Storing Data in a MongoDB Database

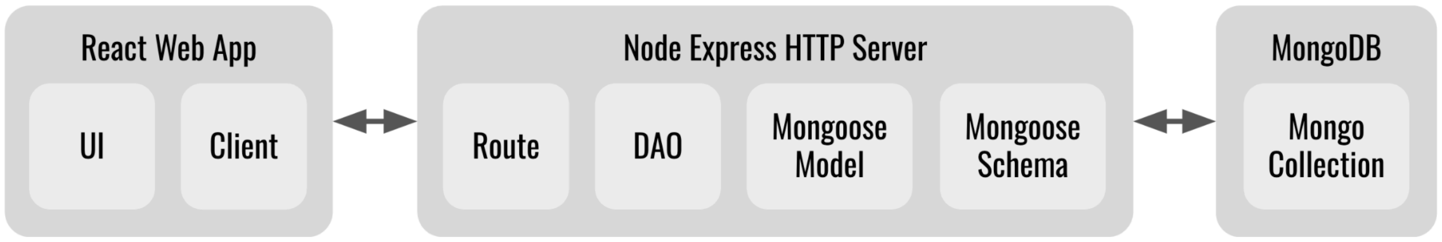
1 Introduction

There are two main categories of databases: ***relational databases*** and ***non-relational databases***. Relational databases such as ***MySQL***, ***SQL Server***, and ***Postgre***, store data in ***tables*** containing ***records*** of the same type, e.g., a table called ***courses*** would contain records representing all the ***courses***, and a ***users table*** would contain all the ***users*** of an application. Records are represented as ***rows*** in the tables where each ***column*** stores data for attributes specific to the type of the table, e.g., the rows in the ***courses*** table might have columns such as ***name***, ***description***, ***startDate***, ***endDate***, etc. Some of the columns might refer, or ***relate*** to other records in other tables such as the ***instructor*** column in the ***courses*** table might refer to, or relate to a particular row in the ***users*** table signifying that that particular user is the ***instructor*** of that particular course. Rows in one table relating to rows in another table is where ***relational*** databases get their name. The ***structured query language*** or ***SQL***, is a computer language commonly used to interact with relational databases. The ***query*** in ***SQL*** generally means to ***ask for***, or ***retrieve*** data that matches some criteria, often written as a ***boolean expression*** or ***predicate***.

More recently there has been a growing interest in representing and storing data using alternative strategies which have collectively come to be referred to as ***non relational databases***, or ***NoSQL databases***. Non relational databases such as ***MongoDB***, ***Firebase***, and ***Couchbase***, store their data in ***collections*** containing ***documents*** which are roughly analogous to ***tables*** and ***records*** in their relational counterparts. The biggest difference though is that the columns, or ***fields*** in the rows in relational databases generally can only contain ***primitive data types***, e.g., simple strings, numbers, dates, and booleans, whereas the fields in non relational documents can be arbitrarily ***complex data types***, e.g., strings, numbers, booleans, dates as well as combinations of these in complex objects containing arrays of objects of arrays, etc. The other big difference is that relational databases require the structure, or ***schema*** of the data to be explicitly described before storing any data, whereas non relational databases do not require predefined schemas. Instead, non relational databases delegate this responsibility to the applications using the database. The structure, or schema in relational databases is where ***structured query language*** gets its name.

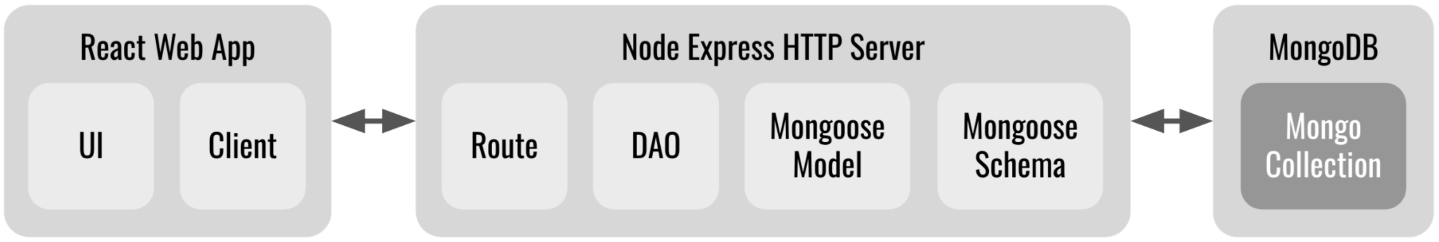
In the previous chapter we learned how to create an HTTP server with Node.js and integrated it with a React.js Web front end application to store the application state on the server. In this chapter we expand this idea to store the data to ***MongoDB***, a popular non relational database. The first section demonstrates how to download, install and use a local instance of the MongoDB database. The next section covers how to use the ***Mongoose*** library to integrate and program a MongoDB database with a Node.js server application. The final section describes how to deploy the database to ***Mongo Atlas***, a remote MongoDB database hosted as a cloud service.

The following figure illustrates the overall architecture of what we'll be building in this chapter. From right to left, we'll first create a MongoDB database called ***kanbas*** where we'll create several collections such as ***users***, ***courses***, ***modules***, ***assignments***, etc. We'll use the ***Mongoose*** library to connect to the database programmatically from a Node server. A Mongoose schema will describe the structure of the collections in the MongoDB database and a Mongoose model will implement generic ***CRUD*** operations. We'll create higher level functions in ***Data Access Objects*** (***DAO***s) that operate on the database, and expose those operations through Express routes as RESTful Web APIs. A React Web app will integrate with the RESTful API through a client that will allow the user interface to interact with the database.



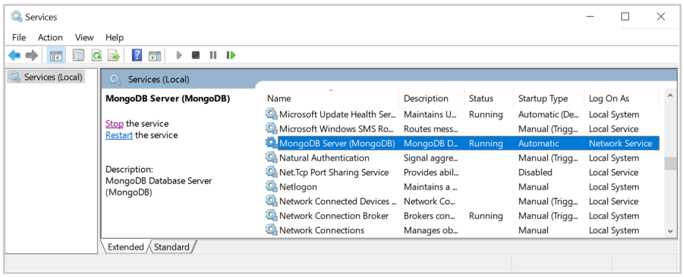
2 Working with a local MongoDB instance

MongoDB is one of an increasingly popular family of non relational databases. Data is stored in collections of documents usually formatted as JSON objects which makes it very convenient to integrate with JavaScript based frameworks such as Node.js and React.js. This section describes how to install, configure and get started using MongoDB. We'll use the image below through this chapter to highlight which part of the architecture we're discussing. As shown below, we're focusing on implementing ***Mongo Collections*** in a ***MongoDB*** database.



2.1 Installing and configuring MongoDB

To get started, download MongoDB for free selecting the latest version for your operating system, and click ***Download***. Run the installer and, if given the choice, choose to ***run the database as a service*** so that you don't have to bother having to restart the database sever every time you login or restart your computer. The ***MongoDB*** database will automatically start whenever you start your computer. On Windows, confirm the database is running by searching for ***MongoDB*** in the ***Services*** dialog. On macOS, confirm the database is running by clicking the ***MongoDB*** icon in the ***Systems Settings*** dialog. The service dialog gives you controls to start and stop the database, but it should already be configured to start automatically when you restart your computer.



MongoDB Server (MongoDB)

Stop the service
Restart the service

Description:
MongoDB Database Server
(MongoDB)

Name	Description	Status	Startup Type	Log On As
Microsoft Update Health Ser...	Maintains U...	Running	Automatic (De...	Local System
Microsoft Windows SMS Ro...	Routes mess...		Manual (Trigg...	Local Service
MongoDB Server (MongoDB)	MongoDB D...	Running	Automatic	Network Service
Natural Authentication	Signal aggre...		Manual (Trigg...	Local System
Net.Tcp Port Sharing Service	Provides abil...		Disabled	Local Service
Netlogon	Maintains a ...		Manual	Local System
Network Connected Devices ...	Network Co...		Manual (Trigg...	Local Service
Network Connection Broker	Brokers con...	Running	Manual (Trigg...	Local System
Network Connections	Manages ob...		Manual	Local System

2.1.1 Installing MongoDB manually (optional)

On ***macOS*** you can install ***MongoDB*** using ***brew*** by typing the following at the command line

```
$ brew install mongodb-atlas
$ atlas setup
```

Alternatively you can unzip the MongoDB server to a local file system and add the right commands to your operating system ***PATH*** environment variable. On ***macOS***, unzip the file into ***/usr/local*** which should create a directory such as ***/usr/local/mongodb-macos-x86_64-5.0.3*** (your version might differ). To be able to execute the database related commands, add the path to the ***.bash_profile*** or ***.zshrc*** file located in your home directory. Add the following line in the configuration file as shown below. Your actual version might differ.

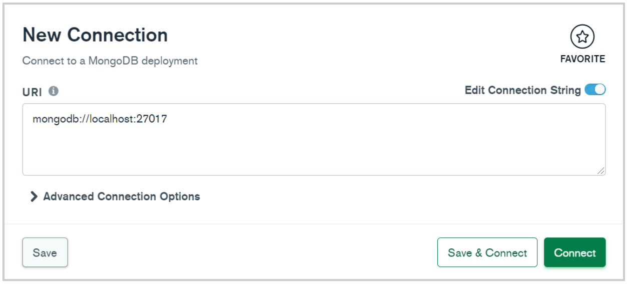
```
~/.bash_profile or ~/.zshrc

export PATH="$PATH:/usr/local/mongodb-macos-x86_64-5.0.3/bin"
```

If the ***.bash_profile*** or ***.zshrc*** file does not exist in your home directory, create it as a plain text file, but with no extensions and a period in front of it. Configure it as shown above and then ***restart your computer***.

On ***Windows***, unzip the file into ***C:\Program Files***. To configure environment variables on ***Windows*** press the ***Windows + R*** key combination to open the ***Run*** prompt, type ***sysdm.cpl*** and press ***OK***. In the ***System Properties*** window that appears, press the ***Advanced*** tab and then the ***Environment Variables*** button. In the ***Environment Variables*** configuration window select the ***Path*** variable and press the ***Edit*** button. Copy and paste the path of the ***bin*** directory in the ***mongodb*** directory you unzipped the MongoDB download, e.g., "***C:\Program Files\mongodb-macos-x86_64-5.0.3\bin***". The actual path might differ. Press ***OK*** and ***restart the computer***.

2.2 Using MongoDB Compass to interact with the MongoDB database

Your installation should have installed ***MongoDB Compass***, a user interface client to the MongoDB database. You can start the ***Compass*** from your applications folder, or search for it in your operating system's search feature. On macOS bring up ***Spotlight*** by pressing the magnifying glass on the top right menu bar, or press the ***Command*** (⌘) and Spacebar. Type MongoDB ***Compass*** in the search bar and select the application from the result list. On Windows press the Window key to bring up the search field, type MongoDB Compass, and select the application from the result list. When Compass comes up, confirm that the connection string ***mongodb://127.0.0.1:27017*** appears in the New Connection screen, and press ***Connect*** to connect to MongoDB.


