**Storing Data in a MongoDB Database**

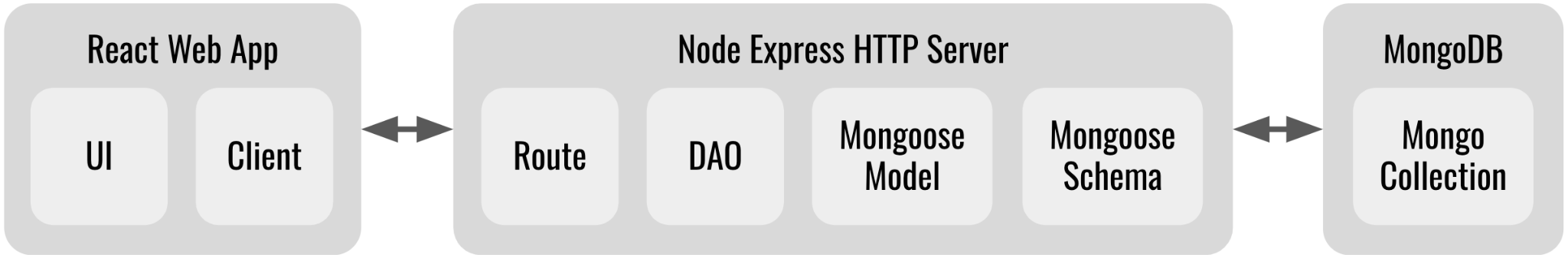
# 1 Introduction

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, or ***properties*** 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 user interface application to store the application state on the server. In this chapter we expand on 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.

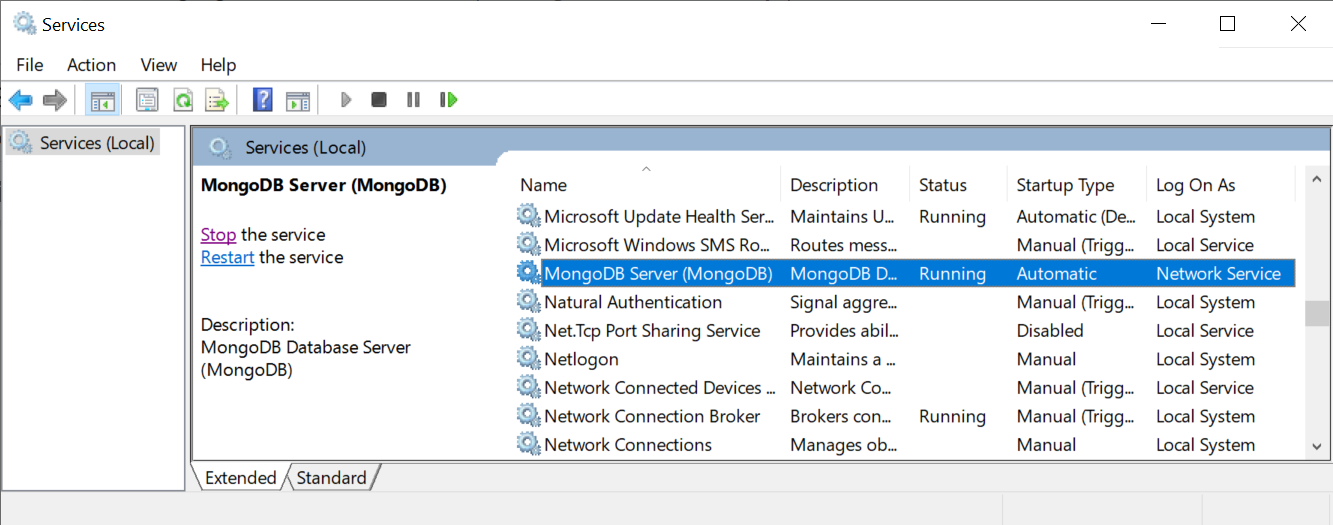
The following figure illustrates the overall architecture of what we'll be building in this chapter. From right to left, we'll first create a MongoDB database called ***kanbas*** where we'll create several ***collections*** such as ***users***, ***courses***, ***modules***, ***assignments***, etc. We'll 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.

## 2.1 Installing and Configuring MongoDB

To get started, [download MongoDB for free](https://www.mongodb.com/try/download/community) 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.

### 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 from the downloaded archive 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 creates 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***.

### 2.1.2 Starting MongoDB from the Command Line

If you installed ***MongoDB*** as a service, it is already running in the background and can be configured and restarted in ***Windows*** from the ***Services*** dialog or from the ***System Settings*** on ***macOS***. Alternatively you can start the MongoDB server from the command line using the ***mongod*** executable in the ***bin*** directory where you installed ***MongodDB***. First you'll need to create a ***data*** folder where the server will store all its data. You can create a data folder in your home directory as shown below.

| **cd ~**  **mkdir data** |
| --- |

When you start ***MongoDB***, you'll need to tell it where the ***data*** folder is with the ***dbpath*** option. If you installed MongoDB on ***Windows*** in ***C:\Program Files\mongodb-macos-x86\_64-5.0.3***, you can start MongoDB from your home directory as shown below.

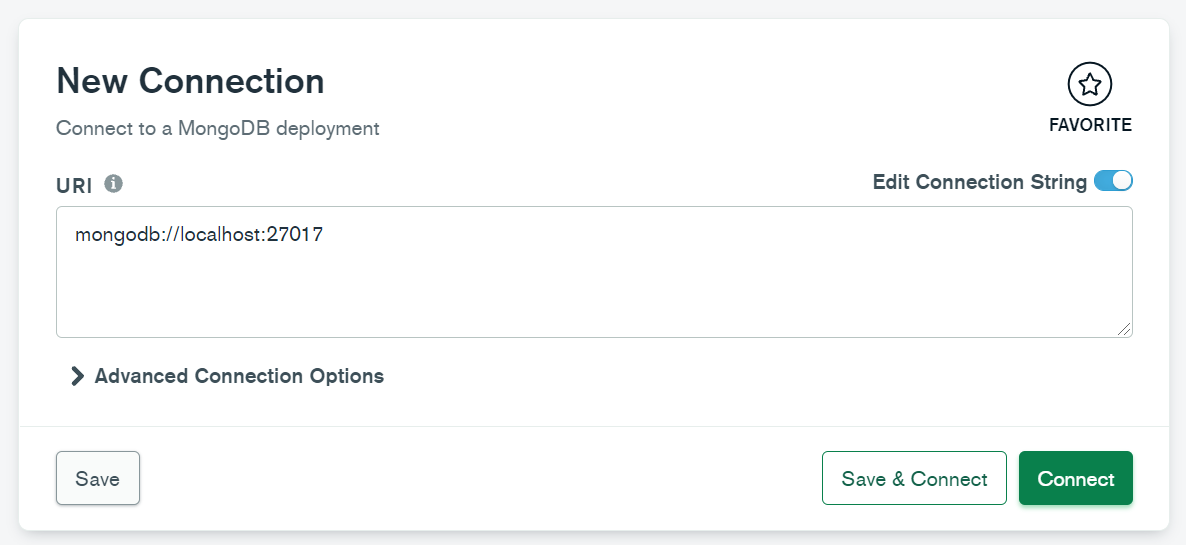
| **cd ~**  **C:\Program Files\mongodb-macos-x86\_64-5.0.3\bin\mongod --dbpath data** |
| --- |

Make sure to include the ***dbpath data*** option to tell ***MongoDB*** where to find the ***data*** directory.

If you installed ***MongoDB*** on ***macOS*** in ***/usr/local/mongodb-macos-x86\_64-6.0.5***, you can start ***MongoDB*** from your home directory as shown below.

| **cd ~**  **/usr/local/mongodb-macos-x86\_64-6.0.5/bin/mongod --dbpath data** |
| --- |

## 2.2 Using MongoDB Compass to Interact with MongoDB

Your installation should have installed ***MongoDB Compass***, a user interface client to the MongoDB database. If not, [***MongoDB Compass*** can be downloaded from MongoDB's download page](https://www.mongodb.com/try/download/compass). You can start ***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.

## 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.

|  |  |
| --- | --- |

## 2.4 Inserting and Retrieving Data with Compass

In MongoDB, data is organized into ***collections***, which are analogous to ***tables*** in relational databases. Data contained in collections are referred to as ***documents***, which are analogous to ***records*** in relational databases. To create, or ***insert*** documents into a collection in a MongoDB database using Compass, 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 ***users*** collection as shown below on the right. On the right side, selected ***Add Data*** and then ***Insert Document***. In the ***Insert to Collection kanbas.users*** 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***](https://gist.github.com/jannunzi/261af8e7c1d39301861e73c8e9dc8210) file we used in earlier assignments under the ***Database*** directory of your React 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](https://gist.github.com/jannunzi/5ac95f04a645311338a7f3672cb44bf1), [assignments.json](https://gist.github.com/jannunzi/a417dd6b623db7de3f12e0b7251a40de), [users.json](https://gist.github.com/jannunzi/faf916933614023d043c9ad7b8e62780)

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 in this document.

## 2.5 Interacting with a MongoDB Database with the 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  } |
| --- |

# 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***](https://mongoosejs.com/). 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***](https://mongoosejs.com/) 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 a later 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";  ...  constCONNECTION\_STRING = **"mongodb://127.0.0.1:27017/kanbas"**  mongoose.connect(CONNECTION\_STRING);  const app = express();  ... | *// load the mongoose library*  *// connect to the* ***kanbas*** *database* |

## 3.2 Configuring Connection Strings as Environment Variables

Instead of hard coding the ***connection string*** in the source code, it's better to configure it as an environment variable and then reference it from the code. This will come in handy when the server application is deployed to a remote service such as ***Render*** or ***Heroku*** and the ***connection string*** can be configure to reference the online remote database running on Atlas cloud servive. In a new ***.env*** file, declare the following connection string environment variable.

| ***.env*** |
| --- |
| **MONGO\_CONNECTION\_STRING=mongodb://127.0.0.1:27017/kanbas** |

Install the ***dotenv*** library to read configurations in the local environment.

| **$ npm install dotenv** |
| --- |

Then in ***App.js***, import the ***dotenv*** library to read the ***connection string*** as shown below.

| ***App.js*** |  |
| --- | --- |
| import "dotenv/config";  import express from "express";  import mongoose from "mongoose";  ...  constCONNECTION\_STRING = ***process***.**env**.**MONGO\_CONNECTION\_STRING** || **"mongodb://127.0.0.1:27017/kanbas"**  mongoose.connect(CONNECTION\_STRING);  const app = express();  app.use(cors());  app.use(express.json());  ...  app.listen(process.env.PORT || 4000); | |

## 3.3 Implementing Mongoose Schemas and Models

Now that we have the collections setup in our database, let's now discuss how to connect and interact with 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 like relational databases do. That responsibility has been delegated to the applications using 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 or modified through the Mongoose library. The ***schema*** shown below describes the structure for the ***users*** collection imported earlier. Create the schema in a ***Users*** directory in your Node.js projects.

