

Lab 15: *MongoDB*

Dataend

CRUD + JWT

MongoDB, Mongoose, CRUD, JWT

IMPORTANT NOTE: Install Mongo Shell for Backend

Table of Content: MongoDB App

| # of Parts | Topic | Page |
|--------------------------|--|------|
| Introduction | Lab Introduction <i>MongoDB, MQL, Atlas</i> | 3 |
| Basic Setup | | |
| Goal 0 | Launch MongoDB Server - (mongod) | 6 |
| Goal 1 | Initialize Project with package.json | 7 |
| Goal 2 | Express Static Server | 8 |
| CRUD Operations | | |
| Goal 3 | Connect to MongoDB Server | 10 |
| Goal 4 | CREATE - User data | 12 |
| Goal 5 | READ All - User data | 15 |
| Goal 6 | READ One - User data | 17 |
| Goal 7 | UPDATE - User data | 19 |
| Goal 8 | DELETE - User data | 21 |
| JWT Authorization | | |
| Goal 9 | JWT Login | 23 |
| Goal 10 | Token Access | 25 |
| End | Concluding Notes <i>Summary and Submission notes</i> | 27 |

Lab Introduction

Prerequisites

Install Node (npm/node) and Mongo (mongod).

Motivation

Learn to connect your express app to a mongodb database.

Goal

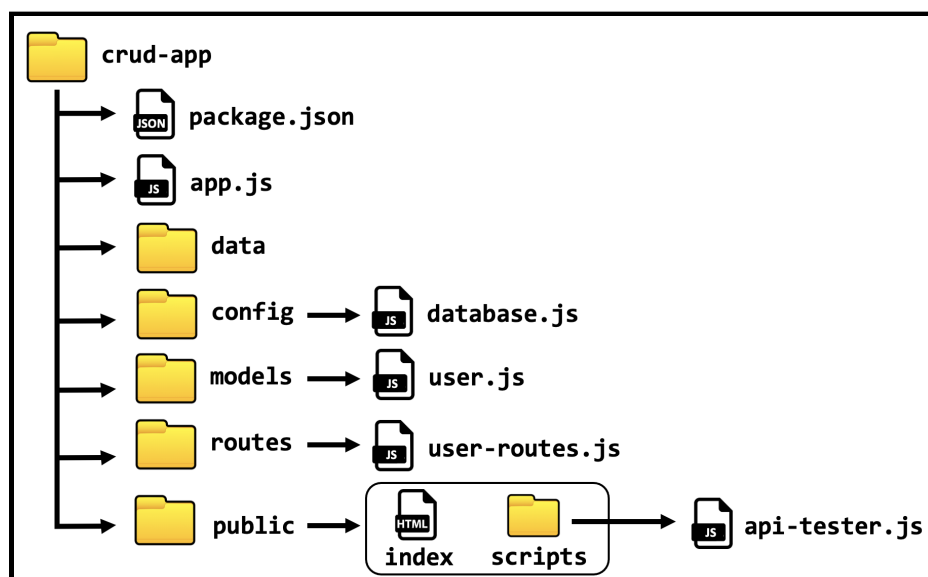
Deploy a Mongo Database Server on Localhost and connect to it with an Express App.

Learning Objectives

- MongoDB Server (mongod)
- Mongoose client
- Mongoose Schemas to define ORM
- CRUD operations in MongoDB (Create, Read, Update, Delete)
- Authentication with JSON Web Tokens

Server-side Architecture:

Start this project by making a project folder where all your assets & scripts will be organized. Create all necessary files and folders as illustrated below.



Concepts

User Authentication - JSON Web Tokens (JWT)

JWT is a novel form of authentication in that it maintains statelessness in the server. Unlike Session-based approaches, where the server maintains a list of client ids. With JWT, the server issues the client a token which then must include it in its request for any private data.

- **Headers:** identifies which algorithm is used to generate the signature.
 - **Payloads:** contains the claims of the token.
 - **Signature:** The signature is created from the encoded header, encoded payload, a secret (or private key, read further) and a cryptographic algorithm. All these four components allow the creation of a signature.
 - **Clientend:** it's the client's responsibility to store and maintain its auth token and send it in its requests to the server.
 - **Protecting Routes:** A route or endpoint that a request cannot access if they do not have the proper authentication. Examples may include not accessing a user-profile page unless the Request comes from that particular user.
-

MongoDB Server

MongoDB is an open source NoSQL database management program. NoSQL is used as an alternative to traditional relational databases. NoSQL databases are quite useful for working with large sets of distributed data. MongoDB is a tool that can manage document-oriented information, store or retrieve information.