New Connection

Connect to a MongoDB deployment

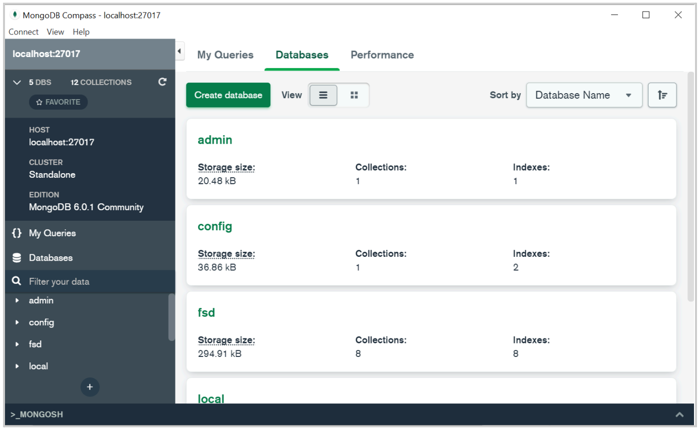
URI

Edit Connection String

mongodb://localhost:27017

2.3 Creating a MongoDB database

Once you are connected to a running MongoDB server, click on ***Databases*** on the left side bar and then click the ***Create database*** button on the ***Databases*** tab. In the ***Create Database*** dialog that appears, name your database ***kanbas*** and your first collection as ***users***. Click ***Create Database*** to create the ***kanbas*** database.


MongoDB Compass - localhost:27017

localhost:27017

5 DBS 12 COLLECTIONS

HOST localhost:27017

CLUSTER Standalone

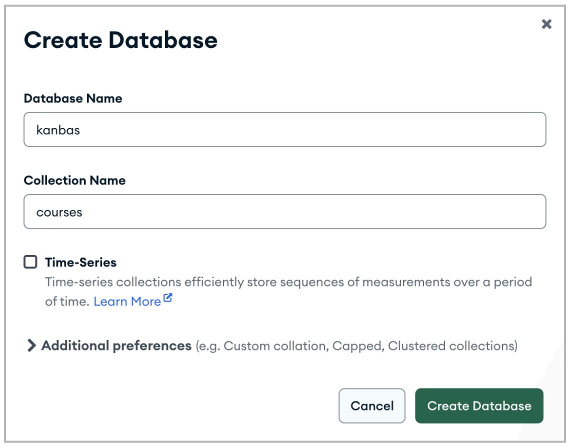
EDITION MongoDB 6.0.1 Community

admin — Storage size: 20.48 kB, Collections: 1, Indexes: 1

config — Storage size: 36.86 kB, Collections: 1, Indexes: 2

fsd — Storage size: 294.91 kB, Collections: 8, Indexes: 8

local


Create Database

Database Name

kanbas

Collection Name

courses

Time-Series

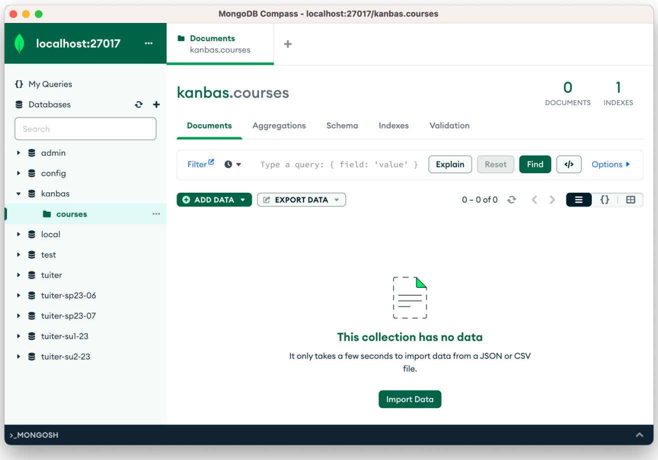
Time-series collections efficiently store sequences of measurements over a period of time. Learn More

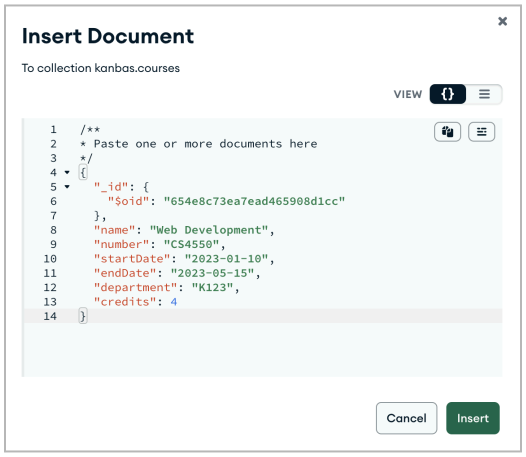
Additional preferences (e.g. Custom collation, Capped, Clustered collections)

2.4 Inserting and retrieving data into a MongoDB database with Compass

In MongoDB, data is organized into ***collections***, not ***tables*** like in relational databases. Data contained in collections are referred to as ***documents***, not ***records*** like in other databases. To create, or ***insert*** documents into a collection in a MongoDB database, select the database on the left sidebar and then select the collection you want to insert documents into. For instance, select the ***kanbas*** database and then the ***courses*** collection as shown here on the right. On the right side, selected ***Add Data*** and then ***Insert Document***. In the ***Insert to Collection kanbas.courses*** dialog that appears, insert the document as shown below. Click ***Insert*** to insert the document. Confirm the document inserted as expected.

You can also import entire JSON files containing data. Import the ***courses.json*** file we used in earlier assignments under the ***Database*** directory of your project. To import click ***ADD DATA***, then ***Import JSON or CSV file***. Navigate to the location of ***courses.json***, select the file and click ***Import***. Confirm the courses are imported. Also create the following collections and import the JSON files linked to each of the collection names. Confirm all collections are imported: modules.json, assignments.json, users.json





```
/**
* Paste one or more documents here
*/
{
  "_id": {
    "$oid": "654e8c73ea7ead465908d1cc"
  },
  "name": "Web Development",
  "number": "CS4550",
  "startDate": "2023-01-10",
  "endDate": "2023-05-15",
  "department": "K123",
  "credits": 4
}
```

Note the objects stored in the database have a primary key ***_id*** automatically added by MongoDB when they were inserted. MongoDB primary keys are of type ***ObjectId*** and are created automatically by the database so your ***_id*** values will differ from the ones shown above.

2.5 Interacting with a MongoDB database with a Command Line (optional)

Compass is a great graphical user interface to the MongoDB database, but there is also value to knowing how to interact with the database through a command line interface. At the bottom of the ***Compass*** window there's a ***_MONGOSH*** window you can expand to type commands to the database. Let's practice a few commands to retrieve data on the command line. First select the database we want to interact with.

```
> use kanbas
'switched to db kanbas
```

All the documents in a collection can be retrieved using the ***find()*** command on a collection as shown below.

```
> db.courses.find();
```

Documents in a collection can be retrieved by pattern matching their properties. The example below illustrates how to retrieve documents by pattern matching their primary key ***_id***, that is, retrieving the document whose ***_id*** field matches ***ObjectId('6370104926906053f1597ce6')***. Your ID will likely be different.

```
> db.courses.find({_id: ObjectId("654e8c73ea7ead465908d1cc")})
{
  _id: ObjectId("654e8c73ea7ead465908d1cc"),
  name: 'Web Development',
  number: 'CS4550',
  startDate: '2023-01-10',
  endDate: '2023-05-15',
  department: 'K123',
  credits: 4
}
```

We can also pattern match any of the other fields individually or combined with other fields. The following example retrieves a document from the ***courses*** collection whose ***number*** property is equal to ***RS4560***.

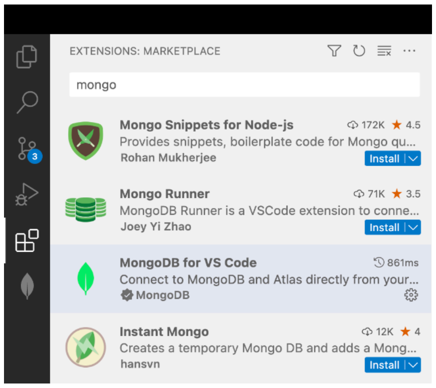
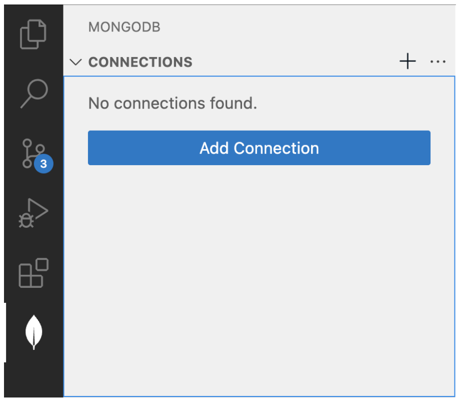
```
> db.courses.find({number: 'RS4560'})
{
  _id: 'RS102',
  name: 'Aerodynamics',
  number: 'RS4560',
  startDate: '2023-01-10',
  endDate: '2023-05-15'
}
```

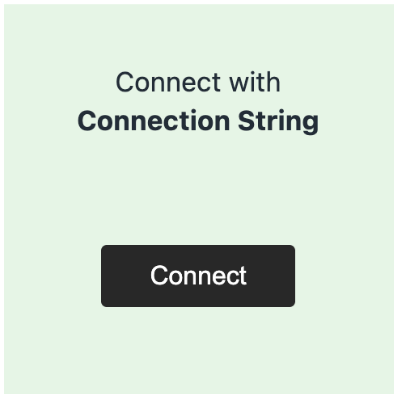
Here's another example retrieving ***courses*** in the ***D134*** department.

```
> db.courses.find({department: 'D134'})
{ _id: 'CH101',
  name: 'Organic Chemistry',  number: 'CH1230',
  startDate: '2023-01-10',  endDate: '2023-05-15',
  department: 'D134',  credits: 3
}
{ _id: 'CH102',
  name: 'Inorganic Chemistry',  number: 'CH1240',
  startDate: '2023-01-10',  endDate: '2023-05-15',
  department: 'D134',  credits: 3
}
{ _id: 'CH103',
  name: 'Physical Chemistry',  number: 'CH1250',
  startDate: '2023-01-10',  endDate: '2023-05-15',
  department: 'D134',  credits: 3
}
```

2.6 Interacting with a MongoDB with VS Code Mongo Extension (optional)

If you are using ***VS Code*** as your IDE, you can install the ***MongoDB*** extension to interact with ***MongoDB*** right from the IDE. From your ***Extensions Marketplace***, search for ***mongo***, and install the MongoDB for ***VS Code*** extension as shown below. Once installed, click on the new MongoDB icon on the side toolbar and then click on ***Add Connection*** and then ***Connect***.


EXTENSIONS: MARKETPLACE
mongo
Mongo Snippets for Node-js
172K ★ 4.5
Provides snippets, boilerplate code for Mongo qu...
Rohan Mukherjee
Install
Mongo Runner
71K ★ 3.5
MongoDB Runner is a VSCode extension to conne...
Joey Yi Zhao
Install
MongoDB for VS Code
861ms
Connect to MongoDB and Atlas directly from your...
MongoDB
Instant Mongo
12K ★ 4
Creates a temporary Mongo DB and adds a Mong...
hansvn
Install




In the text field that appears at the top of the screen, type the connection string to the MongoDB server instance you want to connect to, e.g., ***mongodb://127.0.0.1:27017*** and press enter. Confirm you can see the ***kanbas*** database and the collections inserted earlier in this assignment

2.7 Exercises

1. Install MongoDB as described in this section
2. Create the database as described in this section
3. Start the database as described in this section
4. Insert the data as described in this section
5. Update the data as described in this section
6. Delete the data as described in this section

3 Programming with a MongoDB database

In the previous section we practiced interacting with the MongoDB database through the Compass graphical interface as well as manually on the command line with ***MONGOSH***. This is all and good to make occasional simple queries to confirm the data behaves as expected, but to create applications we're going to need to interact with the database programmatically with libraries such as ***Mongoose***. The following sections describe how to install, configure, and connect a Node.js application to a MongoDB database server using the ***Mongoose*** library. The final section discusses how to configure the application to integrate to a MongoDB database hosted in the Atlas cloud service. Do all your work in a new GitHub branch called ***a6*** in both your React.js and Node.js projects.