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

## 3.4 Implementing Mongoose Models

In earlier sections we demonstrated using the command line client to interact manually with the MongoDB server using the ***find*** command. Mongoose ***models*** provide similar functionality 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";  import schema from "./schema.js";  const model = mongoose.model("UserModel", schema);  export default model; | *// load mongoose library*  *// load users schema*  *// create mongoose model from the schema*  *// export so it can be used elsewhere* |

## 3.5 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) => {} // implemented later  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 }); | |

## 3.6 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);  } | | | |

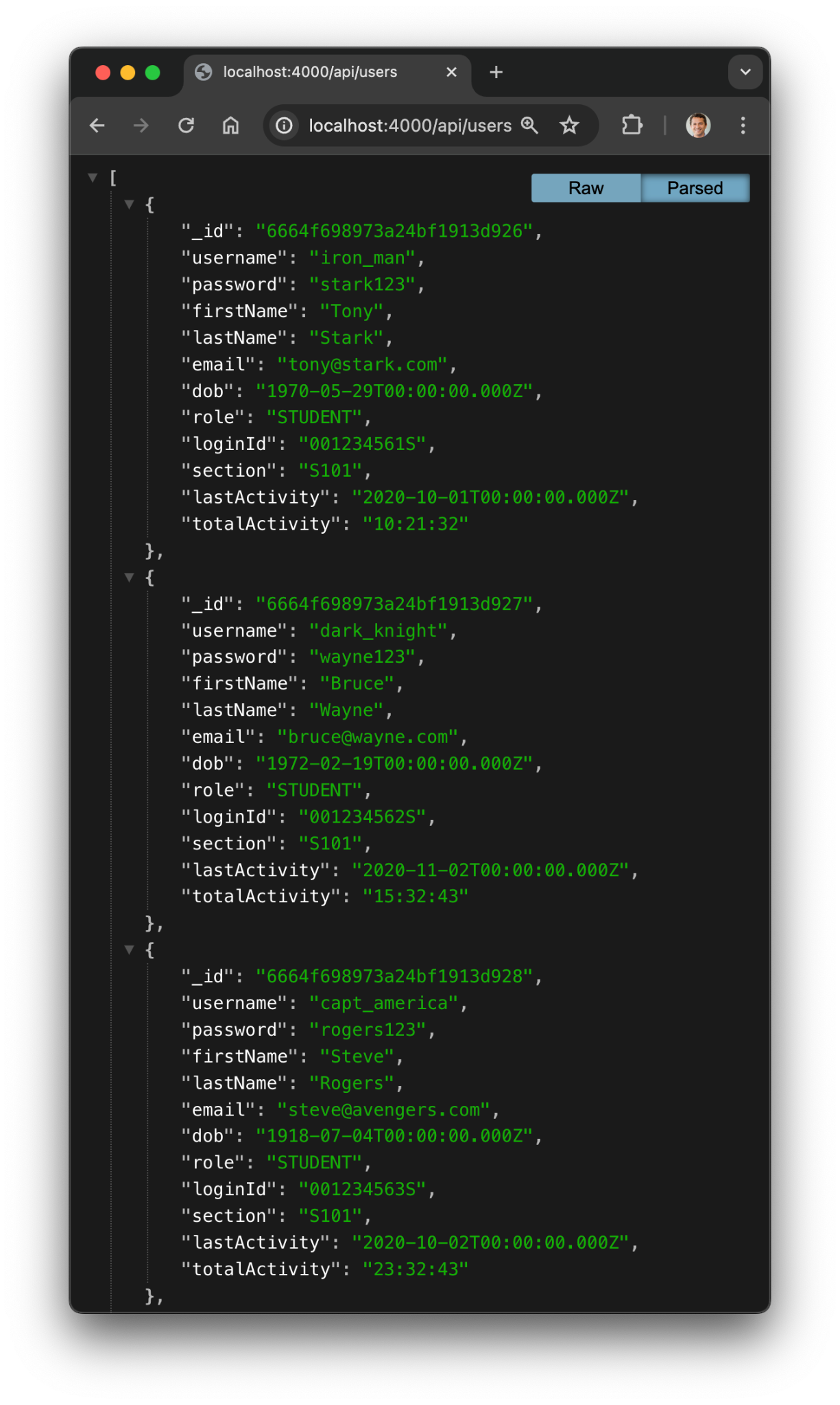
Import and configure the routes in ***App.js*** as shown below.

| ***App.js*** | | |  |
| --- | --- | --- | --- |
| import "dotenv/config";  import express from "express";  import cors from "cors";  ...  import mongoose from "mongoose";  import UserRoutes from "./Users/routes.js";  **const** CONNECTION\_STRING = ***process***.**env**.**MONGO\_CONNECTION\_STRING** || **'mongodb://127.0.0.1:27017/kanbas';**  mongoose.connect(CONNECTION\_STRING);  const app = express();  app.use(cors());  app.use(express.json());  UserRoutes(app);  ...  app.listen(process.env.PORT || 4000); | | | |

### 3.6.1 Retrieving All Documents from MongoDB with Mongoose

***DAO***s implement high level data operations based on lower level ***Mongoose*** models. The ***Mongoose*** model ***find*** function retrieves a all documents from a collection. The ***findAllUsers*** function below uses ***find*** to retrieve all the users from the users collection.

| ***Users/dao.js*** | |
| --- | --- |
| import model from "./model.js";  export const findAllUsers = () => model.find(); | |

***Routes*** implement RESTful Web APIs that user interface clients can use to interact with server functionality. The route implemented below uses the ***findAll*** function implemented by the DAO to retrieve all the users from the database. The route responds with the collection of users retrieved from the database. Confirm the route works by navigating to [***http://localhost:4000/api/users***](http://localhost:4000/api/users) with your browser.

| ***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);  } | | | |

Meanwhile in the React user interface application, under ***src/Courses/People***, implement the ***client*** shown below to interact with the user routes implemented in the server. The client function ***findAllUsers*** shown below sends a GET request to the server and ***await***s the server's ***response*** containing an array of users in the ***data*** property.

| ***src/Courses/People/client.ts*** | | |  |
| --- | --- | --- | --- |
| import axios from "axios";  export const REMOTE\_SERVER = process.env.REACT\_APP\_REMOTE\_SERVER;  export const USERS\_API = `${REMOTE\_SERVER}/api/users`;  export const findAllUsers = async () => {  const response = await axios.get(USERS\_API);  return response.data;  }; | | | |

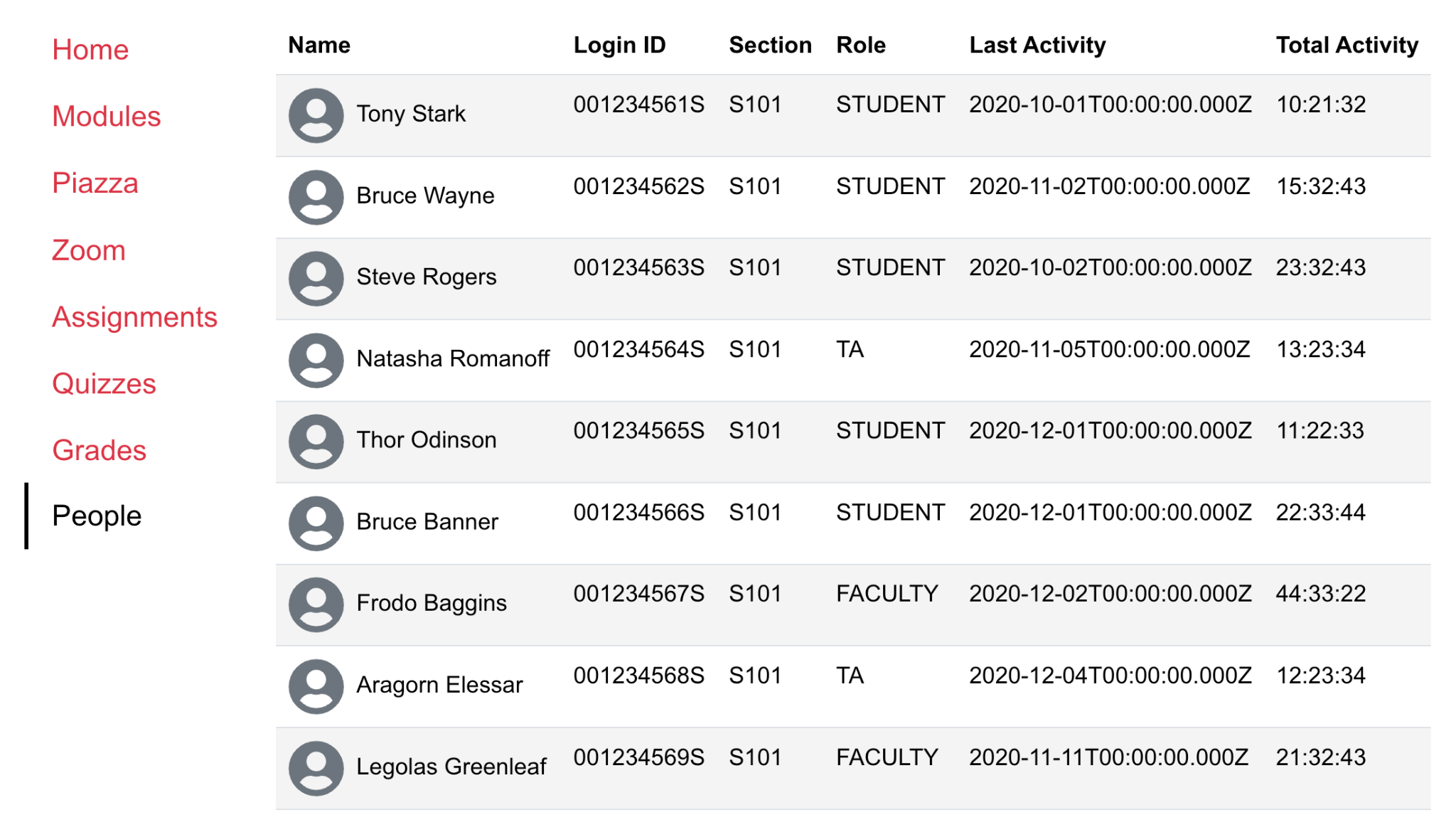
To display the array of users from the database, implement a ***People*** screen that retrieves all the users from the database and renders them in a ***table*** as shown below.