- **mongod:** the primary mongo daemon process for the MongoDB system. It handles data requests, manages data access, and performs background management operations..
-

Mongo Client

The Mongo Client is the bindings in JavaScript that allows NodeJS apps to connect to a MongoDB server.

- **Mongoose:** a Node library for interfacing with MongoDB databases. Mongoose is an ODM (Object Document Mapping) - it acts as an interface to MongoDB by letting you define Models. In this instance, a Model is like a blueprint of a Mongo Index. It maps the properties of the Documents in that Index and lets you interact with them in meaningful ways.
- **Schemas:** a schema represents the structure of a particular document, either completely or just a portion of the document. It's a way to express expected properties and values as well as constraints and indexes. A model defines a programming interface for interacting with the database (read, insert, update, etc).

CRUD Operations

CRUD Meaning: CRUD is an acronym that comes from the world of computer programming and refers to the four functions that are considered necessary to implement a persistent storage application: create, read, update and delete.

- **Create:** Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.
 - **Read:** Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:
 - **Update:** Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection
 - **Delete:** Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection.
-

Goal 0: MongoDB Server

'Approach' → Plan phase

Launch MongoDB Server app on Localhost

'Apply' → Do phase

→ Backend - MongoDB

Step 1: Download and install MongoDB on your machine.

See NPM for more info: <https://www.mongodb.com/docs/manual/installation/>

Step 2: (bash) `mongod` → Launch the MongoDB server app from the CLI terminal

```
mongod --dbpath data/ --port 27017 --bind_ip '127.0.0.1'
```

| | |
|------------------------|--|
| <code>mongod</code> | The primary daemon process for the MongoDB system. It handles data requests, manages data format, and performs background management operations. |
| <code>--dbpath</code> | Specify a directory for the mongod instance to store its data |
| <code>--port</code> | Specifies a port for the mongod to listen for client connections. The default port is 27017 |
| <code>--bind_ip</code> | The IP address that the mongod process will bind to and listen for connections. |

IMPORTANT: open terminal in project's root folder, there must also be an empty `data/` subfolder

'Assess' → Test phase

The server should publish live log data into the terminal, as it waits for incoming connections

```
dataDirectory : data/diagnostic.data } }
{"t":{"$date":"2022-05-02T17:31:21.410-05:00"},"s":"I", "c":"NETWORK", "id":23015,
"ctx":"listener","msg":"Listening on","attr":{"address":"/tmp/mongodb-27017.sock"}}
{"t":{"$date":"2022-05-02T17:31:21.410-05:00"},"s":"I", "c":"NETWORK", "id":23015,
"ctx":"listener","msg":"Listening on","attr":{"address":"127.0.0.1"}}
{"t":{"$date":"2022-05-02T17:31:21.410-05:00"},"s":"I", "c":"NETWORK", "id":23016,
"ctx":"listener","msg":"Waiting for connections","attr":{"port":27017,"ssl":"off"}}
```

IMPORTANT: the `mongod` daemon process should remain running throughout this entire lab.

Goal 1: Configuration file (*package.json*)

'Approach' → Plan phase

*The configuration file (*package.json*) for managing the backend of this project*

'Apply' → Do phase

→ Backend - NodeJS

Step 1 (json): *package.json* → app metadata & dependencies

Create a *package.json* file with the configuration information for this app. This file is used for managing the project's dependencies, scripts, version, etc.

package.json

```
{
  "name": "crud-app",
  "version": "1.0.0",
  "description": "App that connects mongoose client to mongodb server for CRUD ops plus JWT auth",
  "author": "my_name",
  "main": "app.js",
  "scripts": {
    "start": "npm install && node app"
  },
  "dependencies": {
    "express": "*"
  }
}
```

Step 2 (bash) *npm* → install dependencies

NPM uses the *package.json* file to fetch and download the app's dependencies with the install command.

```
npm install
```

'Approve' → Test phase

All the app's dependencies should be downloaded into a `node_modules` folder.