3.1 Installing and connecting to a MongoDB database

The ***Mongoose*** library provides a set of operations and abstractions that enhance a MongoDB database and leverages the familiarity of the MONGOSH command line client. To use the Mongoose library, install it from the root of the Node.js project as shown below.

```
$ npm  install  mongoose
```

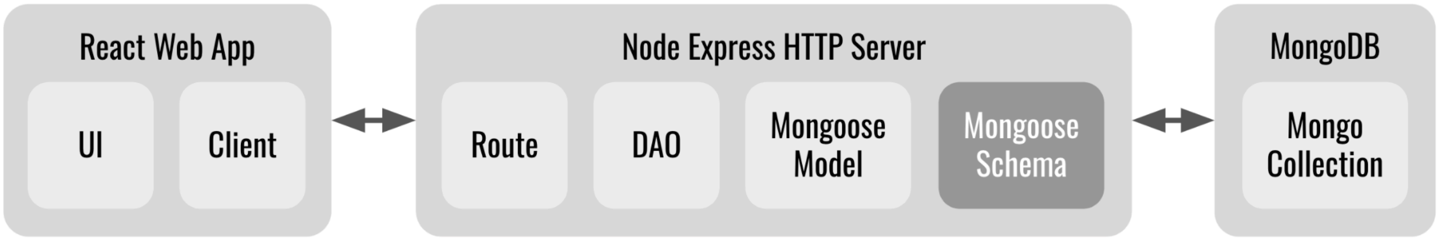
To connect to the database server programmatically, import the Mongoose library and then use the ***connect*** function as shown below. The URL in the ***connect*** function is called the ***connection string*** and is currently referring to a MongoDB server instance running in the ***localhost*** machine (your current laptop or desktop) listening at port ***27017*** and the ***kanbas*** database existing in that server. In the following section we'll revisit the connection string and configure it to connect to a database server running in a remote machine hosted by Mongo's Atlas cloud service.

App.js

```
import express from "express";
import mongoose from "mongoose";                          // load the mongoose library
...
mongoose.connect("mongodb://127.0.0.1:27017/kanbas");     // connect to the kanbas database
const app = express();
...
```

3.2 Implementing Mongoose schemas and models

Now that we have the collections setup in our database, let's now discuss how to connect to the collections in the database using the ***Mongoose*** library. We'll create Mongoose ***Schemas*** and ***Models*** so that we can connect and interact to the database programmatically.



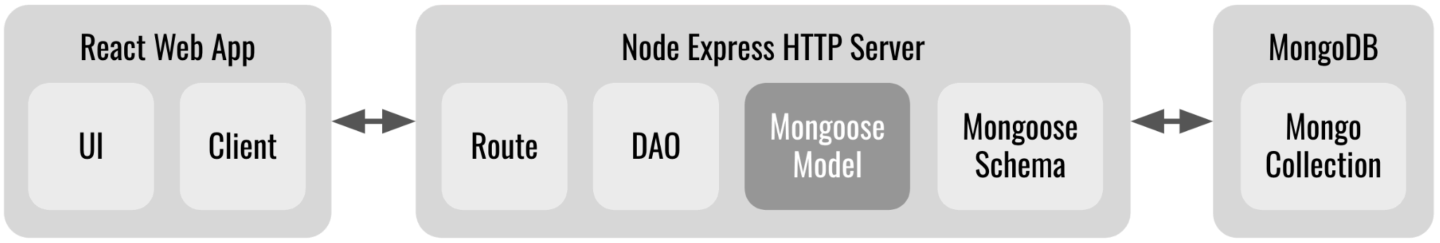
As mentioned earlier, non relational database do not require specifying the structure, or schema of the data stored in collections. That responsibility has been delegated to the applications using the non relational databases. Mongoose ***schemas*** describe the structure of the data being stored in the database and it's used to validate the data being stored of modified through the Mongoose library. The ***schema*** shown below describes the structure for the ***users*** collection we worked on earlier. Create the schema in a ***users*** folder in your Node.js projects.

Users/schema.js

```
import mongoose from "mongoose";                                  // load the mongoose library
const userSchema = new mongoose.Schema({                          // create the schema
    username: { type: String, required: true, unique: true },     // String field that is required and unique
    password: { type: String, required: true },                   // String field that in required but not unique
    firstName: String,                                   // String fields
    email: String,                                       // with no additional
    lastName: String,                                    // configurations
    dob: Date,                                           // Date field with no configurations
    role: {
      type: String,                                      // String field
      enum: ["STUDENT", "FACULTY", "ADMIN", "USER"],     // allowed string values
      default: "USER",},                                 // default value if not provided
  },
  { collection: "users" });
export default userSchema;                               // store data in "users" collection
```

3.3 Implementing Mongoose models

In earlier sections we demonstrated using the command line client to interact manually with the mongo server using the ***find*** command. Mongoose ***models*** provide similar functions to interact with MongoDB programmatically instead of manually. The functions are similar to the ones found in the mongo shell client: ***find()***, ***create()***, ***updateOne()***, ***removeOne()***, etc.



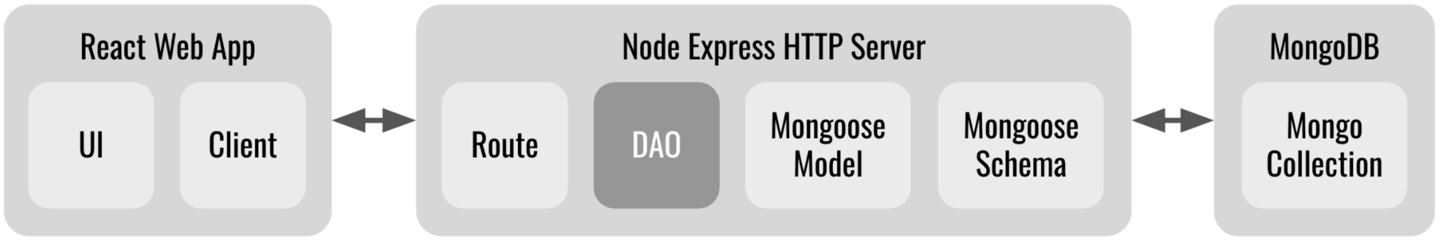
In ***Users/model.js*** below, create a Mongoose model from the users schema. The functions provided by Mongoose models are deliberately generic because they can interact with any collection configured in the schema. In the next section we'll create a ***data access object*** that implements higher level functions specific to the domain of ***kanbas***.

Users/model.js

```
import mongoose from "mongoose";                      // load mongoose library
import schema from "./schema.js";                     // load users schema
const model = mongoose.model("UserModel", schema);    // create mongoose model from the schema
export default model;                                 // export so it can be used elsewhere
```

3.4 Retrieving data from Mongo with Mongoose

The Mongoose model created in the previous section provides low level functions such as ***find***, ***create***, ***updateOne***, and ***deleteOne***, that are deliberately vague since they need to be able to operate on any collection. It is good practice to wrap these low level generic functions into higher level functions that are specific to the use cases of the specific projects. For instance instead of just using the generic ***find()*** function, we'd prefer something such as ***findUsers()*** or ***findUserById()*** or ***findUserByUsername()***.



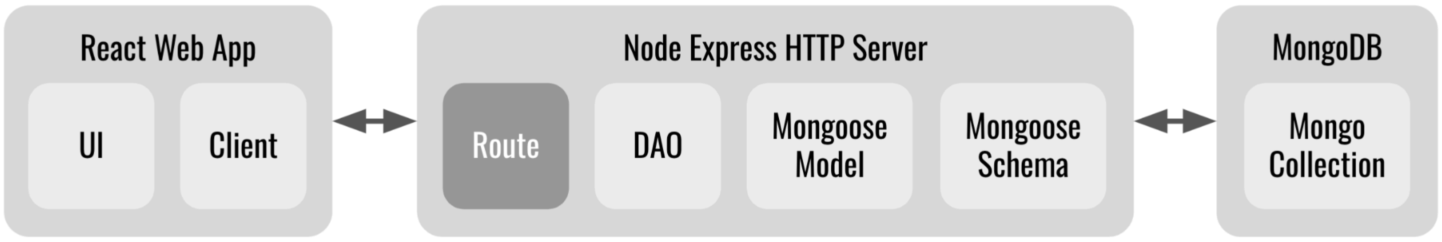
The ***data access object*** (***DAO***) ***design pattern*** implements this encapsulation and abstraction principle by grouping data access by data type or collection. The following ***Users/dao.js*** implements various ***CRUD*** operations for the ***users*** collection written in terms of the low level Mongoose model operations.

Users/dao.js

```
import model from "./model.js";
export const createUser = (user) => model.create(user);
export const findAllUsers = () => model.find();
export const findUserById = (userId) => model.findById(userId);
export const findUserByUsername = (username) =>  model.findOne({ username: username });
export const findUserByCredentials = (username, password) =>  model.findOne({ username, password });
export const updateUser = (userId, user) =>  model.updateOne({ _id: userId }, { $set: user });
export const deleteUser = (userId) => model.deleteOne({ _id: userId });
```

3.5 Implementing APIs to interact with MongoDB from a React client application

DAOs implement an interface between an application and the low level database access, providing a high level API to the rest of the application hiding the details and idiosyncrasies of using a particular database vendor. Likewise routes implement an interface between the HTTP network world and the JavaScript object and function world by converting a stream of bits from a network connection request into a set of objects, maps, and function event handlers that participate in the client/server architecture of a multi tiered application.



The Node.js server we've been implementing uses routes to interact with the user interface and the DAOs to talk to the database. The server sits between these two layers and therefore it is often referred to as the ***middle tier*** in a ***multi tiered application***. The following routes make the database operations available through a RESTful API. We'll implement each of the functions in the following sections.

Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  const createUser = async (req, res) => { };
  const deleteUser = async (req, res) => { };
  const findAllUsers = async (req, res) => { };
  const findUserById = async (req, res) => { };
  const updateUser = async (req, res) => { };
  const signup = async (req, res) => { };
  const signin = async (req, res) => { };
  const signout = (req, res) => { };
  const profile = async (req, res) => { };
  app.post("/api/users", createUser);
  app.get("/api/users", findAllUsers);
  app.get("/api/users/:userId", findUserById);
  app.put("/api/users/:userId", updateUser);
  app.delete("/api/users/:userId", deleteUser);
  app.post("/api/users/signup", signup);
  app.post("/api/users/signin", signin);
  app.post("/api/users/signout", signout);
  app.post("/api/users/profile", profile);
}
```

```
//
```

Import and configure the routes in ***App.js*** as shown below.

App.js

```
import express from "express";
import mongoose from "mongoose";
import UserRoutes from "./Users/routes.js";
import cors from "cors";
mongoose.connect("mongodb://127.0.0.1:27017/kanbas");
const app = express();
app.use(cors());
app.use(express.json());
UserRoutes(app);
app.listen(4000);
```

```
//
```

3.5.1 Retrieving a single document from MongoDB with Mongoose

DAOs implement high level data functions based on lower level ***Mongoose*** models. The ***Mongoose*** model ***findOne*** function retrieves a single document that matches the fields with the parameters. The ***findUserByCredentials*** function below uses ***findOne*** to retrieve a document that matches properties ***username*** and ***password*** with the parameters ***usr*** and ***pass*** passed from the user interface through the RESTful API.

Users/dao.js

```
import model from "./model.js";
export const findUserByCredentials = (username, password) => model.findOne({ username, password });
```

Routes implement RESTful Web APIs that clients can use to interact with server functionality. The route implemented below extracts properties ***username*** and ***password*** from the request's body and passess them to the ***findUserByCredentials*** function implemented by the DAO above. The resulting user is stored in the server variable ***currentUser*** to remember the logged in user. The user is then sent to the client in the response. Later sections ask you to add error handling.

Users/routes.js

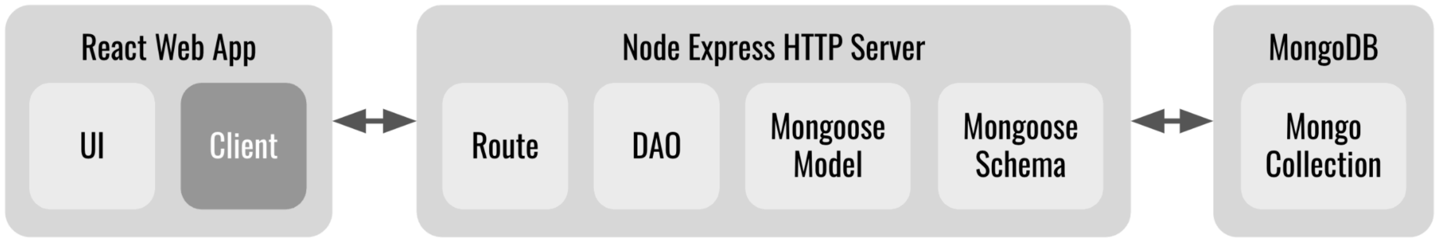
```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  ...
  const signin = async (req, res) => {
    const { username, password } = req.body;
    currentUser = await dao.findUserByCredentials(username, password);
    res.json(currentUser);
  };
  ...
  app.post("/api/users/signin", signin);
}
```

Now let's switch over to the React.js Web app project. Declare an environment variable as shown below.

.env.local

```
REACT_APP_BASE_API_URL=http://localhost:4000
```

```
//
```



Under ***Users***, implement the ***client*** shown below to interact with the user routes implemented in the server. The client function ***signin*** shown below posts a ***credentials*** object containing the ***username*** and ***password*** expected by the server. If the credentials are found, the response should contain the logged in user.

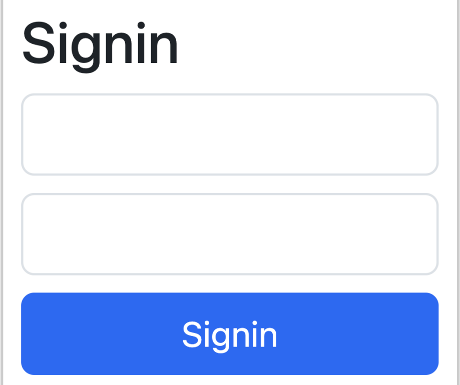
src/Users/client.ts

```
import axios from "axios";
export const BASE_API = process.env.REACT_APP_BASE_API_URL;
export const USERS_API = `${BASE_API}/api/users`;
export interface User { _id: string; username: string; password: string; role: string;
firstName: string, lastName: string };
export const signin = async (credentials: User) => {
  const response = await axios.post( `${USERS_API}/signin`, credentials );
  return response.data;
};
```

Implement a ***Sign in*** screen users can use to authenticate with the application. The following component declares state variable ***credentials*** to edit the ***username*** and ***password***. Clicking the ***Sign in*** button sends the ***credentials*** to the server using the ***client.signin*** function. When the server responds successfully, we navigate to the ***Profile*** screen, implemented in a later section.

src/Users/Signin.tsx

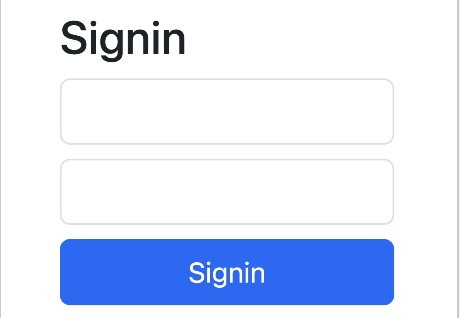
```
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import { User } from "./client";
import * as client from "./client";
export default function Signin() {
  const [credentials, setCredentials] = useState<User>({ _id: "",
    username: "", password: "", firstName: "", lastName: "", role: "USER"
  });
  const navigate = useNavigate();
  const signin = async () => {
    await client.signin(credentials);
    navigate("/Kanbas/Account/Profile");
  };
  return (
    <div>
      <h1>Signin</h1>
      <input value={credentials.username} onChange={(e) =>
        setCredentials({ ...credentials, username: e.target.value })}/>
      <input value={credentials.password} onChange={(e) =>
        setCredentials({ ...credentials, password: e.target.value })}/>
      <button onClick={signin}> Signin </button>
    </div>
  );
}
```



Implement a default navigation route to the ***Signin*** screen as shown below. When users click the ***Account*** link in the ***Kanbas Navigator***, they should see the ***Signin*** screen.

src/Kanbas/Account/index.tsx

```
import Signin from "../../Users/Signin";
import { Routes, Route, Navigate } from "react-router-dom";
export default function Account() {
  return (
    <div className="container-fluid">
      <Routes>
        <Route path="/" element={<Navigate to="/Kanbas/Account/Signin" />} />
        <Route path="/Signin" element={<Signin />} />
      </Routes>
    </div>
  );
}
```



Add the ***Account*** component to the ***Kanbas*** router as shown below.

src/Kanbas/index.tsx

```
import Account from "./Account";
import KanbasNavigator from "./Navigation";
export default function Kanbas() {
  return (
    <div className="d-flex">
      <KanbasNavigator />
      <div>
        <Routes>
          <Route path="/Account/*" element={<Account />} />
          <Route path="/Dashboard" element={<Dashboard />} />
          <Route path="/Courses/:courseId" element={<Courses />} />
        </Routes>
      </div>
    </div>
);}
```

```
//
```

3.5.2 Retrieving the currently signed in account

When someone successfully signs in, the account information is stored in a server variable called ***currentUser***. The variable will remember who is currently signed in as long as the server is running. We implemented the ***Sign in*** screen so that it would navigate to the ***Profile*** route after signing in. Let's now implement a new ***Profile*** screen mapped to the ***Profile*** route where we can display information about the user that just signed in. First let's implement a route in the server that will give us access to the ***currentUser*** as shown below.

Users/routes.js

```
let currentUser = null;
export default function UserRoutes(app) {
  const signin = async (req, res) => { ... };
  const profile = async (req, res) => {
    res.json(currentUser);
  };
  app.post("/api/users/signin", signin);
  app.post("/api/users/profile", profile);
}
```

Then in the React.js Web app, let's implement a function to retrieve the ***account*** information from the server route implemented above as shown below.

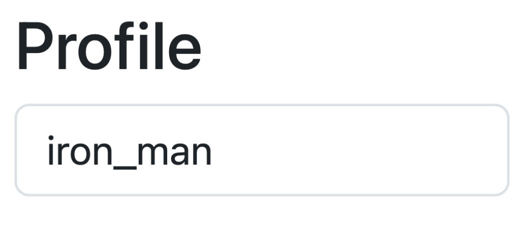
Users/client.ts

```
import axios from "axios";
export const USERS_API = process.env.REACT_APP_API_URL;
export const signin = async (user) => { ... };
export const profile = async () => {
  const response = await axios.post(`${USERS_API}/profile`);
  return response.data;
};
```

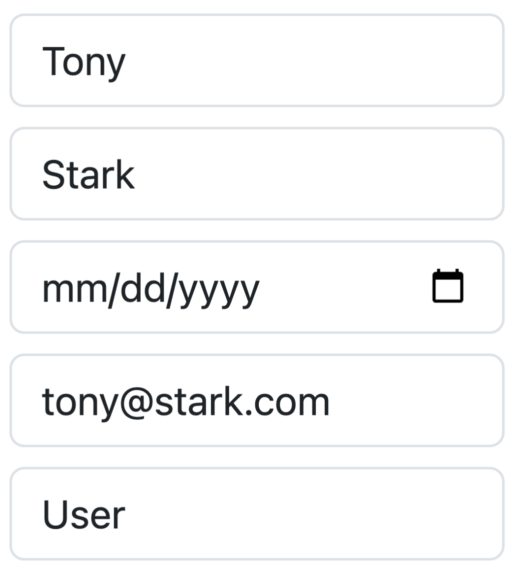
Implement the ***Profile*** screen as shown below to use the ***profile*** client function to retrieve the ***currentUser*** from the server and display it in the user interface. Styling has been removed for brevity.

Users/Profile.tsx

```
import * as client from "./client";
import { useState, useEffect } from "react";
import { useNavigate } from "react-router-dom";
export default function Profile() {
  const [profile, setProfile] = useState({ username: "", password: "",
    firstName: "", lastName: "", dob: "", email: "", role: "USER" });
  const navigate = useNavigate();
  const fetchProfile = async () => {
    const account = await client.profile();
```



```
    setProfile(account);
  };
  useEffect(() => {
    fetchProfile();
  }, []);
  return (
    <div>
      <h1>Profile</h1>
      {profile && (
        <div>
          <input value={profile.username} onChange={(e) =>
            setProfile({ ...profile, username: e.target.value })}/>
          <input value={profile.password} onChange={(e) =>
            setProfile({ ...profile, password: e.target.value })}/>
          <input value={profile.firstName} onChange={(e) =>
            setProfile({ ...profile, firstName: e.target.value })}/>
          <input value={profile.lastName} onChange={(e) =>
            setProfile({ ...profile, lastName: e.target.value })}/>
          <input value={profile.dob} type="date" onChange={(e) =>
            setProfile({ ...profile, dob: e.target.value })}/>
          <input value={profile.email} onChange={(e) =>
            setProfile({ ...profile, email: e.target.value })}/>
          <select onChange={(e) =>
              setProfile({ ...profile, role: e.target.value })}>
            <option value="USER">User</option>
            <option value="ADMIN">Admin</option>
            <option value="FACULTY">Faculty</option>
            <option value="STUDENT">Student</option>
          </select>
        </div>
      )}
    </div>
  );
}
```



Tony

Stark

mm/dd/yyyy

tony@stark.com

User

In the ***Account*** component, add routes and links as necessary to navigate to the ***Profile*** screen as necessary. Confirm that signing in navigates to the new ***Profile*** screen and displays information about the user that just signed in. A later section will add ability to edit the account information.

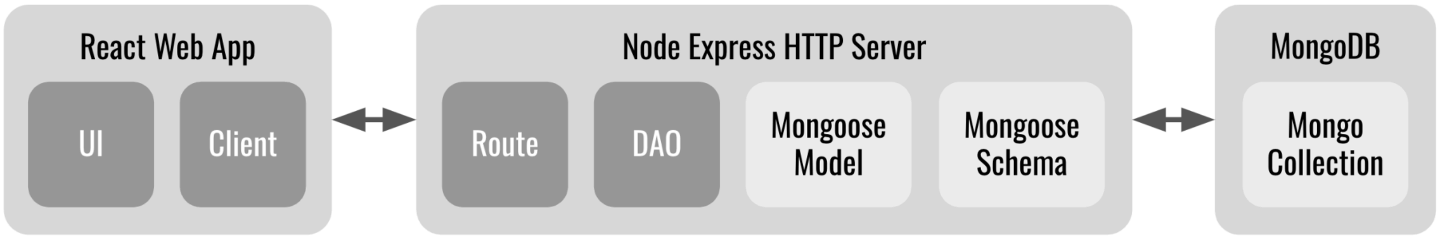
src/Kanbas/Account/index.tsx