| ***src/Courses/People/Table.tsx*** | | |  |
| --- | --- | --- | --- |
| import React, { useState, useEffect } from "react";  import \* as client from "./client";  export default function PeopleTable() {  const [users, setUsers] = useState<any[]>([]);  const fetchUsers = async () => {  const users = await client.findAllUsers();  setUsers(users);  };  useEffect(() => {  fetchUsers();  }, []);  return (  <div id="wd-people-table">  <table className="table table-striped">  <thead>  <tr>  <th>Name</th><th>Login ID</th><th>Section</th><th>Role</th><th>Last Activity</th><th>Total Activity</th>  </tr>  </thead>  <tbody>  {users.map((user: any) => (  <tr key={user.\_id}>  <td className="wd-full-name text-nowrap">  <span className="wd-first-name">{user.firstName}</span>  <span className="wd-last-name">{user.lastName}</span>  </td>  <td className="wd-login-id">{user.loginId}</td>  <td className="wd-section">{user.section}</td>  <td className="wd-role">{user.role}</td>  <td className="wd-last-activity">{user.lastActivity}</td>  <td className="wd-total-activity">{user.totalActivity}</td>  </tr>  ))}  </tbody>  </table>  </div>  );  } | | | |

Add a ***People*** link to the ***Courses Navigation*** sidebar that navigates to the ***PeopleTable*** component above mapped to ***/Kanbas/Courses/${cid}/People***, where ***cid*** is the ID of the current course. In the ***Courses*** component, add a new route that displays the new ***PeopleTable*** mapped to the path ***People*** as shown below.

| ***src/Kanbas/Courses/index.tsx*** | |
| --- | --- |
| <Routes>  <Route path="/" element={<Navigate to="Home" />} />  <Route path="Home" element={<Home />} />  <Route path="Modules" element={<Modules />} />  <Route path="Assignments" element={<Assignments />} />  <Route path="Assignments/:aid" element={<AssignmentEditor />} />  <Route path="People" element={<PeopleTable />} />  </Routes> | |

Navigate to a course from the ***Dashboard*** and then to the ***People*** screen and confirm that it renders as shown below.



### 3.6.2 Retrieving Documents by Predicate from MongoDB with Mongoose

In the ***User***'s ***DAO***, implement ***findUsersByRole*** that filters the ***users*** collection by the ***role*** property as shown below. Mongoose model's ***find*** function takes as argument a JSON object used to pattern match documents in the collection. The ***{role: role}*** object means that documents will be filtered by their ***role*** property that matches the value ***role***.

| ***Users/dao.js*** | | |  |
| --- | --- | --- | --- |
| export const findUsersByRole = (role) => model.find({ role: role }); *// or just model.find({ role })* | | | |

In the ***User*** ***routes***, refactor the ***findAllUsers*** function so that it parses the ***role*** from the query string, and then uses 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 React user interface application, add ***findUserByRole*** in the client so that it encodes the ***role*** in the query string of the URL as shown below.

| ***src/Kanbas/Courses/People/client.ts*** | | |  |
| --- | --- | --- | --- |
| export const findUsersByRole = async (role: string) => {  const response = await  axios.get(`${USERS\_API}?role=${role}`);  return response.data;  }; | | | |

In the ***PeopleTable*** component, add a dropdown that invokes a ***filterUsers*** event handler function with the selected ***role***. The function updates a ***role*** state variable and requests from the server the list of users filtered by their role. Confirm that selecting various roles actually filters the users by their role.

| ***src/Kanbas/Courses/People.tsx*** | |
| --- | --- |
| export default function PeopleTable() {  const { cid } = useParams();  const [users, setUsers] = useState<any[]>([]);  const [role, setRole] = useState("");  const filterUsersByRole = async (role: string) => {  setRole(role);  if (role) {  const users = await client.findUsersByRole(role);  setUsers(users);  } else {  fetchUsers();  }  };  const fetchUsers = async () => { ... };  return (  <div id="wd-people-table">  <select value={role} onChange={(e) =>filterUsersByRole(e.target.value)}  className="form-select float-start w-25 wd-select-role" >  <option value="">All Roles</option> <option value="STUDENT">Students</option>  <option value="TA">Assistants</option> <option value="FACULTY">Faculty</option>  </select>  ...  </div>  );  } | |

Now practice filtering users by their ***first*** or ***lastName*** by creating a ***regular expression*** used to pattern match the ***firstName*** or ***lastName*** fields of the documents in the ***users*** collection.

| ***Users/dao.js*** | |
| --- | --- |
| export const findUsersByPartialName = (partialName) => {  const regex = new RegExp(partialName, "i"); // 'i' makes it case-insensitive  return model.find({  $or: [{ firstName: { $regex: regex } }, { lastName: { $regex: regex } }],  });  }; | |

In the the ***User routes***, parse a ***name*** parameter from the ***query string*** and use it to find users whose first or last names partially match the ***name*** parameter.

| ***Users/routes.js*** | | |
| --- | --- | --- |
| const findAllUsers = async (req, res) => {  const { role, name } = req.query;  if (role) { ... }  if (name) {  const users = await dao.findUsersByPartialName(name);  res.json(users);  return;  }  const users = await dao.findAllUsers();  res.json(users);  }; | | |

In the React client application, implement the ***findUserByPartialName*** client function as shown below which encodes a ***name*** in the ***query string*** the server can use to filter users by their ***first*** and ***lastName***.

| ***src/Kanbas/Courses/People/client.ts*** | |
| --- | --- |
| export const findUsersByPartialName = async (name: string) => {  const response = await axios.get(`${USERS\_API}?name=${name}`);  return response.data;  }; | |

In the ***PeopleTable*** component, create a new ***name*** state variable and corresponding ***input*** field used to invoke the ***findUsersByPartialName*** client function and update the ***users*** state variable with a subset of users that match the ***name***. Confirm that typing a name in the input field actually filters the users by their first or last name. Note that the current implementation does not consider a combination of filtering by ***role*** and by ***name***. Feel free to explore how you would go about filtering by both.

| ***src/Kanbas/Courses/People/Table.tsx*** | |
| --- | --- |
| export default function PeopleTable() {  const { cid } = useParams();  const [users, setUsers] = useState<any[]>([]);  const [role, setRole] = useState("");  const [name, setName] = useState("");  const filterUsersByName = async (name: string) => {  setName(name);  if (name) {  const users = await client.findUsersByPartialName(name);  setUsers(users);  } else {  fetchUsers();  }  };  ...  return (  <div id="wd-people-table">  <input onChange={(e) => filterUsersByName(e.target.value)} placeholder="Search people"  className="form-control float-start w-25 me-2 wd-filter-by-name" />  ...  </div>  );  } | |

### 3.6.3 Retrieving Documents by Primary Key from MongoDB with Mongoose

A 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/Kanbas/Courses/People/client.ts*** | | |  |
| --- | --- | --- | --- |
| export const findUserById = async (id: string) => {  const response = await axios.get(`${USERS\_API}/${id}`);  return response.data;  }; | | | |

In a new ***PeopleDetails*** component, use the client's ***findUserById*** function to retrieve the user when a faculty clicks on the user's name. Parse a ***uid*** path parameter and use it to retrieve the user by their ID when the component loads. If the ***uid*** does not exist, return ***null*** so that the component does not render on the screen. In ***useEffect***, add ***uid*** as a dependency so that if the component re-renders if you click on another user while the component is still displaying.

| ***src/Kanbas/Courses/People/Details.tsx*** | |
| --- | --- |
| import { useEffect, useState } from "react";  import { FaUserCircle } from "react-icons/fa";  import { IoCloseSharp } from "react-icons/io5";  import { useParams } from "react-router";  import { Link } from "react-router-dom";  import \* as client from "./client";  export default function PeopleDetails() {  const { uid, cid } = useParams();  const [user, setUser] = useState<any>({});  const fetchUser = async () => {  if (!uid) return;  const user = await client.findUserById(uid);  setUser(user);  };  useEffect(() => {  if (uid) fetchUser();  }, [uid]);  if (!uid) return null;  return (  <div className="wd-people-details position-fixed top-0 end-0 bottom-0 bg-white p-4 shadow w-25">  <Link to={`/Kanbas/Courses/${cid}/People`} className="btn position-fixed end-0 top-0 wd-close-details">  <IoCloseSharp className="fs-1" /> </Link>  <div className="text-center mt-2"> <FaUserCircle className="text-secondary me-2 fs-1" /> </div><hr />  <div className="text-danger fs-4 wd-name"> {user.firstName} {user.lastName} </div>  <b>Roles:</b> <span className="wd-roles"> {user.role} </span> <br />  <b>Login ID:</b> <span className="wd-login-id"> {user.loginId} </span> <br />  <b>Section:</b> <span className="wd-section"> {user.section} </span> <br />  <b>Total Activity:</b> <span className="wd-total-activity">{user.totalActivity}</span> </div> ); } | |

Add a ***close button*** rendered as an ***X*** at the top left that hides the component by navigating back to ***PeopleTable*** screen as shown above. Confirm that clicking on the the name of a user displays the user's details. Also confirm that closing the ***PeopleDetails*** hides the component. In the ***Courses*** component, add a route that encodes the ***uid*** path parameter in the ***URL*** that renders the same ***PeopleTable*** component.

| ***src/Kanbas/Courses/index.tsx*** | |
| --- | --- |
| <Routes>  <Route path="/" element={<Navigate to="Home" />} />  <Route path="Home" element={<Home />} />  <Route path="Modules" element={<Modules />} />  <Route path="Assignments" element={<Assignments />} />  <Route path="Assignments/:aid" element={<AssignmentEditor />} />  <Route path="People" element={<PeopleTable />} />  <Route path="People/:uid" element={<PeopleTable />} />  </Routes> | |

### 3.6.4 Deleting a Document in MongoDB with Mongoose

To delete user documents from the ***users*** MongoDB collection, implement the ***deleteUser*** operation as shown below. The ***DAO*** function below removes a single ***user*** document from the database based on its primary key.