Goal 2: Express Static Server

[Approach] → Plan phase

Create an Express app to serve browser HTML files and listen for HTTP Request

[Apply] → Do phase

→ Backend - NodeJS

Step 1: (JS): app.js → Create a Static Server with Express

Import express, create an app, use public/ for static files, define a route for index then listen for request.

app.js

```
const express = require ('express');           //import express module
const app = express();                         //create express app
const port = 3000;                             //define server port

app.use( express.static('public') );           //use public/ for static files
app.get("/", getIndex);                       //GET endpoint for index.html

app.listen(port, () => console.log("server is running port ", port) ); //app listens for request

function getIndex(request, response){          //callback fxn for request
  response.sendFile('./public/index.html', { root: __dirname }); //sends index.html file
}
```

← Frontend - Browser

Step 1: (HTML) index.html → Create HTML file in public/ folder

HTML document to load into the browser for all client-end code in the app i.e. test REST endpoints.

public/index.html

```
<html>
  <head>
    <title> CRUD API Tester</title>
  </head>
  <body>
    <!-- CRUD API TESTER -->
    <h1>CRUD API Tester</h1>
    <hr>
    <!-- CREATE -->
    <!-- READ ALL -->
    <!-- READ ONE -->
    <!-- UPDATE -->
    <!-- DELETE -->

    <!-- JWT AUTH TESTER -->
    <!-- JWT LOGIN -->
    <!-- TOKEN AUTH -->
    <script src="scripts/api-tester.js"></script>
  </body>
</html>
```

[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

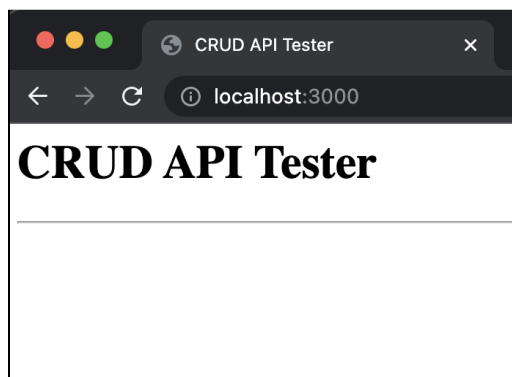
SERVER

The terminal running the web server should display the message:

```
server is running port 3000
```

CLIENT

Open your browser to <http://localhost:3000>



Goal 3: Connect to Database

'Approach' → Plan phase

Use Mongoose client in Express app to connect to the MongoDB Server

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (bash) *npm* → install mongoose

Mongoose is a library built on top of the MongoDB Client that also offers Schema support as JS Objects.

```
npm install mongoose
```

Step 2: (JS) *config/database.js* → Create a **Database Configuration** module

This module contains the private configuration data for accessing the mongodb server. It includes the url, port, and name of the database. *Note: Don't push this module into public repos.*

config/database.js

```
module.exports = {  
  "database": "mongodb://127.0.0.1:27017/my_app_data"  
}
```

Step 3 (JS): *app.js* → Import Mongoose module & connect to MongoDB server

Import mongoose & the database config module to establish a connection to the mongodb server.

Learn more about mongoose and its config options: <https://mongoosejs.com/>

app.js

```
const express = require('express');           //import express module  
const app = express();                        //create express app  
const port = 3000;                           //define server port  
const { database } = require('./config/database'); //import db config settings  
const mongoose = require('mongoose');        //import mongoose  
  
const mongoose_config = {useNewUrlParser: true, useUnifiedTopology: true}; //connection configs  
const connection = mongoose.connect(database, mongoose_config); //connect to mongo server
```

```
if (connection){                                     //log connection result
  console.log('database connected');
}
else{
  console.log('database connection error')
}

app.use( express.static('public') );                 //public/ for static files
app.get("/", getIndex);                             //GET endpoint - index.html

app.listen(port, () => console.log("server is running port ", port) ); //app listens for request

function getIndex(request, response){               //callback fxn for request
  response.sendFile('./public/index.html', { root: __dirname }); //sends index.html file
}
```

'Approve' → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

SERVER

The terminal running the web server should display a message that the database connected successfully:

```
database connected
server is running port 3000
```

Goal 4: Create - User Data

'Approach' → Plan phase

Define a REST endpoint in app router to 'Create' a User document in Database

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (JS): *models/user.js* → Schema for User data

Make a schema that represents the user's fields and datatypes in the database.

models/user.js

```
const mongoose = require('mongoose');           //import mongoose module
const Schema = mongoose.Schema                  //import the Schema class

const userSchema = new Schema({                 //make a new instance of Schema
  email: String,                                //define collection's fields & types
  password: String
});

const User = module.exports = mongoose.model('User', userSchema); //export the schema as a module
```

Step 2: (JS) *api/user-routes.js* → User-Routes module

user-routes.js manages all user-related requests in API. Let's define an endpoint to create a user in DB.