```
import Signin from "../../Users/Signin";
import Profile from "../../Users/Profile";
...
export default function Account() {
  ...
  return (
        ...
        <Routes>
          <Route path="/" element={<Navigate to="/Kanbas/Account/Signin" />} />
          <Route path="/Signin" element={<Signin />} />
          <Route path="/Profile" element={<Profile />} />
        </Routes>
  );
}
```

3.5.3 Updating a document in MongoDB with Mongoose

In the previous section we implemented ***findUserByCredentials***, which retrieved a user by pattern matching the ***username*** and ***password*** submitted from the user interface to the server, with a matching document with the same properties. This is one of several read operations we'll implement in this assignment. Let's turn our attention to implementing an update operation as we explore several ***CRUD*** functions. All the exercises consist of

1. Implementing a function in the DAO to to interact with the database
2. Implementing a route to make the function available as a RESTful Web API
3. Implementing a client function to integrate the user interface with the server's RESTful Web API, AND
4. Implementing a user interface React component or screen that uses the client function to interact with the server



First let's start by implementing a DAO function to interact with the database. The ***updateUser*** DAO function below updates a single document by first identifying it by its primary key, and then updating the matching fields in the ***user*** parameter.

Users/dao.js

```
import model from "./model.js";
export const updateUser = (userId, user) => model.updateOne({ _id: userId }, { $set: user });
```

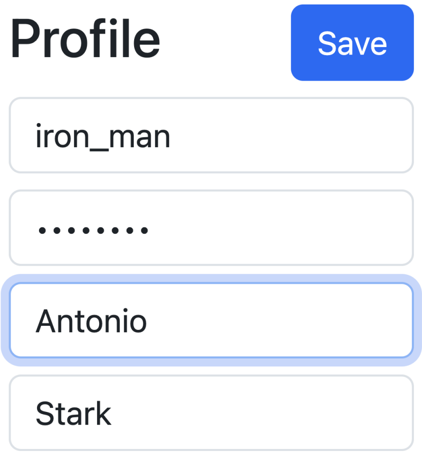
Now we make the DAO function available as RESTful Web API as shown below. We map a route that accepts a user's primary key as a path parameter, passes the ID and request body to the DAO function and then updates the ***currentUser*** with the updated user. We respond with the status.

Users/routes.js

```
export default function UserRoutes(app) {
  ...
  const updateUser = async (req, res) => {
    const { userId } = req.params;
    const status = await dao.updateUser(userId, req.body);
    currentUser = await dao.findUserById(userId);
    res.json(status);
  };
  ...
  app.put("/api/users/:userId", updateUser);
}
```

```
//
```

Now that we're done with the server, let's focus on the client in the React.js Web app. Implement the ***updateUser*** client function as shown below to interact with the ***updateUser*** route implemented in the server. We'll send an ***HTTP PUT*** request with the user's ID encoded as a path parameter, and the user updates in the request body.



iron_man
••••••••
Antonio
Stark

Users/client.ts

```
export const updateUser = async (user: any) => {
  const response = await axios.put(`${USERS_API}/${user._id}`, user);
  return response.data;
};
```

We can then use the new ***updateUser*** client function in the ***Account*** screen to send the ***account*** information to the server to update the ***currentUser*** when we click the ***Save*** button.

Users/Profile.tsx

```
  ...
  const save = async () => {
    await client.updateUser(profile);
  };
  ...
  <button onClick={save}>
     Save
  </button>
  ...
```

3.5.4 Retrieving all documents from MongoDB with Mongoose

Another common read operation is to retrieve all the documents from a collection. As always, we'll implement a ***DAO*** function to interact with the database. The ***DAO findAllUsers*** function retrieves all the users from the document from the ***users*** collection.

Users/dao.js

```
import model from "./model.js";
export const findAllUsers = () => model.find();
```

We then make the DAO function through a RESTful Web API so the user interface can interact with the database.

Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  const findAllUsers = async (req, res) => {
    const users = await dao.findAllUsers();
    res.json(users);
  };
  app.get("/api/users", findAllUsers);
}
```

```
//
```

Over on the React Web App, implement the ***findAllUsers*** client function shown below to interact with the RESTful Web API implemented earlier.

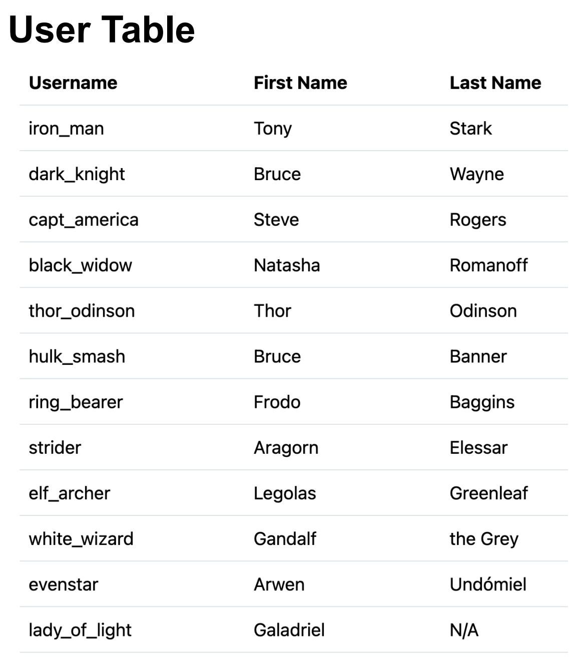
src/Users/client.ts

```
export const findAllUsers = async () => {
  const response = await axios.get(`${USERS_API}`);
  return response.data;
};
```

Implement a ***User Table*** screen that retrieves all the users from the database and renders them as a table as shown below.

src/Users/Table.tsx

```
import React, { useState, useEffect } from "react";
import * as client from "./client";
import { User } from "./client";
export default function UserTable() {
  const [users, setUsers] = useState<User[]>([]);
  const fetchUsers = async () => {
    const users = await client.findAllUsers();
    setUsers(users);
  };
  useEffect(() => { fetchUsers(); }, []);
  return (
    <div>
      <h1>User Table</h1>
      <table className="table">
        <thead>
          <tr>
            <th>Username</th>
            <th>First Name</th>
            <th>Last Name</th>
          </tr>
        </thead>
        <tbody>
          {users.map((user: any) => (
            <tr key={user._id}>
              <td>{user.username}</td>
              <td>{user.firstName}</td>
              <td>{user.lastName}</td>
```

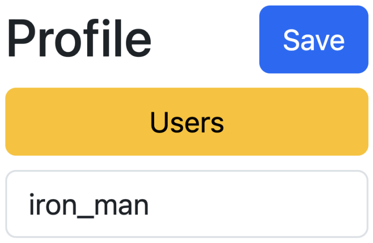


User Table

Username	First Name	Last Name
iron_man	Tony	Stark
dark_knight	Bruce	Wayne
capt_america	Steve	Rogers
black_widow	Natasha	Romanoff
thor_odinson	Thor	Odinson
hulk_smash	Bruce	Banner
ring_bearer	Frodo	Baggins
strider	Aragorn	Elessar
elf_archer	Legolas	Greenleaf
white_wizard	Gandalf	the Grey
evenstar	Arwen	Undómiel
lady_of_light	Galadriel	N/A

```
            </tr>))}
          </tbody>
        </table>
      </div>
    );
}
```

In the ***Profile*** screen, add a link to navigate to the new ***User Table*** as shown below. Add any necessary routes.



src/Users/Profile.tsx

```
  ...
  <Link to="/Kanbas/Account/Admin/Users"
    className="btn btn-warning w-100">
    Users
  </Link>
  ...
```

In the ***Account*** screen, add necessary routes as shown below.

src/Kanbas/Account/index.js

```
import UserTable from "../../Users/Table";
export default function Account() {
  return (
    <div className="container-fluid">
      <Routes>
        ...
        <Route path="/Admin/Users" element={<UserTable />} />
```

```
//
```

3.5.5 Creating new documents in MongoDB with Mongoose

So far we've implemented several reading operations and an update operation. Let's now explore creating new documents in the database. The ***DAO*** function below inserts a new ***user*** object into the ***users*** collection.

Users/dao.js

```
export const createUser = (user) => {
  delete user._id
  return model.create(user);
}
```

```
// remove _id field just in case client sends it
// database will create _id for us instead
```

The route below makes the DAO operation available as a RESTful Web API for the user interface to interact with. The new user is posted to the route in the request's body. The DAO ***createUser*** function inserts the new user into the database and returns the newly inserted ***user*** which is sent back to the user interface in the response.

Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
export default function UserRoutes(app) {
  ...
  const createUser = async (req, res) => {
    const user = await dao.createUser(req.body);
    res.json(user);
  };
  app.post("/api/users", createUser);
  ...
}
```

Over on the React Web App, create a ***createUser*** client function to interact with the route create earlier. Post the new user object to the server as shown below.

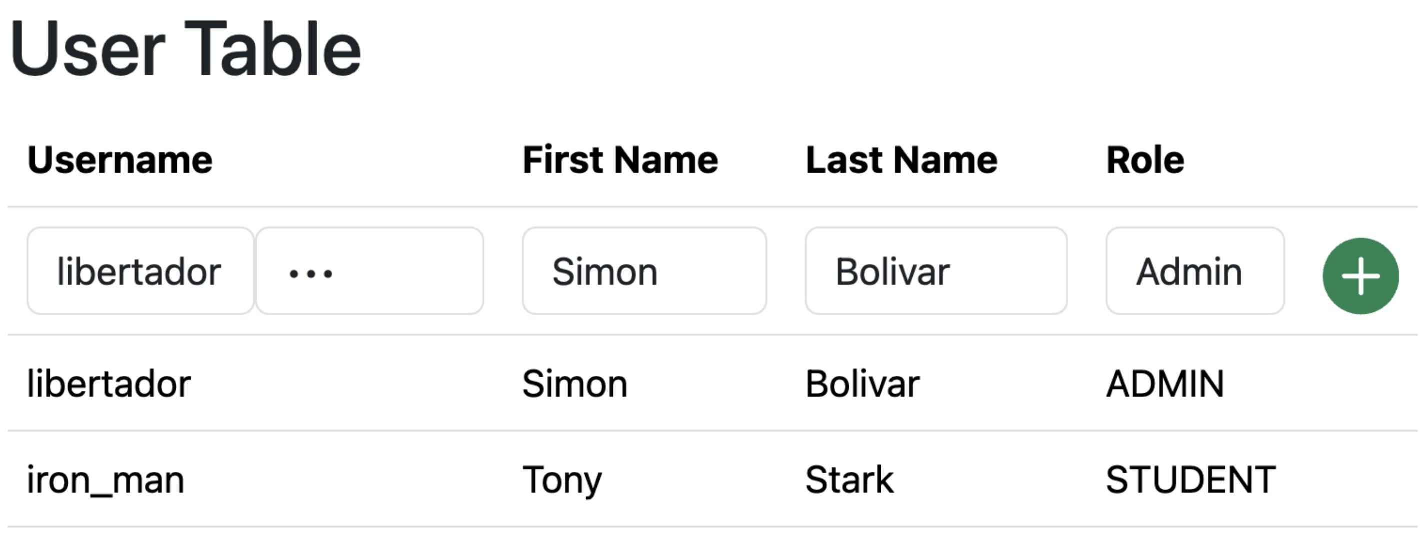
src/Users/client.ts