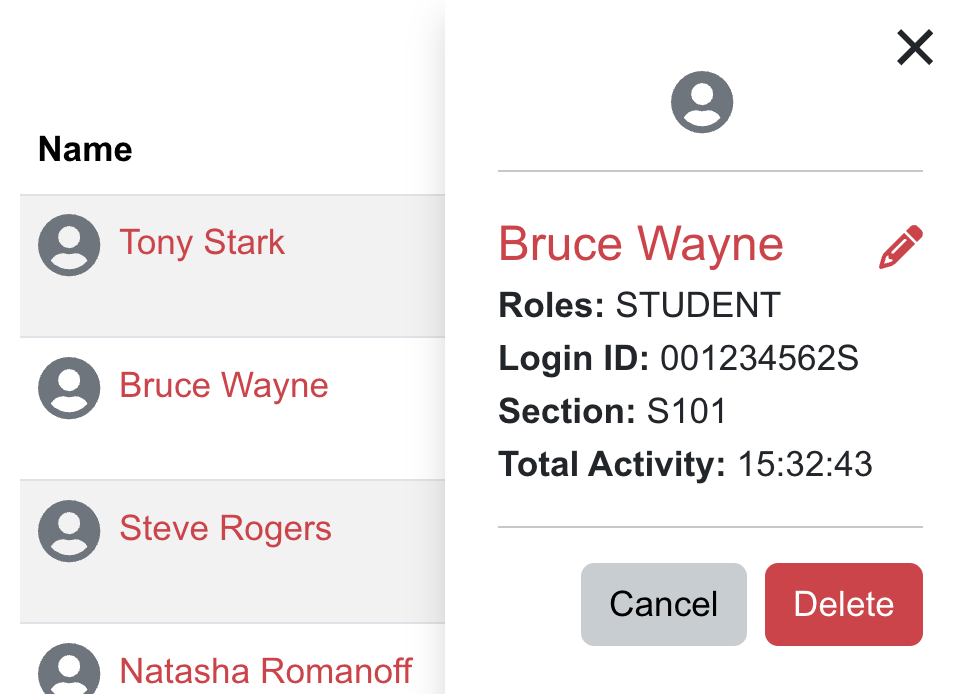
| ***Users/dao.js*** | | |  |
| --- | --- | --- | --- |
| export const deleteUser = (userId) => model.deleteOne({ \_id: userId }); | | |  |

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 (userId: string) => {  const response = await axios.delete( `${USERS\_API}/${userId}` );  return response.data;  }; | | | |

In the ***PeopleDetails*** component add buttons ***Cancel*** and ***Delete*** as shown below. The ***Delete*** button invokes a new ***deleteUser*** event handler function with ***uid***, the ID of the user to delete. Pass a reference to ***fetchUsers*** as a parameter so that ***PeopleDetails*** can notify ***PeopleTable*** that a user has been remove and that the list of users must be updated.

| ***src/Kanbas/Courses/People/Details.tsx*** | |
| --- | --- |
| ...  import { useNavigate, useParams } from "react-router";  export default function PeopleDetails({ fetchUsers }:  { fetchUsers: () => void; }) {  const navigate = useNavigate();  const deleteUser = async (uid: string) => {  await client.deleteUser(uid);  fetchUsers();  navigate(`/Kanbas/Courses/${cid}/People`);  };  ...  return (  <div className="...">  ...  <hr />  <button onClick={() => deleteUser(uid)} className="btn btn-danger float-end wd-delete" > Delete </button>  <button onClick={() => navigate(`/Kanbas/Courses/${cid}/People`)}  className="btn btn-secondary float-start float-end me-2 wd-cancel" > Cancel </button>  </div>  );  } | |

Use the ***client***'s ***deleteUser*** to remove the user, notify the ***PeopleTable*** to refresh the users, and navigate to the ***PeopleTable*** to hide the details. The ***Cancel*** button just hides the details without removing any documents. In the ***PeopleTable*** component, pass a reference of ***fetchUsers*** to ***PeopleDetails*** so that it can notify if the table needs to be refreshed. Confirm that clicking the ***Cancel*** and ***Delete*** buttons actually work.

| ***src/Kanbas/Courses/People/Table.tsx*** | |
| --- | --- |
| ...  <PeopleDetails fetchUsers={fetchUsers} />  ... | |

### 3.6.5 Updating a Document in MongoDB with Mongoose

The ***Mongoose*** ***update*** function updates documents in ***MongoDB*** databases. In the ***User***'s ***DAO***, implement ***updateUser*** as shown below to update 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 }); | | | |

In the ***User***'s ***routes***, make the ***DAO*** function available as a RESTful Web API as shown below. 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 responds 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);  res.json(status);  };  ...  app.put("/api/users/:userId", updateUser);  } | | | |

In the React client application, add client function ***updateUser*** to send user updates to the server to be saved to the database.

| ***src/Kanbas/Courses/People/client.ts*** | | |  |
| --- | --- | --- | --- |
| export const updateUser = async (user: any) => {  const response = await axios.put(`${USERS\_API}/${user.\_id}`, user);  return response.data;  }; | | | |

In the ***PeopleDetails*** component, add a ***name*** state variable to edit the ***first*** and ***last*** name of the user. Also add a ***editing*** state variable to toggle the input field that edits the ***name***. With the ***navigate*** routing function, navigate back to ***PeopleTable*** once the update is done. Create a new ***saveUser*** function that splits the ***name*** state variable into the ***firstName*** and ***lastName*** and sends an updated version of the user to the server. Also update the local ***user*** state variable, turn off editing, notify the ***PeopleTable*** to fetch the users again, and ***navigate*** to the ***PeopleTable***.

| ***src/Kanbas/Courses/People/Details.tsx*** | |
| --- | --- |
| ...  import { FaPencil } from "react-icons/fa6";  import { FaCheck, FaUserCircle } from "react-icons/fa";  export default function PeopleDetails( ... ) {  ...  const [name, setName] = useState("");  const [editing, setEditing] = useState(false);  const saveUser = async () => {  const [firstName, lastName] = name.split(" ");  const updatedUser = { ...user, firstName, lastName };  await client.updateUser(updatedUser);  setUser(updatedUser);  setEditing(false);  fetchUsers();  navigate(`/Kanbas/Courses/${cid}/People`);  };  ...  return (  ...  <div className="text-danger fs-4">  {!editing && (  <FaPencil onClick={() => setEditing(true)}  className="float-end fs-5 mt-2 wd-edit" /> )}  {editing && (  <FaCheck onClick={() => saveUser()}  className="float-end fs-5 mt-2 me-2 wd-save" /> )}  {!editing && (  <div className="wd-name"  onClick={() => setEditing(true)}>  {user.firstName} {user.lastName}</div>)}  {user && editing && (  <input className="form-control w-50 wd-edit-name"  defaultValue={`${user.firstName} ${user.lastName}`}  onChange={(e) => setName(e.target.value)}  onKeyDown={(e) => {  if (e.key === "Enter") { saveUser(); }}}  />  )}  </div>  ...  );} | *// to edit the user's first and last name*  *// whether we are editing or not*  *// to save updates to user's name*  *// split the name into an array and get first*  *// and last and create new version of user overriding*  *// first and last. Send update to server*  *// update local copy of the user*  *// we're done editing*  *// tell PeopleTable to fetch users again*  *// go back to PeopleTable*  *// if not editing show pencil icon*  *// clicking pencil turns on editing and hides pencil*  *// if editing show check mark. Clicking check turns*  *// off editing, saves and hides check*  *// if not editing show first and last name*  *// clicking on name turns on editing*  *// if editing show input field to edit name*  *// name is initially concatenation of first and last*  *// update name as we type*  *// save the user if Enter key is pressed* |

Add ***pencil*** and ***check*** icons to turn ***editing*** on and off. Hide each icon based on the ***editing*** boolean state variable. Clicking the name of the user also turns ***editing*** on. If ***editing*** is on, hide the user's name and instead display an input field that shows the current user's ***firstName*** and ***lastName*** and edits the ***name*** state variable. Pressing the ***Enter*** key saves the updated user's details and notifies the ***PeopleTable*** component to fetch the users again and update the table. Confirm users can be edited. On your own, add the ability to edit a user's email and role. Use an input field of type ***email*** to edit the email. To edit the user's role, use a dropdown similar to the one used to filter users by their role.