api/user-routes.js

```
const express = require('express');               //import express module
const router = express.Router();                  //create a router object
const User = require('../models/user');           //import User model

router.post("/register", registerPOST);            //POST endpoint for CREATE

async function registerPOST(request, response){    //callback fxn for endpoint
  const {email, password} = request.body;          //user fields from req body
  const newUser = new User({ email: email, password: password }); //create new user instance
  const document = await newUser.save()            //schema's save() into db
  const json = {state: true, msg: "data inserted", document: document } //results as json
  response.json(json);                             //send json with response
}

module.exports = router;                          //export router module
```

Step 3: (bash) npm → body-parser module

body-parser module allows access into a request's body data. Use npm to quickly install from a terminal.

```
npm install body-parser
```

Step 4 (JS): app.js → Import and use Body-Parser & User-Routes Modules

import the body-parser & user-routes modules and have the express app use them as middleware.

app.js

```
const express = require ('express');           //import express module
const app = express();                         //create express app
const port = 3000;                            //define server port
const { database } = require ('./config/database'); //import db config settings
const mongoose = require ('mongoose');        //import mongoose

const mongoose_config = {useNewUrlParser: true, useUnifiedTopology: true}; //connection configs
const connection = mongoose.connect(database, mongoose_config); //connect to mongo server

if (connection){                               //log connection result
  console.log('database connected');
}
else{
  console.log('database connection error')
}

const bodyParser = require('body-parser');     //import body-parser module
const userRoutes = require('./api/user-routes'); //import user-routes module

app.use(bodyParser.json());                    //use body-parser for json
app.use('/', userRoutes);                     //use router on root path
app.use( express.static('public') );          //public/ for static files
app.get("/", getIndex);                       //GET endpoint - index.html

app.listen(port, () => console.log("server is running port ", port) ); //app listens for request

function getIndex(request, response){         //callback fxn for request
  response.sendFile('./public/index.html', { root: __dirname }); //sends index.html file
}
```

← Frontend - Browser**Step 1: (HTML) index.html → HTML inputs/button to test CREATE endpoint**

Find the ← create → comment in the <body> and add the following HTML elements under it.

public/index.html → <body>

```
<!-- CREATE -->
<button onclick=testCreate() > CREATE </button>
<input type="email" id="createEmail" placeholder="email">
<input type="text" id="createPassword" placeholder="password">
<hr>
```

Step 2: (JS) *public/scripts/api-tester.js* → testCreate function

Make api-tester.js file in the public/scripts folder. Implement the button's testCreate function from HTML

public/scripts/api-tester.js

```
//sets up a HTTP POST Request to send to app's CREATE endpoint and displays result
async function testCreate(){
  const config = new Object();
  config.method = "POST";
  config.headers = { 'Accept': 'application/json', 'Content-Type': 'application/json' };
  config.body = JSON.stringify({'email': createEmail.value, 'password': createPassword.value});
  const response = await fetch("http://localhost:3000/register", config);
  const data = await response.json()
  document.body.innerHTML += `<p>${JSON.stringify(data)}</p>`
}
```

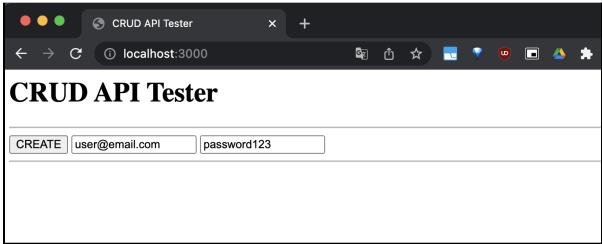
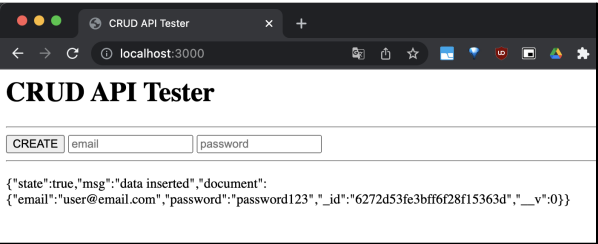
[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

CLIENT

Open your browser to <http://localhost:3000>

| Input an email + password, then press 'CREATE' | The Response JSON should display from server |
|---|--|
|  |  |

Goal 5: Read All - User Data

'Approach' → Plan phase

Define a REST endpoint in app router to 'Read All' User documents in Database

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (JS): *api/user-routes.js* → READ - ALL USERS endpoint