```
export const createUser = async (user: any) => {
  const response = await axios.post(`${USERS_API}`, user);
  return response.data;
};
```

In the ***User Table*** screen, add a ***user*** state variable so we can create the user object we want to insert into the database as shown below. The ***createUser*** function handles the click on the ***plus*** sign to insert the new user into the database. The new user is appended to the ***users*** state variable. Form elements are added to the first row of the table to capture the values of the new user including ***username***, ***password***, ***firstName***, ***lastName***, and ***role***. You might need to add an additional ***Role*** column to the header row.

Users/Table.tsx

```
import React, { useState, useEffect } from "react";
import { BsTrash3Fill, BsPlusCircleFill } from "react-icons/bs";
import * as client from "./client";
export default function UserTable() {
  const [users, setUsers] = useState<User[]>([]);
  const [user, setUser] = useState<User>({
    _id: "", username: "", password: "", firstName: "",
    lastName: "", role: "USER" });
  const createUser = async () => {
    try {
      const newUser = await client.createUser(user);
      setUsers([newUser, ...users]);
    } catch (err) {
      console.log(err);
    }
  };
  ...
  return (
    <div>
      <h1>User Table</h1>
      <table className="table">
        <thead>
          <tr> <!-- add Role to header row --> </tr>
          <tr>
            <td>
              <input value={user.password} onChange={(e) =>
                setUser({ ...user, password: e.target.value })}/>
              <input value={user.username} onChange={(e) =>
                setUser({ ...user, username: e.target.value })}/>
            </td>
            <td>
              <input value={user.firstName} onChange={(e) =>
                setUser({ ...user, firstName: e.target.value })}/>
            </td>
            <td>
              <input value={user.lastName} onChange={(e) =>
                setUser({ ...user, lastName: e.target.value })}/>
            </td>
            <td>
              <select value={user.role} onChange={(e) =>
                setUser({ ...user, role: e.target.value })}>
                <option value="USER">User</option>
                <option value="ADMIN">Admin</option>
                <option value="FACULTY">Faculty</option>
                <option value="STUDENT">Student</option>
              </select>
            </td>
            <td>
              <BsPlusCircleFill onClick={createUser}/>
            </td>
          </tr>
        </thead>
        ...
      </table>
    </div>
  );
}
```



User Table

Username		First Name	Last Name	Role	
libertador	•••	Simon	Bolivar	Admin	+
libertador		Simon	Bolivar	ADMIN	
iron_man		Tony	Stark	STUDENT	

3.5.6 Deleting a document in MongoDB with Mongoose

Finally let's implement ***deleteUser*** operation, wrapping up all the ***CRUD*** operations. The ***DAO*** function below removes a single ***user*** document from the database based on its primary key.

Users/dao.js

```
export const deleteUser = (userId) => model.deleteOne({ _id: userId });
```

The route below makes the ***deleteUser*** operation available as a RESTful Web API for integration with the user interface which encodes the id of the user to remove as a path parameter.

Users/routes.js

```
  const deleteUser = async (req, res) => {
      const status = await dao.deleteUser(req.params.userId);
      res.json(status);
  };
  app.delete("/api/users/:userId", deleteUser);
```

In the React Web App, implement client function that integrates with the ***deleteUser*** route in the server.

src/Users/client.ts

```
export const deleteUser = async (user: any) => {
  const response = await axios.delete(
    `${USERS_API}/${user._id}`);
  return response.data;
};
```

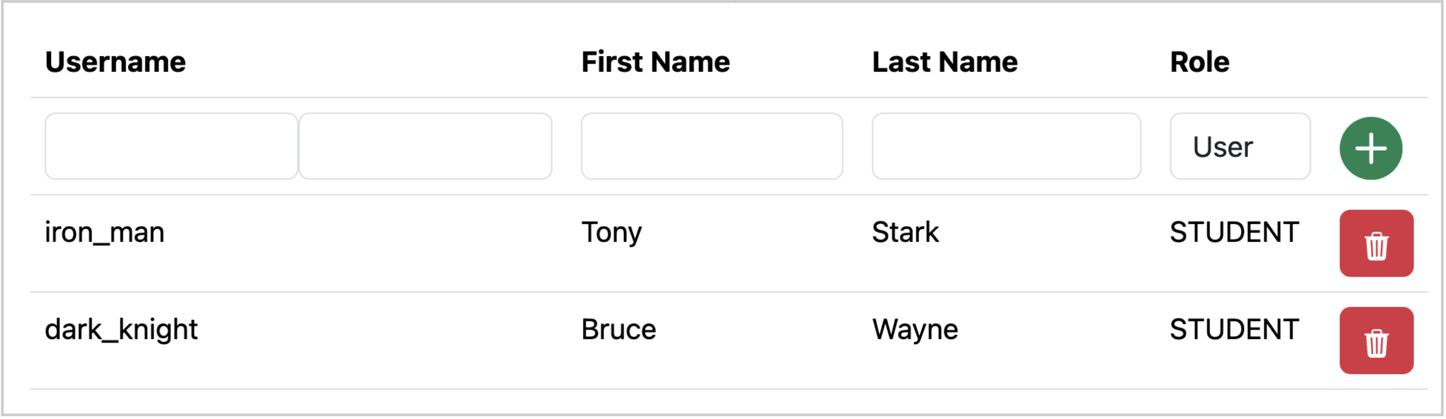
```
//
```

The ***User Table*** screen below adds a ***trash*** icon to all the user rows that invoke the ***deleteUser*** event handler witch sends the delete command to the server and then removes the user from the local ***users*** state variable. Styling has been removed for brevity.

Users/Table.tsx

```
...
import { BsTrash3Fill } from "react-icons/bs";
...
import * as client from "./client";
```

```
function UserTable() {
  const [users, setUsers] = useState<User[]>([]);
  const deleteUser = async (user: User) => {
    try {
      await client.deleteUser(user);
      setUsers(users.filter((u) => u._id !== user._id));
    } catch (err) {
      console.log(err);
    }
  };
  ...
  return (
    <div>
      <h1>User List</h1>
      <table className="table">
        <thead>
          <tr>
            ...
            <th>&nbsp;</th>
          </tr>
        </thead>
        <tbody>
          {users.map((user) => (
            <tr key={user._id}>
              <td>{user.username}</td>
              <td>{user.firstName}</td>
              <td>{user.lastName}</td>
              <td>
                <button onClick={() => deleteUser(user)}>
                  <BsTrash3Fill />
                </button>
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}
export default UserTable;
```



Username		First Name	Last Name	Role	
				User	+
iron_man		Tony	Stark	STUDENT	
dark_knight		Bruce	Wayne	STUDENT	

3.5.7 Retrieving documents by their primary key from MongoDB with Mongoose (graduate students)

Another common database operation is to retrieve documents by their primary key. The ***DAO*** function below retrieves a ***user*** document by its primary key.

Users/dao.js

```
export const findUserById = (userId) =>
  model.findById(userId);
```

```
//
```

Make the ***findUserById*** DAO function available as a RESTful Web API as shown below.

Users/routes.js

```
  const findUserById = async (req, res) => {
    const user = await dao.findUserById(req.params.userId);
    res.json(user);
  };
  app.get("/api/users/:userId", findUserById);
```

```
//
```

The user interface can then interact with the server using the ***findUserById*** client function shown below.

src/Users/client.ts

```
export const findUserById = async (id: string) => {
  const response = await axios.get(`${USERS_API}/${id}`);
  return response.data;
};
```

```
//
```

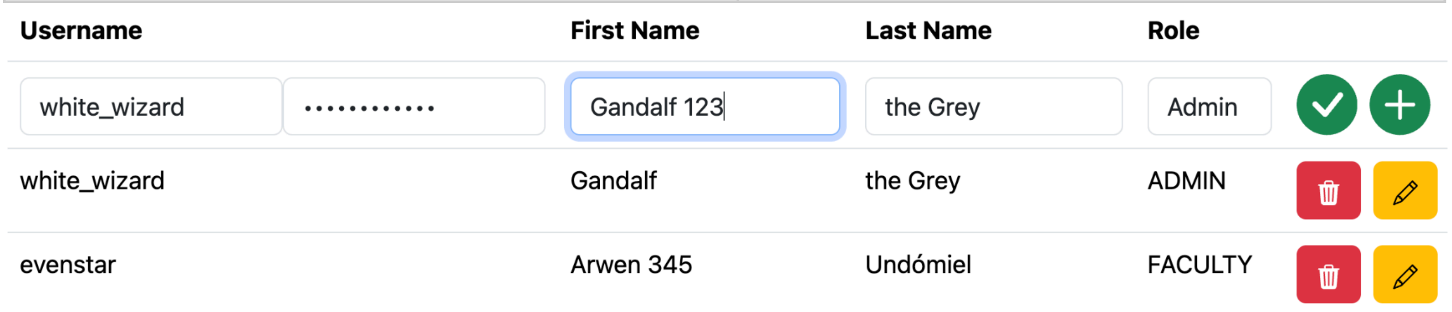
The ***User Table*** screen can then use the client's ***findUserById*** function to retrieve the user when we click on the new ***pencil*** edit button. A new ***check*** button sends the new updates to the server and updates the local array of users to match the database.

src/Users/Table.tsx

```
import {
  BsFillCheckCircleFill, BsPencil,
  BsTrash3Fill, BsPlusCircleFill,
} from "react-icons/bs";
import * as client from "./client";
...
function UserTable() {
  const [user, setUser] = useState({
    username: "", password: "", role: "USER" });
  ...
  const selectUser = async (user: User) => {
    try {
      const u = await client.findUserById(user._id);
      setUser(u);
    } catch (err) {
      console.log(err);
    }
  };
  const updateUser = async () => {
    try {
      const status = await client.updateUser(user);
      setUsers(users.map((u) =>
        (u._id === user._id ? user : u)));
    } catch (err) {
      console.log(err);
    }
  };
  ...
  <td className="text-nowrap">
    <BsFillCheckCircleFill
      onClick={updateUser}
      className="me-2 text-success fs-1 text"
    />
    <BsPlusCircleFill
      onClick={createUser}
      className="text-success fs-1 text"
    />
  </td>
  ...
  <td className="text-nowrap">
    <button className="btn btn-danger me-2">
      <BsTrash3Fill onClick={() => deleteUser(user)} />
    </button>
    <button className="btn btn-warning me-2">
      <BsPencil onClick={() => selectUser(user)} />
    </button>
  </td>
```