### 3.6.6 Creating New Documents in MongoDB with Mongoose

In the ***User***'s ***DAO***, implement the ***createUser*** function as shown below to insert 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* |

Make sure the incoming ***user*** object does not have an ***\_id*** property since it can interfere with the database insert operation. In the ***User***'s ***routes***, make 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);  ...  } | | | |

In the React client application, implement a ***createUser*** client function to interact with the route created above. Post the new user object to the server as shown below.

| ***src/Kanbas/Courses/People/client.ts*** | | |  |
| --- | --- | --- | --- |
| export const createUser = async (user: any) => {  const response = await axios.post(`${USERS\_API}`, user);  return response.data;  }; | | | |

In the ***PeopleTable*** component, implement a new ***createUser*** event handler that sends a new user object to be inserted in the database. Use default values for the fields as shown and confirm that clicking the new ***+ People*** button actually creates the new user. Optionally implement editing those fields in the ***PeopleDetails*** component.

| ***src/Kanbas/Courses/People/Table.tsx*** | |
| --- | --- |
| export default function PeopleTable() {  const { cid } = useParams();  const [users, setUsers] = useState<any[]>([]);  const [role, setRole] = useState("");  const [name, setName] = useState("");  const createUser = async () => {  const user = await client.createUser({  firstName: "New",  lastName: `User${users.length + 1}`,  username: `newuser${Date.now()}`,  password: "password123",  section: "S101",  role: "STUDENT",  });  setUsers([...users, user]);  };  return (  <div id="wd-people-table">  <button onClick={createUser} className="float-end btn btn-danger wd-add-people">  <FaPlus className="me-2" />  People  </button>  ...  </div>  );  } | |

# 4 Integrating with MongoDB Hosted in Atlas Cloud Service

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 ***Render***, ***Heroku*** or ***AWS***, the server needs to connect to a database that is also hosted on a public site. ***MongoDB Atlas Cloud Service*** 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.

## 4.1 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 sign up process. In the ***Deploy your cluster*** screen choose a ***Free*** plan for now which should be enough for this course. Name your cluster ***Kanbas***. In the ***Provider*** section, choose any of the cloud providers and in the ***Region*** section choose a region close to your geographic area, for instance ***AWS*** and ***North Virginia***, and then click ***Create Deployment***. In the ***Connect to Kabas*** screen, in the ***Create a database user*** section 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 Render or Heroku. If you forget these credentials you'll need to create new ones later. I went with ***giuseppi*** and ***supersecretpassword*** :) and clicked ***Create Database User***.

### 4.1.1 Connecting to a Remote Database from Compass

Then click the ***Choose a connection method*** button, and in the ***Access your data through tools***, select ***Compass***. In the ***Connecting with MongoDB Compass*** screen, in the ***Copy the connection string, then open MongoDB Compass*** section, copy the connection string which should look something like so

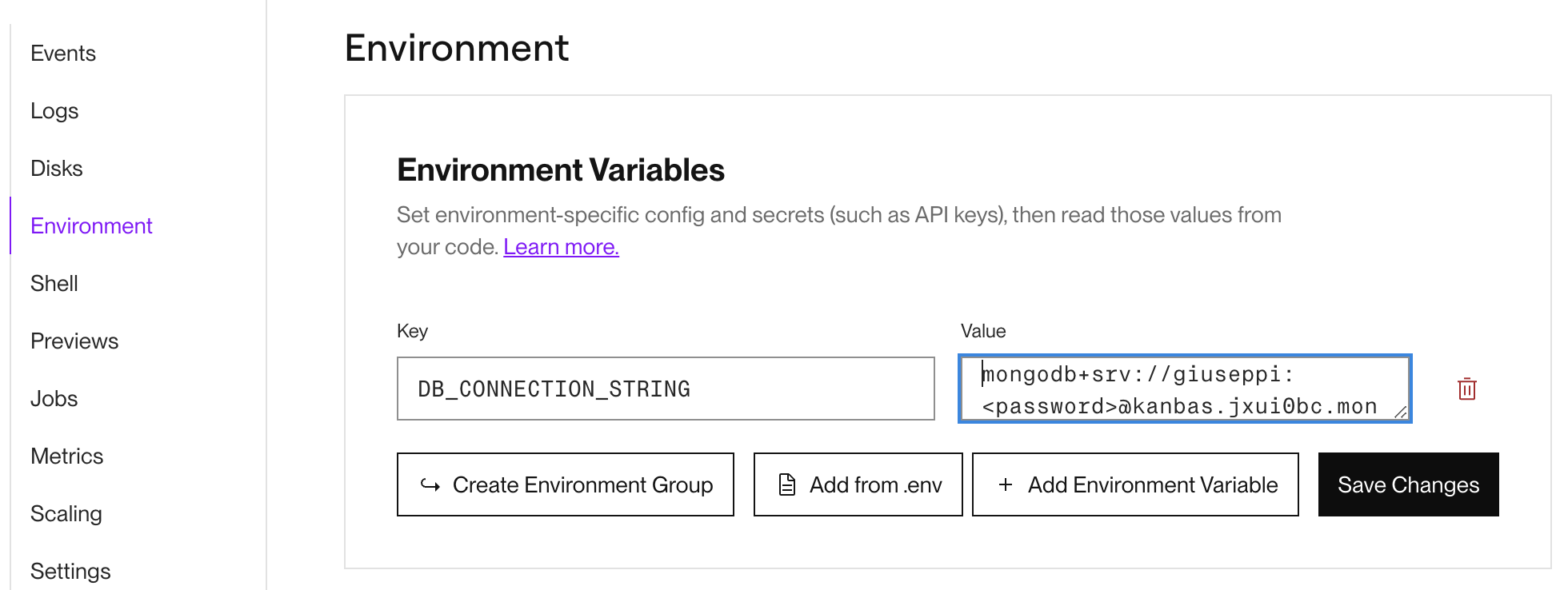
| **mongodb+srv://giuseppi:supersecretpassword@kanbas.jxui0bc.mongodb.net/** |
| --- |

Click ***Done***. From ***Compass*** select ***Connect***, ***New Window***, paste the connection string in the ***URI*** field, and then click connect. Following the same steps used earlier, import the ***courses***, ***modules***, and ***users*** into the remote database.

### 4.1.2 Connecting from Node.js

In the ***Atlas*** main window click on ***Network Access*** and in the ***Network Access*** screen ***IP Access List*** tab, select ***+ ADD IP ADDRESS***. In the ***Add IP Access List Entry*** dialog click on ***ALLOW ACCESS FROM ANYWHERE***. This will add ***0.0.0.0/0*** to the ***Access List Entry***, allowing any computer to connect. Click ***Confirm*** and verify the new entry appears in the ***Network Access*** screen. Click ***Database*** on the left and in the ***Clusters*** screen, click ***Connect***. In the ***Connect to Kanbas*** dialog, in the ***Connect to your application***, select ***Drivers and version*** section, confirm the ***Driver*** is set to ***Node.js*** and ***Version*** is set to ***5.5 or later***. In the ***Add your connection string into your application code*** section, copy the the URL. It should look similar to the following.

| ***Node MongoDB Database Connection String*** |
| --- |
| **mongodb+srv://giuseppi:<password>@kanbas.jxui0bc.mongodb.net/kanbas?retryWrites=true&w=majority&appName=Kanbas** |

Commit and push your code to a branch called ***a6*** and deploy the server application to a new remote service running on ***Render***, ***Heroku***, or ***AWS***. Make sure not to deploy to the server for prior assignments since TAs might still be grading it. In the new remote server, configure an environment variable called ***MONGO\_CONNECTION \_STRING*** with the URL value above. For instance, in the ***Render*** dashboard click on ***Environment***, type ***MONGO\_CONNECTION\_STRING*** in the ***Key*** field and the URL in the ***Value*** field. Replace the ***<password>*** with the actual password created in an earlier step. Commit and deploy the React application to a new ***a6*** branch. Configure the ***REACT\_APP\_REMOTE\_SERVER*** to point to the new remote server. For the environment variables to take effect, you might need to redeploy and/or restart the remote Node server on Render as well as the remote React application server on Netlify.

# 5 Implementing User Accounts

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. This section implements user authentication operations such as sign in, sign up, and multiple users support. Do your work in ***Google Chrome***.

## 5.1 Implementing a Sign In Screen

Signing into an application usually consists of providing credentials such as a ***username*** and ***password***. 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 of the same name passed from the user interface through the RESTful API.

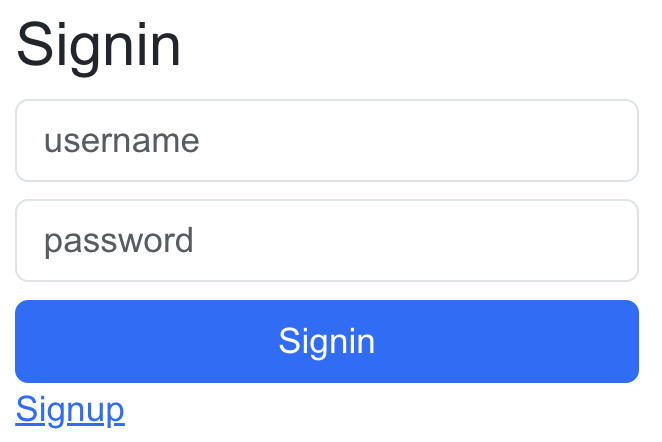
| ***Users/dao.js*** |  |
| --- | --- |
| import model from "./model.js";  export const findUserByCredentials = (username, password) => model.findOne({ username, password }); | |

***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 will 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);  } | | | |

In the React user interface, under ***Kanbas/Account***, 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/Kanbas/Account/client.ts*** | | |  |
| --- | --- | --- | --- |
| import axios from "axios";  export const REMOTE\_SERVER = process.env.REACT\_APP\_REMOTE\_SERVER;  export const USERS\_API = `${REMOTE\_SERVER}/api/users`;  export const signin = async (credentials: any) => {  const response = await axios.post( `${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. Style the component so it looks as shown below on the right.