Define a route and callback fxn for finding all User documents from the database. Mongo find() method retrieves all user documents from the database.

api/user-routes.js

```
router.get("/getuser", getuserGET); //GET endpoint for READ ALL

async function getuserGET(request, response) { //callback fxn for READ ALL
  const documents = await User.find(); //find() gets all Users in DB
  const json = {status:200, msg:'Users data fetched', data: documents}; //results as json
  response.json(json); //send json with response
}
```

← Frontend - Browser

Step 1: (HTML) *index.html* → HTML button to test READ ALL endpoint

Find the ← read all → comment in the <body> and add the following HTML elements under it.

public/index.html → <body>

```
<!-- READ ALL -->
<button onclick=testReadAll() > READ ALL</button>
<hr>
```

Step 2: (JS) *public/scripts/api-tester.js* → testReadAll function

Implement the HTML button's testReadAll function. Sends a fetch GET request to READ ALL endpoint.

public/scripts/api-tester.js

```
async function testReadAll(){
  const config = new Object();
  config.method = "GET";
  const response = await fetch("http://localhost:3000/getuser", config);
  const data = await response.json()
  document.body.innerHTML += `

`${JSON.stringify(data)}`</p>`
}


```

[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

CLIENT

Open your browser to <http://localhost:3000>

| Press the READ ALL button | The Response JSON should include all Users from DB |
|--|---|
| <div> CRUD API Tester </div> <div> <input type="button" value="CREATE"/> <input type="text" value="email"/> <input type="password" value="password"/> </div> <div> <input type="button" value="READ ALL"/> </div> | <div> CRUD API Tester </div> <div> <input type="button" value="CREATE"/> <input type="text" value="email"/> <input type="password" value="password"/> </div> <div> <input type="button" value="READ ALL"/> </div> <div> <pre>{ "status": 200, "msg": "Users data fetched", "data": [{ "_id": "6272d53fe3bff6f28f15363d", "email": "user@email.com", "password": "password123", "__v": 0 }, { "_id": "6272d7b5e3bff6f28f15363f", "__v": 0 }] }</pre> </div> |

Goal 6: Read One - User Data

'Approach' → Plan phase

Define a REST endpoint in app router to 'Read One' User document in Database

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (JS): *api/user-routes.js* → READ - ONE USER endpoint

Define a route and callback fxn for finding a single User document from the database. Mongo `findById()` method retrieves one user document from the database using an id.

api/user-routes.js

```
router.get("/get/:id", userGET); //GET endpoint for READ ONE

async function userGET(request, response){ //callback fxn for READ ONE
  const documents = await User.findById(request.params.id) //findById() gets a User in DB
  if (documents) { //user exist then send as json
    response.status(200).json(documents);
  }
  else { //otherwise send 404 status
    response.status(404).json({ msg: 'data not found' });
  }
}
```

← Frontend - Browser

Step 1: (HTML) *index.html* → HTML input/button to test READ ONE endpoint

Find the ← read one → comment in the <body> and add the following HTML elements under it.

public/index.html → <body>

```
<!-- READ ONE -->
<button onclick=testRead() > READ ID</button>
<input type="text" id="readId" placeholder="id">
<hr>
```

Step 2: (JS) *public/scripts/api-tester.js* → testRead function

Implement the HTML button's testRead function. Sends a fetch GET request to the READ ONE endpoint.

public/scripts/api-tester.js

```

async function testRead(){
  const config = new Object();
  config.method = "GET";
  const response = await fetch(`http://localhost:3000/get/${readId.value}`, config);
  const data = await response.json()
  document.body.innerHTML += `

${JSON.stringify(data)}</p>`
}


```

[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

CLIENT

Open your browser to <http://localhost:3000>

| A valid <code>_id</code> returns that user document from the DB | An invalid <code>_id</code> returns a 'data not found' message |
|---|--|
| <p>CRUD API Tester</p> <p>CREATE <input type="text" value="email"/> <input type="text" value="password"/></p> <p>READ ALL</p> <p>READ ID <input type="text" value="id"/></p> <p><code>{"_id":"6272d53fe3bff6f28f15363d","email":"user@email.com","password":"password123","_v":0}</code></p> | <p>CRUD API Tester</p> <p>CREATE <input type="text" value="email"/> <input type="text" value="password"/></p> <p>READ ALL</p> <p>READ ID <input type="text" value="id"/></p> <p><code>{"msg":"data not found"}</code></p> |