```
//
```

```
  ...
```



Username		First Name	Last Name	Role
white_wizard	••••••••••••	Gandalf 123	the Grey	Admin
white_wizard		Gandalf	the Grey	ADMIN
evenstar		Arwen 345	Undómiel	FACULTY

3.5.8 Retrieving documents by predicate from MongoDB with Mongoose (graduate students)

The examples so far have illustrated retrieving data from the database based on parameters encoded in the body and path. The signin screen sends credentials embedded in the body and it responds with the currently signed in user. The user table can retrieve, update, and delete users by encoding their ID in the path as a path parameter. Data sent to the server can also be encoded as a query string. Query parameters are often used to filter data by fields other than the primary key. Using query strings, let's retrieve users that are in a particular role. In the ***DAO***, add the following function to retrieve users with a particular role.

Users/dao.js

```
export const findUsersByRole = (role) => model.find({ role: role });
```

In the user routes, modify the ***findAllUsers*** function so that it parses the role from the query string, and then users the DAO to retrieve users with that particular role.

Users/routes.js

```
  const findAllUsers = async (req, res) => {
    const { role } = req.query;
    if (role) {
      const users = await dao.findUsersByRole(role);
      res.json(users);
      return;
    }
    const users = await dao.findAllUsers();
    res.json(users);
    return;
  };
```

In the user interface, add ***findUserByRole*** in the client so that it encodes the ***role*** in the query string of the URL as shown below.

Users/client.ts

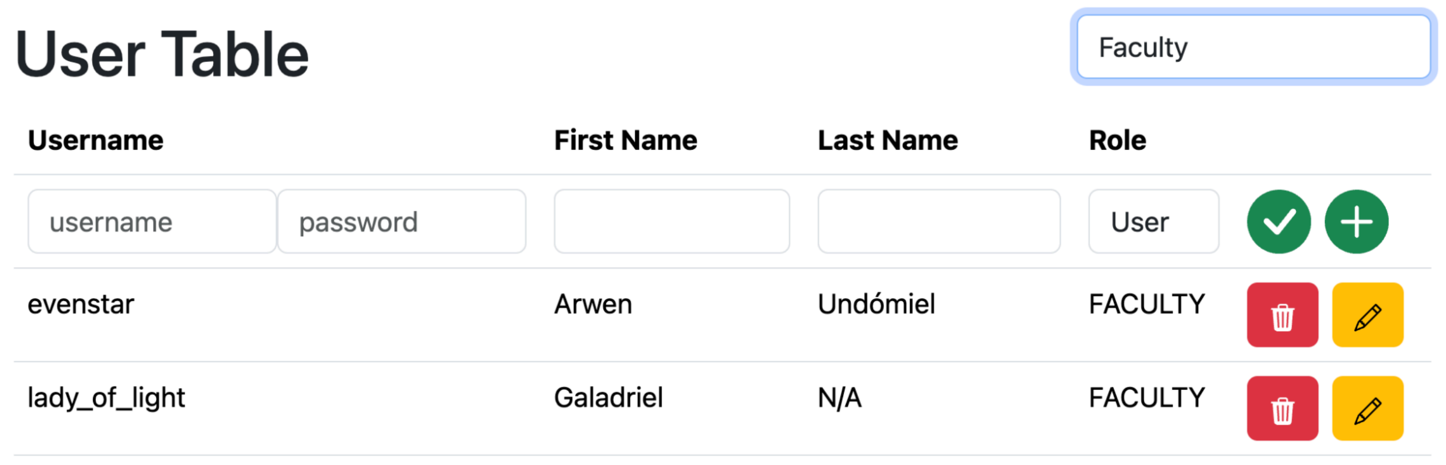
```
export const findUsersByRole = async (role: string) => {
  const response = await
    axios.get(`${USERS_API}?role=${role}`);
  return response.data;
};
```

In the ***User Table*** component, add a drop down so that administrators can filter users by their role. Confirm that selecting roles in the dropdown filters the users by their role.

src/Users/Table.tsx

```
function UserTable() {                                    //
```

```
  const [role, setRole] = useState("USER");
  const fetchUsersByRole = async (role: string) => {
    const users = await client.findUsersByRole(role);
    setRole(role);
    setUsers(users);
  };
  return (
    <div>
      <select
        onChange={(e) => fetchUsersByRole(e.target.value)}
        value={role || "USER"}
        className="form-control w-25 float-end"
      >
        <option value="USER">User</option>
        <option value="ADMIN">Admin</option>
        <option value="FACULTY">Faculty</option>
        <option value="STUDENT">Student</option>
      </select>
      <h1>User Table</h1>
      ...
    </div>
  );
}
export default UserTable;
```



User Table

Faculty

Username		First Name	Last Name	Role	
username	password			User	
evenstar		Arwen	Undómiel	FACULTY	
lady_of_light		Galadriel	N/A	FACULTY	

4 User Authentication

Now that we have an understanding of the basic types of interactions between the user interface, the server, and the database, let's consider implementing user authentication. In a previous section we discussed the ***signin*** route to allow users identify themselves and retrieve the user by their credentials. In this section we'll implement several other user authentication operations and then add support for multiple users.

4.1 Implementing a Signup screen

In the previous section we implemented the ***createUser*** and ***findUserByUsername*** functions in the User's DAO below. The ***createUser*** DAO function accepts a user object from the user interface and then uses the ***model*** to insert the user into the database. The ***findUserByUsername*** accepts a ***username*** from the user interface and users the ***model*** to retrieve the user with the matching ***username***.

Users/dao.js

```
import model from "./model.js";
export const createUser = (user) => model.create(user);
export const findUserByUsername = (username) => model.findOne({ username: username });
```

```
//
```

We can use the DAO functions to implement the ***signup*** operation for users to signup to our application. The ***signup*** route below expects a user object with at least properties ***username*** and ***password***. We use the DAO's ***findUserByUsername*** to check if there's already a user with that username. If there is, we respond with a 400 error status and respond with an error message for the user interface to display. If the username is not already taken, we insert the user into the database and store the inserted user in the ***currentUser*** server variable. We respond with the new user. The ***signup*** route is mapped to the ***/api/users/signup*** path.

Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
...
function UserRoutes(app) {
  ...
  const signup = async (req, res) => {
    const user = await dao.findUserByUsername(req.body.username);
    if (user) {
      res.status(400).json(
        { message: "Username already taken" });
    }
    currentUser = await dao.createUser(req.body);
    res.json(currentUser);
  };
  app.post("/api/users/signup", signup);
  ...
}

export default UserRoutes;
```

```
//
```

Meanwhile in the React user interface Web app, we'll implement the ***signup*** client that will post the new user to the Web API as shown below.

Users/client.ts

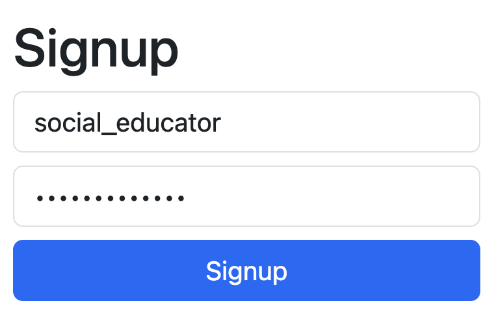
```
import axios from "axios";
export const USERS_API = process.env.REACT_APP_API_URL;
export const signup = async (user) => {
  const response = await axios.post(`${USERS_API}/signup`, user);
  return response.data;
};
```

```
//
```

Implement a ***Signup*** screen component that users can use to type their username and password, and send the credentials to the user for signup. If the signup is successful, navigate to the ***Profile*** screen. If not, display the error message. Confirm that you can signup with a new username and password. Also confirm that you display an error message if you use a username that is already in use.

Users/Signup.tsx

```
import React, { useState } from "react";
import { useNavigate } from "react-router-dom";
import * as client from "./client";
export default function Signup() {
  const [error, setError] = useState("");
  const [user, setUser] = useState({ username: "", password: "" });
  const navigate = useNavigate();
  const signup = async () => {
    try {
      await client.signup(user);
      navigate("/Kanbas/Account/Profile");
    } catch (err) {
      setError(err.response.data.message);
    }
  };
  return (
    <div>
      <h1>Signup</h1>
```



```
      {error && <div>{error}</div>}
      <input value={user.username} onChange={(e) => setUser({
          ...user, username: e.target.value })} />
      <input value={user.password} onChange={(e) => setUser({
          ...user, password: e.target.value })} />
      <button onClick={signup}> Signup </button>
    </div>
  );
}
```

4.2 Implementing Signout

Implement a route for users to signout that basically resets the ***currentUser*** to null as shown below.

Users/routes.js

```
import * as dao from "./dao.js";
let currentUser = null;
function UserRoutes(app) {
  ...
  const signout = (req, res) => {
    currentUser = null;
    res.sendStatus(200);
  };
  app.post("/api/users/signout", signout);
  ...
}
export default UserRoutes;
```

```
//
```

In the React user interface Web application, implement a client function that can post to the ***signout*** route.

Users/client.ts

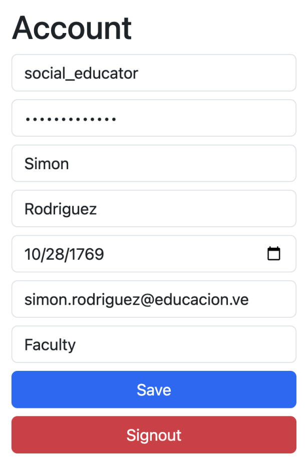
```
export const signout = async () => {
  const response = await axios.post(`${USERS_API}/signout`);
  return response.data;
};
```

```
//
```

In the ***Profile*** screen add a ***Signout*** button that invokes the ***signout*** client function and then navigates to the ***Signin*** screen. Confirm that you can signout and navigate to the ***Signin*** screen.

Users/Profile.tsx

```
  ...
  const signout = async () => {
    await client.signout();
    navigate("/Kanbas/Account/Signin");
  };
  ...
  <button onClick={save}>
    Save
  </button>
  <button onClick={signout}>
    Signout
  </button>
  ...
```


Account

social_educator

•••••••••••••

Simon

Rodriguez

10/28/1769

simon.rodriguez@educacion.ve

Faculty

4.3 Multiple user sessions

The user authentication we have implemented so far is very simple, but supports a single user at a time. Let's add session handling to our Node.js server so that multiple users can be logged in at the same time

4.4 Installing and configuring server session

First we need to narrow down who is allowed to authenticate. Configure ***cors*** to support cookies and restrict network access from the React application as shown below.

App.js

```
const app = express();
app.use(
 cors({
   credentials: true,                       // support cookies
   origin: "http://localhost:3000",         // restrict cross origin resource sharing to the react application
 })
);
app.use(express.json());
const port = process.env.PORT || 4000;
```

In your Node.js project, install the ***express-session*** library as shown below.

```
$ npm  install  express-session
```

Then in your server implementation file, import and configure the session library as shown below. Make sure to configure session ***after*** configuring cors.

App.js

```
import session from "express-session";     // import new server session library
const app = express();
app.use(cors({ ...  }));
const sessionOptions = {                    // configure server session after cors
  secret: "any string",                     // this is a default session configuration that works fine
  resave: false,                            // locally, but needs to be tweaked further to work in a
  saveUninitialized: false,                 // remote server such as AWS, Render, or Heroku. See later
};
app.use(
  session(sessionOptions)
);
```

Install the ***dotenv*** library so we can read configurations in local environment variable.

```
$ npm  install  dotenv
```

In a new ***.env*** file at the root of your project, declare environment variable ***NODE_ENV*** and set it to ***development*** as shown below.