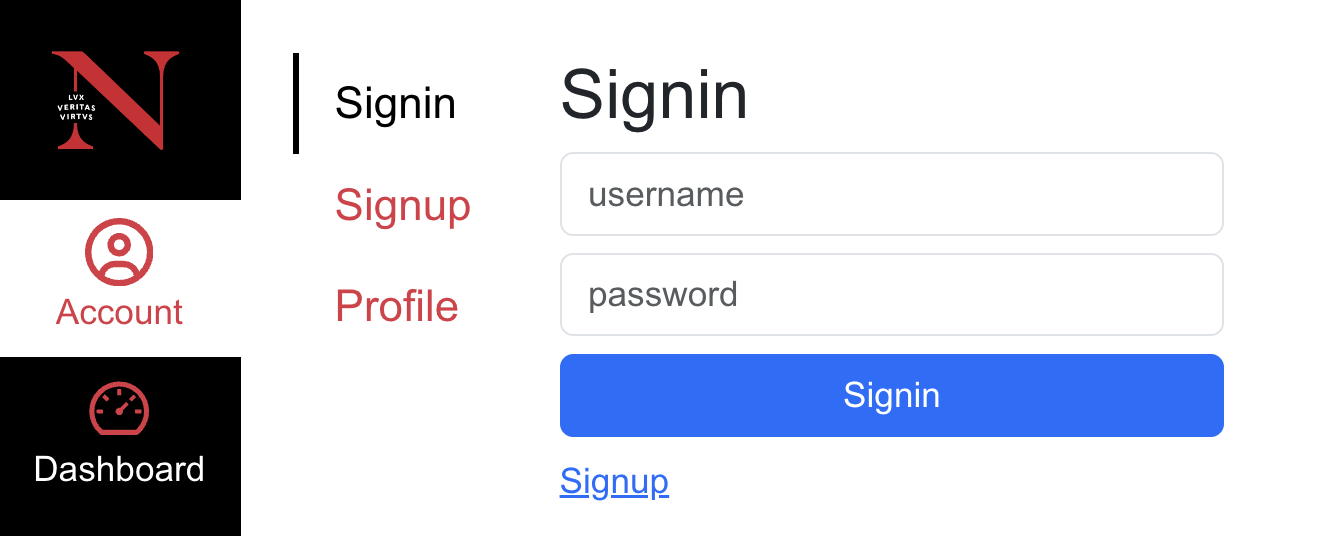
| ***src/Kanbas/Account/Signin.tsx*** | | |  |
| --- | --- | --- | --- |
| import { useState } from "react";  import { Link, useNavigate } from "react-router-dom";  import \* as client from "./client";  export default function Signin() {  const [credentials, setCredentials] = useState<any>({});  const navigate = useNavigate();  const signin = async () => {  await client.signin(credentials);  navigate("/Kanbas/Account/Profile");  };  return (  <div id="wd-signin-screen">  <h1>Sign in</h1>  <input id="wd-username" onChange={(e) => setCredentials({ ...credentials, username: e.target.value })}  value={credentials.username} className="form-control mb-2" placeholder="username" />  <input id="wd-password" onChange={(e) => setCredentials({ ...credentials, password: e.target.value }) }  value={credentials.password} className="form-control mb-2" placeholder="password" type="password" />  <button id="wd-signin-btn" onClick={signin} className="btn btn-primary w-100"> Sign in </button>  <br />  <Link id="wd-signup-link" to="/Kanbas/Account/Signup">Sign up</Link>  </div>  );  } | | | |

Implement an ***Account*** component with a default navigation route to the ***Sign in*** screen as shown below. In the ***Kanbas*** component, update the ***Account*** route to render the ***Account*** component. When users click the ***Account*** link in the ***Kanbas Navigator***, they should see the ***Sign in*** screen. Confirm that clicking the ***Account*** link in ***Kanbas*** displays the ***Sign in*** screen.

| ***src/Kanbas/Account/index.tsx*** | | |  |
| --- | --- | --- | --- |
| import { Routes, Route, Navigate } from "react-router-dom";  import Signin from "./Signin";  import AccountNavigation from "./Navigation";  import { useSelector } from "react-redux";  export default function Account() {  return (  <div className="wd-account-screen">  <div className="d-flex">  <div className="d-none d-md-block">  <AccountNavigation />  </div>  <div className="flex-fill p-4 pt-0">  <Routes>  <Route path="/" element={ <Navigate to="/Kanbas/Account/Signin" /> } />  <Route path="/Signin" element={<Signin />} />  <Route path="/Signup" element={<h1>Sign Up</h1>} />  <Route path="/Profile" element={<h1>Profile</h1>} />  </Routes>  </div>  </div>  </div>  );  } | | | |

The ***AccountNavigation*** component referenced in the ***Account*** component above provides navigation links to screens ***Sign in***, ***Sign up***, and ***Profile*** as implemented below.

| ***src/Kanbas/Account/Navigation/index.tsx*** | |
| --- | --- |
| import { Link, useLocation, useParams } from "react-router-dom";  export default function AccountNavigation() {  const links = ["Signin", "Signup", "Profile"];  const { pathname } = useLocation();  return (  <div id="wd-account-navigation" className="list-group fs-5 rounded-0">  {links.map((link) => (  <Link to={`/Kanbas/Account/${link}`} className={`wd-link border border-0 list-group-item  ${ pathname.includes(link) ? "active text-black" : "text-danger" }`}> {link} </Link>  ))}  </div>  );  } | |

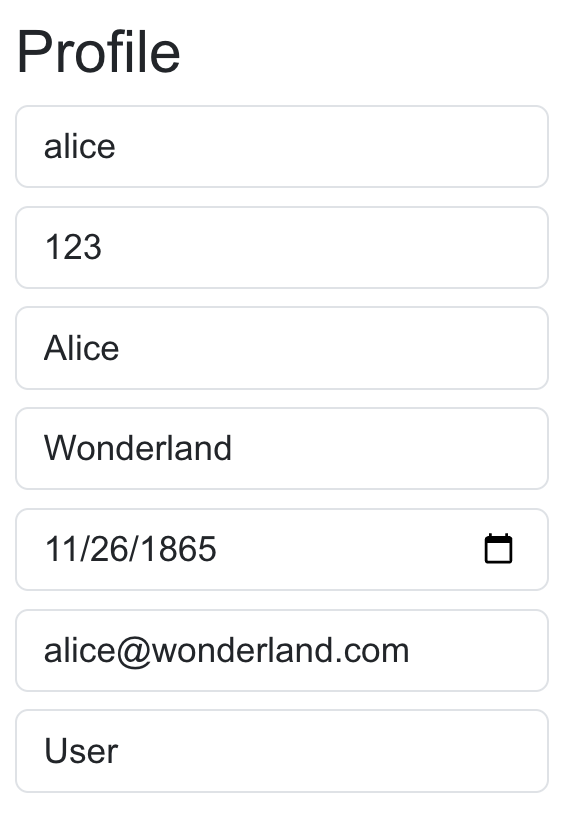
Add the ***Account*** component to the ***Kanbas*** router as shown below. Confirm that navigating to the ***Account*** screen displays the ***Sign in*** screen by default.

| ***src/Kanbas/index.tsx*** | | |  |
| --- | --- | --- | --- |
| <Routes>  <Route path="/Account/\*" element={<Account />} />  <Route path="/Dashboard" element={<Dashboard />} />  <Route path="/Courses/:courseId" element={<Courses />} />  </Routes> | | | |

## 5.2 Implementing a Profile Screen

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 (, 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, implement a function to retrieve the ***account*** information from the server route implemented above as shown below.

| ***src/Kanbas/Account/client.ts*** | |
| --- | --- |
| import axios from "axios";  export const USERS\_API = process.env.REACT\_APP\_REMOTE\_SERVER;  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. Style the screen so it renders as shown here on the right.

| ***src/Kanbas/Account/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<any>({});  const navigate = useNavigate();  const fetchProfile = async () => {  const account = await client.profile();  setProfile(account);  };  useEffect(() => { fetchProfile(); }, []);  return (  <div className="wd-profile-screen">  <h1>Profile</h1>  {profile && (  <div>  <input className="wd-username" value={profile.username}  onChange={(e) => setProfile({ ...profile, username: e.target.value })}/>  <input className="wd-password" value={profile.password}  onChange={(e) => setProfile({ ...profile, password: e.target.value })}/>  <input className="wd-firstname" value={profile.firstName}  onChange={(e) => setProfile({ ...profile, firstName: e.target.value })}/>  <input className="wd-lastname" value={profile.lastName}  onChange={(e) => setProfile({ ...profile, lastName: e.target.value })}/>  <input className="wd-dob" value={profile.dob}  onChange={(e) => setProfile({ ...profile, dob: e.target.value })} type="date"/>  <input className="wd-email" value={profile.email}  onChange={(e) => setProfile({ ...profile, email: e.target.value })}/>  <select className="wd-role" 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>  );  } | | | |

In the ***Account*** component, add a route to navigate to the ***Profile*** screen. Confirm that signing in navigates to the new ***Profile*** screen and displays information about the user that just signed in.

| ***src/Kanbas/Account/index.tsx*** | | |  |
| --- | --- | --- | --- |
| <Routes>  <Route path="/" element={<Navigate to="/Kanbas/Account/Signin" />} />  <Route path="/Signin" element={<Signin />} />  <Route path="/Signup" element={<h1>Sign up</h1>} />  <Route path="/Profile" element={<Profile />} />  </Routes> | | | |

## 5.3 Implementing a Sign Up 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***.

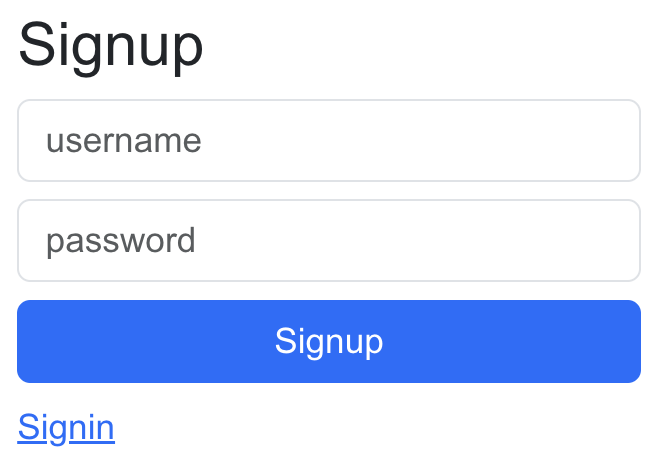
| ***Users/dao.js*** | | |  |
| --- | --- | --- | --- |
| import model from "./model.js";  export const createUser = (user) => model.create(user);  export const findUserByUsername = (username) => model.findOne({ username: username }); | | | |