Goal 7: Update - User Data

'Approach' → Plan phase

Define a REST endpoint in app router to 'Update' a User document in Database

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (JS): *api/user-routes.js* → UPDATE - USER endpoint

Define a route and callback fxn for updating a single User document in the database. Mongo `updateOne()` or `updateMany()` methods update data in a user document from the database using an id.

api/user-routes.js

```
router.put('/update/:id', updateuserPUT); //PUT endpoint for UPDATE

async function updateuserPUT (request,response) { //callback fxn for UPDATE
  const user = { _id: request.params.id }; //id to find a user
  const data = { email: request.body.email, password: request.body.password } //data to update in user
  const document = await User.updateOne(user,data); //updateOne() syncs to DB
  if(!document){ //no user, 404 status
    return response.status(404).json({ msg: 'data not found' });
  }
  return response.status(200).json(document); //otherwise send as json
}
```

← Frontend - Browser

Step 1: (HTML) *index.html* → HTML input/button to test UPDATE endpoint

Find the ← `update` → comment in the `<body>` and add the following HTML elements under it.

public/index.html → `<body>`

```
<!-- UPDATE -->
<button onclick=testUpdate() > UPDATE </button>
<input type="text" id="updateId" placeholder="id">
<input type="email" id="updateEmail" placeholder="email">
<input type="text" id="updatePassword" placeholder="password">
<hr>
```

Step 2: (JS) `public/scripts/api-tester.js` → `testUpdate` function

Implement the HTML button's `testUpdate` function. Sends a fetch PUT request to UPDATE endpoint.

`public/scripts/api-tester.js`

```

async function testUpdate(){
  const config = new Object();
  config.method = "PUT";
  config.headers = { 'Accept': 'application/json', 'Content-Type': 'application/json' };
  config.body = JSON.stringify({ 'email': updateEmail.value, 'password': updatePassword.value });
  const response = await fetch(`http://localhost:3000/update/${updateId.value}`, config);
  const data = await response.json();
  document.body.innerHTML += `<p>${JSON.stringify(data)}</p>`
}

```

[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

CLIENT

Open your browser to <http://localhost:3000>

| | |
|--|--|
| <p>A valid <code>_id</code> returns an acknowledgment that the user document was updated. Reading that user, can confirm it.</p> | <p>An invalid <code>_id</code> returns an acknowledgement but showing no changes occurred</p> |
| <p>CRUD API Tester</p> <p>CREATE <input type="text" value="email"/> <input type="text" value="password"/></p> <p>READ ALL</p> <p>READ ID <input type="text" value="id"/></p> <p>UPDATE <input type="text" value="id"/> <input type="text" value="email"/> <input type="text" value="password"/></p> <pre> {"_id":"6272d53fe3bff6f28f15363d","email":"user@email.com","password":"password123","__v":0} {"acknowledged":true,"modifiedCount":1,"upsertedId":null,"upsertedCount":0,"matchedCount":1} {"_id":"6272d53fe3bff6f28f15363d","email":"updated@email.com","password":"qwerty","__v":0} </pre> | <p>CRUD API Tester</p> <p>CREATE <input type="text" value="email"/> <input type="text" value="password"/></p> <p>READ ALL</p> <p>READ ID <input type="text" value="id"/></p> <p>UPDATE <input type="text" value="id"/> <input type="text" value="email"/> <input type="text" value="password"/></p> <pre> {"acknowledged":true,"modifiedCount":0,"upsertedId":null,"upsertedCount":0,"matchedCount":0} </pre> |

IMPORTANT: This is simplest & brittle implementation, if user inputs invalid id, it can crash the server. In productions you must catch exceptions to prevent the system from crashing.

Goal 8: Delete - User Data

'Approach' → Plan phase

Define a REST endpoint in app router to 'Delete' a User document from Database

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (JS): *api/user-routes.js* → DELETE - USER endpoint

Define a route and callback fxn for deleting a single User document in the database. Mongo `deleteOne()` or `deleteMany()` methods delete documents from the database using an id.

api/user-routes.js

```
router.delete('/delete/:id', deleteuserDELETE) //DELETE endpoint for DELETE

async function deleteuserDELETE(request, response) { //callback for DELETE
  const document = await User.deleteOne({ _id: request.params.id }); //deleteOne() syncs to DB
  const json = { status: 200, msg: 'User data deleted', document: document } //results as json
  response.json(json); //send json with response
}
```

← Frontend - Browser

Step 1: (HTML) *index.html* → HTML input/button to test DELETE endpoint

Find the ← delete → comment in the <body> and add the following HTML elements under it.