.env

```
NODE_ENV=development
FRONTEND_URL=http://localhost:3000
SESSION_SECRET
```

Then in ***App.js***, import the ***dotenv*** library so we can determine whether we are running in our development environment and configure session accordingly.

App.js

```
import "dotenv/config";                        // import the new dotenv library to read .env file
import session from "express-session";
const app = express();
app.use(
  cors({
    credentials: true,
    origin: process.env.FRONTEND_URL           // use different front end URL in dev and in production
  })
);
const sessionOptions = {                        // default session options
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
};
if (process.env.NODE_ENV !== "development") {
  sessionOptions.proxy = true;
  sessionOptions.cookie = {
    sameSite: "none",
    secure: true,
  };
}
app.use(session(sessionOptions));
```

The register API retrieves the username and password from the request body. If there's already a user with that username, then we responds with an error. Otherwise we create the new user and store it in the session's ***currentUser*** property so we can remember that this new user is now the currently logged in user.

Users/routes.js

```
import * as dao from "./dao.js";
~~let currentUser = null;~~
...
function UserRoutes(app) {
  ...
  const signup = async (req, res) => {
    const user = await dao.findUserByUsername(req.body.username);
    if (user) {
      res.status(400).json(
        { message: "Username already taken" });
    }
    const currentUser = await dao.createUser(req.body);
    req.session["currentUser"] = currentUser;
    res.json(currentUser);
  };
```

An existing user can identify themselves by providing their credentials as username and password. The login API below looks up the user by their credentials and responds with the user if they exist. Otherwise we respond with an error.

Users/routes.js

```
  const signin = async (req, res) => {
    const { username, password } = req.body;
    const currentUser = await dao.findUserByCredentials(username, password);
    if (currentUser) {
      req.session["currentUser"] = currentUser;
      res.json(currentUser);
    } else {
      res.sendStatus(401);
    }
  };
```

If a user has already logged in, we can retrieve the current user by using the profile API as shown below.

Users/routes.js

```
const profile = (req, res) => {
    const currentUser = req.session["currentUser"];
    if (!currentUser) {
      res.sendStatus(401);
      return;
    }
    res.json(currentUser);
  };
```

Finally we can logout users by destroying the session.

Users/routes.js

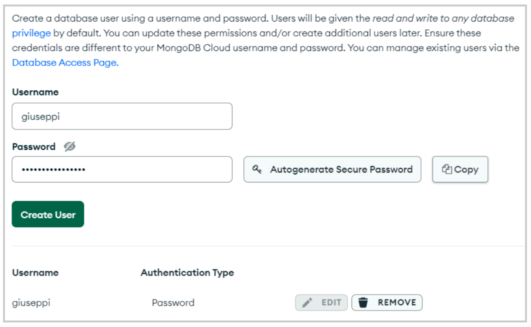
```
  const signout = (req, res) => {
    req.session.destroy();
    res.sendStatus(200);
  };
```

5 Implementing the Kanbas MongoDB

For the last several assignments we have been implementing the Kanbas application rendering various courses, modules and assignments. We used ***JSON*** files to represent the data for the application, and wrapped the data structures into a "Database" that first lived in the React application and then in the Node server. It is time for the data to live where it belongs. In this section, on your own, migrate the ***courses*** and ***modules*** into corresponding collections in the ***kanbas*** MongoDB database, create the ***Mongoose*** schemas, models and DAOs, and refactor the RESTful Web APIs to ***CRUD*** courses and modules in the database.

5 Integrating with MongoDB hosted in the cloud

When you run your server on your development environment, it should be connecting to a MongoDB instance running on the same local development computer. When you deploy the server on a remote server such as Heroku or AWS, the server needs to connect to a database that is also hosted on a public site. MongoDB provides a hosted database service where a MongoDB instances run on public servers, and they provide a connection string to integrate our Node.js application. This section describes setting up and deploying the database online and then integrating with it from our Node.js server running on Heroku.


Create a database user using a username and password. Users will be given the *read and write to any database privilege* by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password. You can manage existing users via the Database Access Page.

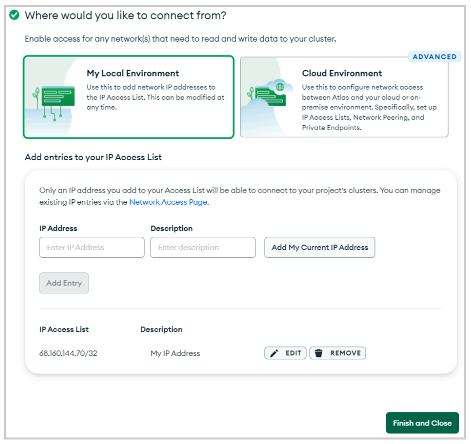
Username: giuseppi

Password: ••••••••••••••••

Username	Authentication Type
giuseppi	Password

5.1 Setting up MongoDB Atlas

To get started, head over to https://www.mongodb.com/ and click on ***Sign*** in at the top right corner. Login with your ***Google*** account or click on ***Sign Up*** to create an account with an email and password. If you get a validation email, confirm it and login. Answer any general questions if asked during the signup process. In the ***Deploy a cloud database*** screen choose a ***FREE Shared*** plan for now which should be enough for this course. In the ***Create a Shared Cluster*** screen, choose any of the cloud providers and region close to where you are, for instance ***AWS*** and ***North Virginia***, and then click ***Create Cluster***. In the ***Security Quickstart*** screen, choose ***Username and Password*** to create credentials to login to your database. In the ***Username*** and ***Password*** fields, type credentials you'll remember later since these are the credentials mongoose will use to login to the database from your Node.js server application when running on Heroku. If you forget these credentials you'll need to create new ones later. I went with ***giuseppi*** and ***supersecretpassword*** :) and clicked ***Create User***. Scroll a bit down and in the ***Where would you like to connect from*** section, select ***My Local Environment*** and then click ***Add My Current IP Address***. This will


Where would you like to connect from?

Enable access for any network(s) that need to read and write data to your cluster.

My Local Environment
Use this to add network IP addresses to the IP Access List. This can be modified at any time.

Cloud Environment
Use this to configure network access between Atlas and your cloud or on-premise environment. Specifically, set up IP Access Lists, Network Peering, and Private Endpoints.

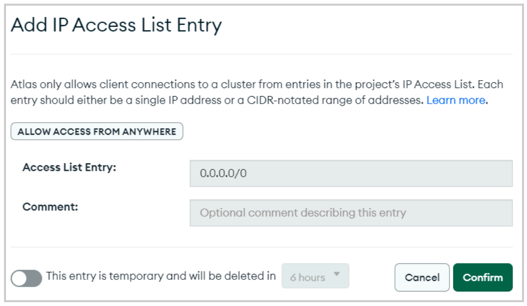
Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters. You can manage existing IP entries via the Network Access Page.

IP Address	Description
Enter IP Address	Enter description

IP Access List	Description
68.160.144.70/32	My IP Address

allow Node.js running on your development environment to connect to MongoDB running on Atlas. Click ***Finish and Close***. Then click on ***Go to Databases***. While in development, it can be convenient to allow connections from anywhere, so click on ***Network Access*** on the left and then ***ADD IP ADDRESS*** on the right. In the ***Add IP Access List Entry*** screen, click ***ALLOW ACCESS FROM ANYWHERE***. This will add 0.0.0.0/0 to the ***Access List Entry***, allowing any machine to connect. Click ***Confirm*** and verify the new entry appears in the ***Network Access*** screen. Click ***Database*** on the left to go back to the ***Database Deployments*** screen. To connect to the database, click ***Connect*** and in the ***Connect to Cluster 0*** screen, select ***Connect your application***. In the ***Select your driver and version*** section, confirm the ***Driver*** is set to ***Node.js*** and ***Version*** is set to ***4.1 or later***. In the ***Add your connection string into your application code*** section, copy the the URL. It should look similar to the following:


Add IP Access List Entry

Atlas only allows client connections to a cluster from entries in the project's IP Access List. Each entry should either be a single IP address or a CIDR-notated range of addresses. Learn more.

ALLOW ACCESS FROM ANYWHERE

Access List Entry: 0.0.0.0/0

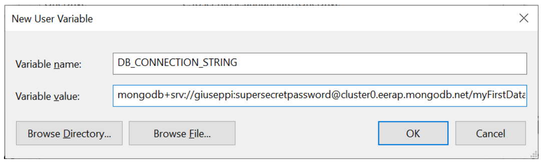
Comment: Optional comment describing this entry

This entry is temporary and will be deleted in 6 hours

Cancel Confirm

```
mongodb+srv://giuseppi:<password>@cluster0.eerap.mongodb.net/kanbas?retryWrites=true&w=majority
```

5.2 Integrating Node.js with MongoDB Atlas


New User Variable

Variable name: DB_CONNECTION_STRING

Variable value: mongodb+srv://giuseppi:supersecretpassword@cluster0.eerap.mongodb.net/myFirstData

Browse Directory... Browse File... OK Cancel

Now that we have a connection string, we can configure our Node.js server to use it to connect to the remote database server instead of the local one. In the ***App.js*** file, replace the connection string to use this new string. Better yet, declare an environment variable called ***DB_CONNECTION_STRING*** and set it's value to the URL connection string copied from Atlas. Make sure to replace ***<password>*** with the actual password created earlier in the ***Security Quickstart*** screen. Then in ***App.js*** use the code below to use the new environment variable. On macOS you can declare the ***DB_CONNECTION_STRING*** environment variable in ***~/.bash_profile***. It's a bad practice to keep credentials in source code. You might need to restart your machine for the environment variable to take effect.

```
App.js
```
```
const CONNECTION_STRING = process.env.DB_CONNECTION_STRING || 'mongodb://127.0.0.1:27017/kanbas
mongoose.connect(CONNECTION_STRING);
```

Start the server and the React application and confirm that the application is still working properly locally. When you deploy the Node.js application to ***Heroku***, make sure to create the ***DB_CONNECTION_STRING*** environment variable in the ***Heroku Settings*** tab pointing to the same Atlas MongoDB instance. Replace ***<password>*** with the actual password. Restart the React.js application in ***Netlify***, and restart the Node.js server in ***Heroku*** and confirm the two are able to interact properly.

5.3 Exercises

1. Create an account at MongoDB Atlas
2. Create a database in MongoDB Atlas
3. Refactor ***App.js*** to use the new database
4. Deploy the Node.js server to Heroku, and configure the server to connect to the remote database.
5. Deploy the React application to Netlify, and confirm it still works with the Node.js server running in Heroku

6 Deliverables

As a deliverable, make sure you complete all the exercises in this chapter. For both the React and Node repositories, all your work should be done in a branches called ***a6***. When done, add, commit and push both branches to their respective GitHub repositories. Deploy the new branches to Netlify and Heroku and confirm they integrate. Submit links to both your GitHub repositories as well as the Netlify and Heroku URLs where the branches deployed. Here's an example on the steps:

Create a branch called a6

```
git  checkout  -b  a6

# do all your work
```

Once you've completed all your work, add, commit and push your work to the remote repositories.

Add, commit and push the new branch

```
git  add  .
git  commit  -am  "a6 Mongo su23"
git  push
```

If you have ***Netlify*** configured to auto deploy, then confirm it auto deployed. If not, then deploy the branch manually.

In Canvas, submit the following

1. The new URLs where your ***a6*** branches deployed to on Netlify and Heroku
2. The link to your new branches in GitHub for the React and Node.js projects