We can use the DAO functions to implement the ***sign up*** operation for users to sign up 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;  ...  export default 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" });  return;  }  currentUser = await dao.createUser(req.body);  res.json(currentUser);  };  app.post("/api/users/signup", signup);  } | | | |

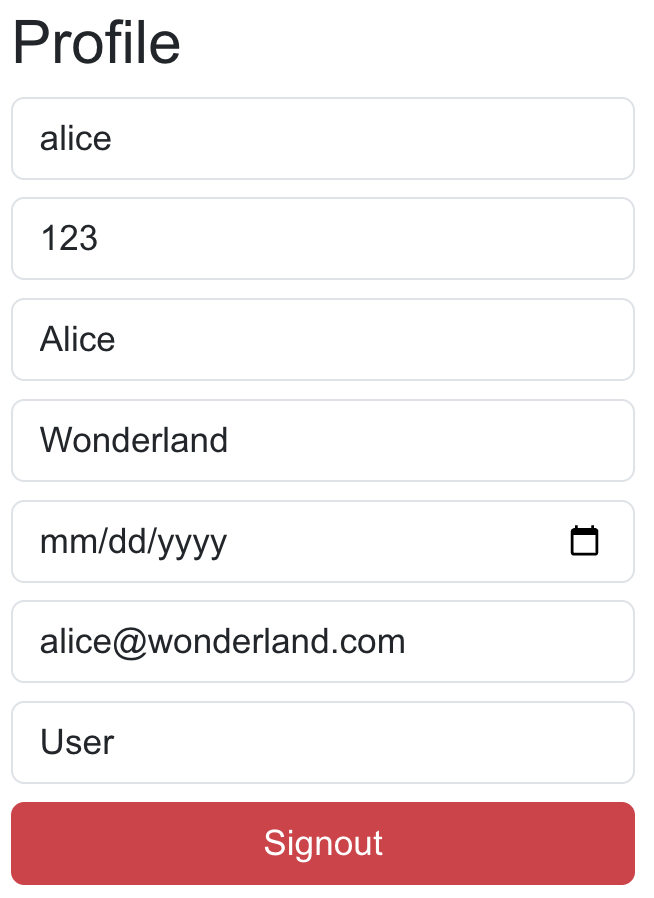
Meanwhile in the React user interface Web app, implement a ***signup*** client that posts the new user to the Web API as shown below.

| ***src/Kanbas/Account/client.ts*** | | |  |
| --- | --- | --- | --- |
| export const signup = async (user: any) => {  const response = await axios.post(`${USERS\_API}/signup`, user);  return response.data;  }; | | | |

Implement a ***Sign up*** screen component that users can use to type their username and password, and send the credentials to the server for sign up. If the sign up is successful, navigate to the ***Profile*** screen. In the ***Account*** component, update the ***signup*** route to display the new ***Sign up*** screen. In the ***Sign in*** screen, create a ***Link*** to navigate to the ***Sign up*** screen. Confirm that you can signup with a new username and password. Style the ***Sign up*** screen so it looks as shown below on the right.

| ***src/Kanbas/Account/Signup.tsx*** | | |  |
| --- | --- | --- | --- |
| import React, { useState } from "react";  import { Link, useNavigate } from "react-router-dom";  import \* as client from "./client";  export default function Signup() {  const [user, setUser] = useState<any>({});  const navigate = useNavigate();  const signup = async () => {  await client.signup(user);  navigate("/Kanbas/Account/Profile");  };  return (  <div className="wd-signup-screen">  <h1>Sign up</h1>  <input value={user.username} onChange={(e) => setUser({ ...user, username: e.target.value })}  className="wd-username form-control mb-2" placeholder="username" />  <input value={user.password} onChange={(e) => setUser({ ...user, password: e.target.value })} type="password"  className="wd-password form-control mb-2" placeholder="password" />  <button onClick={signup} className="wd-signup-btn btn btn-primary mb-2"> Sign up </button><br />  <Link to="/Kanbas/Account/Signin" className="wd-signin-link">Sign in</Link>  </div>  );  } | | | |

## 5.4 Implementing Signout

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

| ***Users/routes.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 ***Sign out*** button that invokes the ***signout*** client function and then navigates to the ***Sign in*** screen. Confirm that you can signout and navigate to the ***Sign in*** screen.

| ***src/Kanbas/Account/Profile.tsx*** | | |  |
| --- | --- | --- | --- |
| ...  const signout = async () => {  await client.signout();  navigate("/Kanbas/Account/Signin");  };  ...  <button onClick={signout} className="wd-signout-btn btn btn-danger w-100">  Sign out  </button>  ... | | | |

## 5.5 Supporting Multiple User Sessions

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

### 5.5.1 Installing and Configuring Server Sessions

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

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

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 sessions ***after*** configuring cors.

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

Install the ***dotenv*** library so we can read configurations from environment variables.

| **$ npm install dotenv** |
| --- |

In a new ***.env*** file at the root of your project, declare the following environment variables.

| ***.env*** |  |
| --- | --- |
| NODE\_ENV=development  NETLIFY\_URL=http://localhost:3000  REMOTE\_SERVER=http://localhost:4000  SESSION\_SECRET=super session secret  MONGO\_CONNECTION\_STRING=mongodb://localhost:27017/kanbas | |

Then in ***App.js***, import the ***dotenv*** library so we can determine whether we are running in our development environment and configure session accordingly. ***Note:*** the following configuration has been tested on ***Google's Chrome browser***, and is not guaranteed to work on other browsers.

| ***App.js*** |  |
| --- | --- |
| import "dotenv/config";  import session from "express-session";  const app = express();  app.use(  cors({  credentials: true,  origin: process.env.NETLIFY\_URL || "http://localhost:3000",  })  );  const sessionOptions = {  secret: process.env.SESSION\_SECRET || "kanbas",  resave: false,  saveUninitialized: false,  };  if (process.env.NODE\_ENV !== "development") {  sessionOptions.proxy = true;  sessionOptions.cookie = {  sameSite: "none",  secure: true,  domain: process.env.NODE\_SERVER\_DOMAIN,  };  }  app.use(session(sessionOptions)); | *// import the new dotenv library*  *// to read .env file*  *// use different front end URL in dev // and in production*  *// default session options*  *// in production*  *// turn on proxy support*  *// configure cookies for remote server* |

The ***signup*** route retrieves the ***username*** the request body. If there's already a user with that username, then we respond 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;~~  ...  export default 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" });  return;  }  const currentUser = await dao.createUser(req.body);  req.session["currentUser"] = currentUser;  res.json(currentUser);  }; | | |

An existing user can identify themselves by providing credentials. The ***signin*** route below looks up the user by their credentials, stores it in ***currentUser*** session, and responds with the user if they exist. Otherwise responds 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.status(401).json({ message: "Unable to login. Try again later." });  }  }; | | |

If a user has already signed in, we can retrieve the ***currentUser*** from the session by using the ***profile*** route as shown below. If there's no ***currentUser***, we respond with an error.

| ***Users/routes.js*** | |  |
| --- | --- | --- |
| const profile = (req, res) => {  const currentUser = req.session["currentUser"];  if (!currentUser) {  res.sendStatus(401);  return;  }  res.json(currentUser);  }; | | |

Finally we can sign out users by destroying the session.

| ***Users/routes.js*** | | |
| --- | --- | --- |
| const signout = (req, res) => {  req.session.destroy();  res.sendStatus(200);  }; | | |

### 5.5.2 Configuring Axios to Support Server Sessions

By default ***axios*** does not support cookies. To configure ***axios*** to include cookies in requests, use the ***axios.create()*** to create an instance of the library that includes cookies for credentials as shown below. Then replace all occurrences of the ***axios*** library with this new version ***axiosWithCredentials***.

| ***src/Kanbas/Account/client.ts*** | |
| --- | --- |
| import axios from "axios";  const axiosWithCredentials = axios.create({ withCredentials: true });  export const REMOTE\_SERVER = process.env.REACT\_APP\_REMOTE\_SERVER;  export const USERS\_API = `${REMOTE\_SERVER}/api/users`;  export const signin = async (credentials: any) => {  const response = await axiosWithCredentials.post(`${USERS\_API}/signin`, credentials);  return response.data;  };  export const profile = async () => {  const response = await axiosWithCredentials.post(`${USERS\_API}/profile`);  return response.data;  };  export const signup = async (user: any) => {  const response = await axiosWithCredentials.post(`${USERS\_API}/signup`, user);  return response.data;  };  export const signout = async () => {  const response = await axiosWithCredentials.post(`${USERS\_API}/signout`);  return response.data;  }; | |

## 5.6 Handling Session Errors in the User Interface

The Web API route for signing in responds with an error if the user is not found. The error can be caught in the ***Signin*** component by wrapping the request with a ***try catch*** block as shown below. If there's an error, the component will not navigate to the ***Profile***, and instead display the error as an alert box. Confirm that trying to sign in with the wrong credentials displays a corresponding error message.

| ***src/Kanbas/Account/Signin.tsx*** | |
| --- | --- |
| export default function Signin() {  const [error, setError] = useState("");  const signin = async () => {  try {  await client.signin(credentials);  navigate("/Kanbas/Account/Profile");  } catch (err: any) {  setError(err.response.data.message);  }  };  ...  return (  <div className="wd-signin-screen">  <h1>Sign in</h1>  {error && <div className="wd-error alert alert-danger">{error}</div>}  ...  </div>  );  } | |

Similarly the Web API route for signing up responds with an error if the username is already taken. The ***Signup*** component below handles the error by wrapping the request with a ***try catch*** block. If there's an error the component will not navigate to the ***Profile***, and instead display the error as an alert box. Confirm that trying to sign up with an existing user displays a corresponding error message.

| ***src/Kanbas/Account/Signup.tsx*** | |
| --- | --- |
| export default function Signup() {  const [error, setError] = useState("");  const signup = async () => {  try {  await client.signup(user);  navigate("/Kanbas/Account/Profile");  } catch (err: any) {  setError(err.response.data.message);  }  };  return (  <div className="wd-signup-screen">  <h1>Sign up</h1>  {error && <div className="wd-error alert alert-danger">{error}</div>}  ...  </div>  );  } | |