public/index.html → <body>

```
<!-- DELETE -->
<button onclick=testDelete() > DELETE </button>
<input type="text" id="deleteId" placeholder="id">
<hr>
```

Step 2: (JS) *public/scripts/api-tester.js* → testDelete function

Implement the HTML button's testDelete function. Sends a fetch DELETE request to DELETE endpoint.

public/scripts/api-tester.js

```
async function testDelete(){
  const config = new Object();
  config.method = "DELETE";
  const response = await fetch(`http://localhost:3000/delete/${deleteId.value}`, config);
  const data = await response.json()
  document.body.innerHTML += `

${JSON.stringify(data)}</p>`
}


```

[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

CLIENT

Open your browser to <http://localhost:3000>

| | |
|--|--|
| <p>A valid <code>_id</code> returns an acknowledgment that the document was deleted. Reading all users, before and after can visually confirm the deletion occurred.</p> | <p>An invalid <code>_id</code> returns an acknowledgement but showing no deletes occurred</p> |
| <p>CRUD API Tester</p> <p>CREATE <input type="text" value="email"/> <input type="text" value="password"/></p> <p>READ ALL</p> <p>READ ID <input type="text" value="id"/></p> <p>UPDATE <input type="text" value="id"/> <input type="text" value="email"/> <input type="text" value="password"/></p> <p>DELETE <input type="text" value="id"/></p> <pre>{ "status": 200, "msg": "Users data fetched", "data": { { "_id": "6272eede2351c1a384eb0c49", "_v": 0 } } } { "status": 200, "msg": "User data deleted", "document": { "acknowledged": true, "deletedCount": 1 } } { "status": 200, "msg": "Users data fetched", "data": [] }</pre> | <p>CRUD API Tester</p> <p>CREATE <input type="text" value="email"/> <input type="text" value="password"/></p> <p>READ ALL</p> <p>READ ID <input type="text" value="id"/></p> <p>UPDATE <input type="text" value="id"/> <input type="text" value="email"/> <input type="text" value="password"/></p> <p>DELETE <input type="text" value="id"/></p> <pre>{ "status": 200, "msg": "User data deleted", "document": { "acknowledged": true, "deletedCount": 0 } }</pre> |

Goal 9: Json Web Token (JWT) - Login

'Approach' → Plan phase

Define a LOGIN endpoint in app router to generate a JWT token for client in response

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (bash) *npm* → install jsonwebtoken

JsonWebToken is an authentication module, where the server does not need to maintain a datastore of sessions, instead the server/client passes a token between each other in the request/response objects.

```
npm install jsonwebtoken
```

Step 2: (JS): *api/user-routes.js* → LOGIN - endpoint

Define a route and callback fxn for user login. Check the email & password are valid and generate a JWT token with the user state and a secret key used to encrypt it. Token is then embedded into the response.

api/user-routes.js

```
const jwt = require('jsonwebtoken'); //import jsonwebtoken module
router.post('/login', loginPOST); //POST endpoint for LOGIN
async function loginPOST(request,response) { //callback fxn for LOGIN
  const userData = request.body; //get data from request body
  const user = await User.findOne( {email:userData.email} ).exec(); //findOne execs to DB, gets user
  if(!user){ //wrong email, send err message
    response.status(401).json( {msg:'Invalid email'} )
  }
  else if (user.password !== userData.password){ //wrong pword, send err message
    response.status(401).json( {msg:'Invalid password'} )
  }
  else{
    const payload = {subject:user._id} //define JWT payload as user id
    const token = jwt.sign(payload,'secretKey') //hash token = payload + secret
    response.status(200).send( {token} ) //send token back to client
  }
}
```

← Frontend - Browser

Step 1: (HTML) *index.html* → HTML input/button to test LOGIN endpoint

Find the `← jwt auth tester →` comment in the `<body>` and add the following HTML elements under it.

public/index.html → **<body>**

```
<!-- JWT AUTH TESTER -->
<h1>JWT Auth Tester</h1>
<hr>
<!-- JWT LOGIN -->
<button onclick=testLogin() > LOGIN </button>
<input type="email" id="loginId" placeholder="email">
<input type="text" id="loginPassword" placeholder="password">
<hr>
```

