

const policySchema = new Schema({

policy\_id: { type: String, required: true, unique: true, index: true },

policyholder\_id: {

type: String, required: true, index: true, validate: {

validator: async function (value) {

const policyholder = await Policyholder.findOne({ policyholder\_id: value });

return policyholder != null;

}

}

},

start\_date: { type: Date, required: true, validate:{

validator: function(value){

return this.end\_date > value;

},

message: 'Start date cannot be after end date'

}},

end\_date: { type: Date, required: true, validate: {

validator: function(value){

return this.start\_date < value;

},

message: 'End date must be after start date'

}},

premium: { type: Number, required: true }

});

module.exports = mongoose.model("Policy", policySchema);

#### **Claim Schema**

const mongoose = require('mongoose');

const Policy = require('./Policy');

const Schema = mongoose.Schema;

const claimSchema = new Schema({

claim\_id: {

type: String, required: true, unique: true, index: true

},

policy\_id: {

type: String, required: true, index: true, validate: {

validator: async function (value) {

const policy = await Policy.findOne({ policy\_id: value });

return policy != null;

},

message: 'Policy does not exist'

}

},

date\_of\_claim: { type: Date, required: true, validate:{

validator: async function (value) {

const policy = await Policy.findOne({ policy\_id: this.policy\_id });

return value >= policy.policy\_start\_date && value <= policy.policy\_end\_date;

},

message: 'Claim date must be after start date and before end date'

}},

claim\_amount: {

type: Number, required: true, validate: {

validator: async function (value) {

const policy = await Policy.findOne({ policy\_id: this.policy\_id });

return value <= policy.premium;

},

message: 'Claim amount must not exceed policy premium'

}

},

status: { type: String, required: true }

});

module.exports = mongoose.model("Claim", claimSchema);

### Updated API Endpoints with Validation and Error Handling

#### Policyholder

// Create a new policyholder

app.post('/policyholders', [

body('policyholder\_id').notEmpty().withMessage('Policyholder ID is required'),

body('name').notEmpty().withMessage('Name is required'),

body('date\_of\_birth').isDate().withMessage('Date of Birth must be a valid date'),

body('address').notEmpty().withMessage('Address is required'),

body('phone').notEmpty().withMessage('Phone number is required')

], async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}

try {

const policyholder = new Policyholder(req.body);

await policyholder.save();

res.status(201).send(policyholder);

} catch (error) {

res.status(400).send(error.message);

}

});

// Retrieve a specific policyholder

app.get('/policyholders/:policyholder\_id', async (req, res) => {

try {

const policyholder = await Policyholder.findOne({ policyholder\_id: req.params.policyholder\_id });

if (!policyholder) return res.status(404).send('Policyholder not found');

res.send(policyholder);

} catch (error) {

res.status(400).send(error.message);

}

});

// Update a policyholder

app.put('/policyholders/:policyholder\_id', async (req, res) => {

try {

const policyholder = await Policyholder.findOneAndUpdate({ policyholder\_id: req.params.policyholder\_id }, req.body, { new: true, runValidators: true });

if (!policyholder) return res.status(404).send('Policyholder not found');

res.send(policyholder);

} catch (error) {

res.status(400).send(error.message);

}

});

// Delete a policyholder

app.delete('/policyholders/:policyholder\_id', async (req, res) => {

try {

const policyholder = await Policyholder.findOneAndDelete({ policyholder\_id: req.params.policyholder\_id });

if (!policyholder) return res.status(404).send('Policyholder not found');

res.send({ message: 'Policyholder deleted successfully' });

} catch (error) {

res.status(400).send(error.message);

}

});

#### Policy

// Create a new policy

app.post('/policies', [

body('policy\_id').notEmpty().withMessage('Policy ID is required'),

body('policyholder\_id').notEmpty().withMessage('Policyholder ID is required'),

body('start\_date').isDate().withMessage('Start Date must be a valid date'),

body('end\_date').isDate().withMessage('End Date must be a valid date'),

body('premium').isNumeric().withMessage('Premium must be a number')

], async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}

try {

const policy = new Policy(req.body);

await policy.save();

res.status(201).send(policy);

} catch (error) {

res.status(400).send(error.message);

}

});

// Retrieve a specific policy

app.get('/policies/:policy\_id', async (req, res) => {

try {

const policy = await Policy.findOne({ policy\_id: req.params.policy\_id });

if (!policy) return res.status(404).send('Policy not found');

res.send(policy);

} catch (error) {

res.status(400).send(error.message);

}

});

// Update a policy

app.put('/policies/:policy\_id', async (req, res) => {

try {

const policy = await Policy.findOneAndUpdate({ policy\_id: req.params.policy\_id }, req.body, { new: true, runValidators: true });

if (!policy) return res.status(404).send('Policy not found');

res.send(policy);

} catch (error) {

res.status(400).send(error.message);

}

});

// Delete a policy

app.delete('/policies/:policy\_id', async (req, res) => {

try {

const policy = await Policy.findOneAndDelete({ policy\_id: req.params.policy\_id });

if (!policy) return res.status(404).send('Policy not found');

res.send({ message: 'Policy deleted successfully' });

} catch (error) {

res.status(400).send(error.message);

}

});

#### Claim

// Create a new claim

app.post('/claims', [

body('claim\_id').notEmpty().withMessage('Claim ID is required'),

body('policy\_id').notEmpty().withMessage('Policy ID is required'),

body('date\_of\_claim').isDate().withMessage('Date of Claim must be a valid date'),

body('claim\_amount').isNumeric().withMessage('Claim Amount must be a number'),

body('status').notEmpty().withMessage('Status is required')

], async (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}

try {

const claim = new Claim(req.body);

await claim.save();

res.status(201).send(claim);

} catch (error) {

res.status(400).send(error.message);

}

});

// Retrieve a specific claim

app.get('/claims/:claim\_id', async (req, res) => {

try {

const claim = await Claim.findOne({ claim\_id: req.params.claim\_id });

if (!claim) return res.status(404).send('Claim not found');

res.send(claim);

} catch (error) {

res.status(400).send(error.message);

}

});

// Update a claim

app.put('/claims/:claim\_id', async (req, res) => {

try {

const claim = await Claim.findOneAndUpdate({ claim\_id: req.params.claim\_id }, req.body, { new: true, runValidators: true });

if (!claim) return res.status(404).send('Claim not found');

res.send(claim);

} catch (error) {

res.status(400).send(error.message);

}

});

// Delete a claim

app.delete('/claims/:claim\_id', async (req, res) => {

try {

const claim = await Claim.findOneAndDelete({ claim\_id: req.params.claim\_id });

if (!claim) return res.status(404).send('Claim not found');

res.send({ message: 'Claim deleted successfully' });

} catch (error) {

res.status(400).send(error.message);

}

});

## Pull Request (PR) for Merging

I have created a pull request for merging the `feature` branch with `main` branch

<https://github.com/devisri-charan/bootcamp/pull/3>

## Deploy on cloud

Deployed the API using render cloud service.   
<https://bootcamp-y8br.onrender.com>