The ***Profile*** component handles errors from the server by navigating to the ***Sign in*** screen as shown below.

| ***src/Kanbas/Account/Profile.tsx*** | |
| --- | --- |
| const fetchProfile = async () => {  try {  const account = await client.profile();  setProfile(account);  } catch (err: any) {  navigate("/Kanbas/Account/Signin");  }  }; | |

## 5.7 Account Reducer

The currently signed in user is information that is worth sharing across the entire application, and a reducer is the best way to share application wide information. Implement the ***account reducer*** shown below.

| ***src/Kanbas/Account/reducer.ts*** | |
| --- | --- |
| import { createSlice } from "@reduxjs/toolkit";  const initialState = {  currentUser: null,  };  const accountSlice = createSlice({  name: "account",  initialState,  reducers: {  setCurrentUser: (state, action) => {  state.currentUser = action.payload;  },  },  });  export const { setCurrentUser } = accountSlice.actions;  export default accountSlice.reducer; | |

Add the new reducer to the store as shown below.

| ***src/Kanbas/store.ts*** | |
| --- | --- |
| import { configureStore } from "@reduxjs/toolkit";  import modulesReducer from "./Courses/Modules/reducer";  import accountReducer from "./Account/reducer";  const store = configureStore({  reducer: {  modulesReducer,  accountReducer,  },  });  export default store; | |

Create a new ***Session*** component that fetches the ***current user*** from the server and stores it in the store so that the rest of the application can have access to the ***current user***. The ***Session*** component will wrap the rest of the application so that it renders before all other components to check if anyone is signed in. Once it figures out either way, it'll store the result in the store and let the rest of the components render.

| ***src/Account/Session.tsx*** | |
| --- | --- |
| import \* as client from "./client";  import { useEffect, useState } from "react";  import { setCurrentUser } from "./reducer";  import { useDispatch } from "react-redux";  export default function Session({ children }: { children: any }) {  const [pending, setPending] = useState(true);  const dispatch = useDispatch();  const fetchProfile = async () => {  try {  const currentUser = await client.profile();  dispatch(setCurrentUser(currentUser));  } catch (err: any) {  console.error(err);  }  setPending(false);  };  useEffect(() => {  fetchProfile();  }, []);  if (!pending) {  return children;  }  } | |

When someone signs in, the ***current user*** needs to be stored in the state variable to let the rest of the application that someone is signed in.

| ***src/Kanbas/Account/Signin.tsx*** | |
| --- | --- |
| import \* as client from "./client";  import { useDispatch } from "react-redux";  import { setCurrentUser } from "./reducer";  export default function Signin() {  ...  const dispatch = useDispatch();  const signin = async () => {  try {  const currentUser = await client.signin(credentials);  dispatch(setCurrentUser(currentUser));  navigate("/Kanbas/Account/Profile");  } catch (err: any) {  setError(err.response.data.message);  }  };  ...  } | |

When someone signs up, the ***current user*** needs to be stored in the state variable to let the rest of the application that someone is signed in.

| ***src/Kanbas/Account/Signup.tsx*** | |
| --- | --- |
| import \* as client from "./client";  import { useDispatch } from "react-redux";  import { setCurrentUser } from "./reducer";  export default function Signup() {  const dispatch = useDispatch();  const signup = async () => {  try {  const currentUser = await client.signup(user);  dispatch(setCurrentUser(currentUser));  navigate("/Kanbas/Account/Profile");  } catch (err: any) {  setError(err.response.data.message);  }  };  } | |

If a user signs out, the ***current user*** state variable needs to be cleared so that the rest of the application knows that no one is signed in anymore.

| ***src/Kanbas/Account/Profile.tsx*** | |
| --- | --- |
| import \* as client from "./client";  import { useDispatch } from "react-redux";  import { setCurrentUser } from "./reducer";  export default function Profile() {  ...  const dispatch = useDispatch();  const signout = async () => {  await client.signout();  dispatch(setCurrentUser(null));  navigate("/Kanbas/Account/Signin");  };  ...  } | |

Now the the ***current user*** state variable can be used throughout the application to write logic based on the ***current user***. For instance, we can control what links show in the navigation based on whether there's someone signed in or not. Confirm that if no one is signed in they see the ***Sign In*** and ***Sign Up*** links, but not the ***Profile*** link. But if someone is signed in, they should only see the ***Profile*** link.

| ***src/Kanbas/Account/Navigation/index.tsx*** | |
| --- | --- |
| export default function AccountNavigation() {  const { currentUser } = useSelector((state: any) => state.accountReducer);  const links = currentUser ? ["Profile"] : ["Signin", "Signup"];  const { pathname } = useLocation();  ...  } | |

Another enhancement is that when some one is signed in, and they navigate to the ***Account***, the user should automatically navigate to their ***Profile***. The ***Account*** component below retrieves the ***currentUser*** state variable and navigates to the ***Profile*** if there's a ***currentUser***, and navigates to the ***Sign in*** screen if no one is signed in.

| ***src/Kanbas/Account/index.tsx*** | |
| --- | --- |
| import { useSelector } from "react-redux";  export default function Account() {  const { currentUser } = useSelector((state: any) => state.accountReducer);  ...  <Routes>  <Route path="/" element={<Navigate to={ currentUser ? "/Kanbas/Account/Profile" : "/Kanbas/Account/Signin" }/>}/>  <Route path="/Signin" element={<Signin />} />  <Route path="/Signup" element={<Signup />} />  <Route path="/Profile" element={<Profile />} />  </Routes>  ...  );  } | |

## 5.8 Protecting Routes

Often applications have screens that are only accessible if users are logged in, usually because the information is sensitive and/or the information they are accessing is based on the identify of the user. We can protect navigating to certain routes in the user interface by checking if a user is signed in already or not and then either allowing access to the route, or navigating users to the sign in screen instead. The ***ProtectedRoute*** component below uses the ***currentUser*** in the store to determine whether there's someone signed in or not. The ***children*** parameter is a reference to the protected component and if there's someone signed in, the ***ProtectedRoute*** returns the ***children*** reference allowing the signed in user to access the route. If no one is signed in, ***ProtedRoute*** navigates the user to the ***Sign in*** screen.

| ***src/Kanbas/ProtectedRoute.tsx*** | |
| --- | --- |
| import React, { useState, useEffect } from "react";  import { useSelector } from "react-redux";  import { Navigate } from "react-router-dom";  export default function ProtectedRoute({ children }: { children: any }) {  const { currentUser } = useSelector((state: any) => state.accountReducer);  if (currentUser) {  return children;  } else {  return <Navigate to="/Kanbas/Account/Signin" />;  }  } | |

Use the ***ProtectedRoute*** component to protect the ***Dashboard*** and ***Courses*** routes so that users will only be able to navigate there if they are signed in.

| ***src/Kanbas/index.tsx*** | |
| --- | --- |
| <Routes>  <Route path="/" element={<Navigate to="Dashboard" />} />  <Route path="Account/\*" element={<Account />} />  <Route path="Dashboard" element={<ProtectedRoute><Dashboard ... /></ProtectedRoute> } />  <Route path="Courses/:cid/\*" element={<ProtectedRoute><Courses courses={courses} /></ProtectedRoute> } />  <Route path="Calendar" element={<h1>Calendar</h1>} />  <Route path="Inbox" element={<h1>Inbox</h1>} />  </Routes> | |

## 5.9 Configuring Session in Remote Servers

The local Node server was configured to support multiple sessions. Similarly, the remote server also needs to be configured to support sessions. In ***Render.com***, navigate to the ***Environment*** section of the application dashboard. Click ***Add Environment Variable***, type the name of the variables in the ***Key*** column and the variable's value in the ***Value*** column. Repeat for each of the environment variables as shown here on the right. You will need to restart the remote server by clicking ***Manual Deploy*** and then ***Deploy latest commit***.

Below is an example of the environment variables used to configure a remote server running on ***Render.com***. Use the same ***Environment Variable*** keys shown on the left column below. Don't use the values shown on the right column below. Instead use the values for your Mongo Database, your Netlify remote server, and your remote Node server.

| ***Environment Variable*** |  | ***Value*** |
| --- | --- | --- |
| MONGO\_CONNECTION\_STRING |  | mongodb+srv://giuseppi:supersecretpassword@kanbas.jxui0bc.mongodb.net/kanbas?retryWrites=true&w=majority&appName=Kanbas |
| NETLIFY\_URL |  | https://a6--kanbas-react-web-app-su24.netlify.app |
| NODE\_SERVER\_DOMAIN |  | kanbas-node-server-app-su24.onrender.com |
| NODE\_ENV |  | production |
| SESSION\_SECRET |  | what ever |

# 6 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. Confirm that all ***courses*** and ***modules*** ***CRUD*** functionality works as expected. Optionally also migrate the assignments into a collection and confirm that all assignment ***CRUD*** operations work. Note that the ***modules*** ***course*** field might be referencing an ***ID*** of a course that does not exist. Update the ***module*** ***course*** fields so that they refer to the correct ***course*** document.

# 7 Deliverables

As a deliverable, make sure you complete all the lab exercises, mongoose schemas, models, DAOs, React components, and they behave as described. For both the React and Node repositories, all your work should be done in a branch called ***a6***. When done, add, commit and push both branches to their respective GitHub repositories. Deploy the new branches to ***Netlify*** and ***Render*** (or ***Heroku***) and confirm they integrate. If you are using ***Render***, it does not create a separate branch deployment like ***Netlify*** so you'll have to deploy to an entirely different Web service so that you don't trample the previous assignment while the TAs are still grading. The Node server application running on Render will need to be configured to interact with the ***a6*** ***Netlify*** branch deployment. All the exercises should work remotely just as well as locally. The Kanbas Dashboard should display the courses and modules from the database. As a deliverable in ***Canvas***, submit the URL to the ***a6*** branch deployment of your React.js application running on Netlify.