Step 2: (JS) *public/scripts/api-tester.js* → testLogin function

Implement the HTML button's `testLogin` function. Sends a fetch POST request to the LOGIN endpoint. The browser must store the token in `sessionStorage/localStorage` and send it back with each Request

```
public/scripts/api-tester.js
```

```
async function testLogin(){
  const config = new Object();
  config.method = "POST";
  config.headers = { 'Accept': 'application/json', 'Content-Type': 'application/json' };
  config.body = JSON.stringify({ 'email': loginId.value, 'password': loginPassword.value });
  const response = await fetch("http://localhost:3000/login", config);
  const data = await response.json()
  sessionStorage.token = data.token;
  document.body.innerHTML += `

`${JSON.stringify(data)}`</p>`
}


```

[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

CLIENT: Open your browser to <http://localhost:3000>

Errors are returned for invalid email or password entries: {msg: wrong_email} / {msg: wrong_password}.
For valid entries, you should get the JWT token returned to the browser:

JWT Auth Tester

LOGIN email

{ "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9IjEudzdWZqZWNOIjo1NjMzMmY2Zml5NWQwS2lmVhYzc1NTk4NzVkIiwiaWF0IjoxNjUxNzAxNTYyZmF6rVcDP0Ay1bFdS70biX2K0ifqs1KtolajSTTrS9wm"}

Goal 10: JWT - Verify Token

'Approach' → Plan phase

Define a token verification middleware function & a protected endpoint in the app's router

'Apply' → Do phase

→ Backend - NodeJS

Step 1: (JS): *api/user-routes.js* → Verify Token - middleware

Define a middleware function that checks if a request contains an authorization in its header. If so, extract the token, and use it with the jwt module to decrypt the payload.

api/user-routes.js

```
function verifyToken( request, response, next ) {           //Middleware function
  if ( !request.headers.authorization ){                    //No authorization in header
    return response.status(401).json( {msg:'Unauthorized request'} ) //Send back with 401 status
  }
  const token = request.headers.authorization.split(' ')[1]; //split token from header
  if ( token === 'null' ){                                  //No token
    return response.status(401).json( {msg:'Unauthorized request'} ) //Send back with 401 status
  }
  const payload = jwt.verify(token,'secretKey')             //Use JWT to verify Token
  if ( !payload ) {                                         //Not valid
    return response.status(401).json( {msg:'Unauthorized request'} ) //Send back with 401 status
  }
  request.userId = payload.subject                          //Embed user id into request
  next()                                                    //Invoke next fxn in chain
}
```

Step 2: (JS): *api/user-routes.js* → SPECIAL - protected endpoint

Define a route, protect it with the verifyToken function, then handle it with a callback fxn for returning this token's user id.

routes/user-routes.js

```
router.get('/special', verifyToken, specialGET)             //GET endpoint for SPECIAL

async function specialGET(request,response){                //callback fxn for SPECIAL
  const data = {user: request.userId}                       //embed userID from payload into json
  response.json(data)                                        //send json with response
}
```

← Frontend - Browser

Step 1: (HTML) *index.html* → HTML input/button to test SPECIAL endpoint

Find the ← *token auth* → comment in the <body> and add the following HTML elements under it.

public/index.html → <body>

```
<!-- TOKEN AUTH -->
<button onclick=testAuth() > SPECIAL </button>
<hr>
```

Step 2: (JS) *public/scripts/api-tester.js* → testAuth function

Implement the HTML button's testAuth function. Sends a fetch GET request to the SPECIAL endpoint. The browser must get the token from sessionStorage/localStorage and send it in the Request header.

public/scripts/api-tester.js

```
async function testAuth(){
  const config = { };
  config.method = "GET";
  config.headers = {"Authorization": 'Bearer ' + sessionStorage.getItem('token')};
  const response = await fetch("http://localhost:3000/special", config);
  const data = await response.json();
  document.body.innerHTML += `<p>${JSON.stringify(data)}</p>`
}
```

[Approve] → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

CLIENT:

Open your browser to <http://localhost:3000>

If you're logged in and click the SPECIAL button, then your user id from DB displays, otherwise nothing happens.

JWT Auth Tester

LOGIN

SPECIAL

```
{ "user": "6272f6fb25d9beac7559875d" }
```

Conclusions

Final Comments

In this lab you implemented a Full Stack CRUD app with JWT token user authentication. This lab covered: mongodb server, mongo client, mongoose, schema, json web tokens (jwt),

Future Improvements

- Improve with fault-tolerance to catch errors on server-side to prevent crashes
- Prevent duplicate emails from being input
- Deploy into production on heroku and atlas
- Add functionality into a full stack app with a purpose

Lab Submission

Compress your project folder into a zip file and submit it to Moodle.