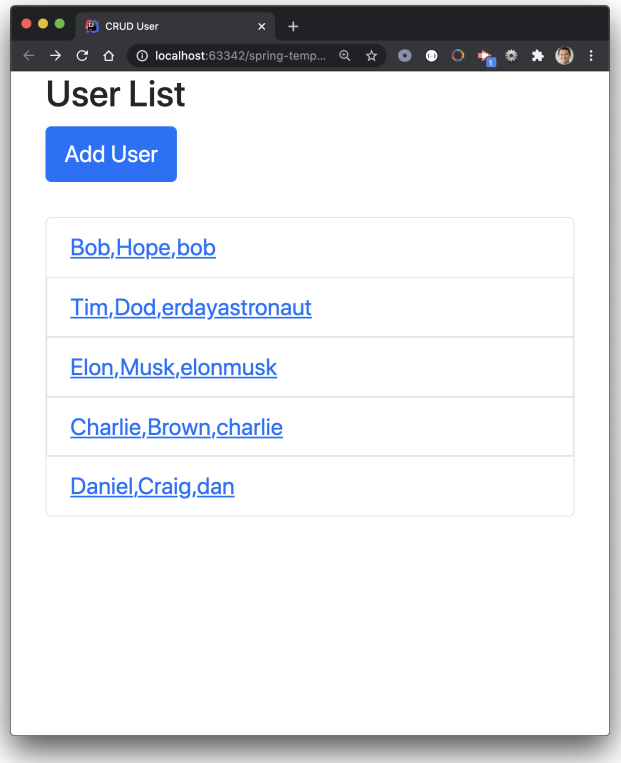
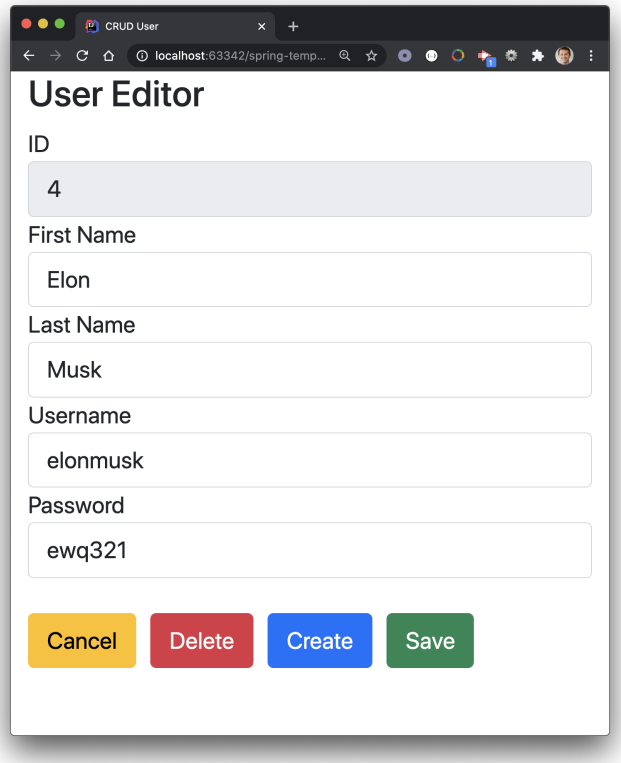
# Data Driven Web Applications

## Introduction

In this assignment we're going to practice building a simple user interface to interact with an existing data model. We'll provide the whole back end including the database, repository, DAO, and RESTful HTTP API. We'll walk you through building a user interface to create, read, update, and delete (CRUD) records in the users table we worked on in a previous assignment. The intention is for you to learn how to build data driven applications in general, and Web data driven applications in particular. The very popular React.js library will be used to build two screens that implement full CRUD operations on a database table. The screens are illustrated below.




We'll take care of the tedious part of plumbing the React.js framework so that you can concentrate on the fun part of interacting with the data model on the server and rendering and updating the user interface.

## Importing a Database

In a previous assignment you created a schema called ***db_design***. Double click the schema ***db_design*** so that it is the default schema and all commands will be based on that schema. Download the data from my GitHub and import it into your ***db_design*** schema. Unzip the downloaded file into a folder. To import the data, from ***MySQL Workbench***, from the main menu, select ***Server***, ***Data Import***. In the ***Data Import*** screen, click the ***Import from Disk*** tab, and then click the ***Import from Dump Project Folder***. Click the browse button all the way on the right to select the directory you unzipped earlier. Select the folder and then click on ***Start Import***. Close the screen when done. Double click the ***db_design*** schema on the left hand side to make it the default schema.

# Clone or download the sample code

We have prepared a Java project that has already been configured to connect to a MySQL database. You'll need to clone it or download the project, configure it and run it from your local computer. Download the project from https://github.com/jannunzi/db-design-orm-assignment. If you have git installed you can clone the project as follows from a terminal or console window:

```
git clone https://github.com/jannunzi/db-design-orm-assignment.git
```

Alternatively you can download the code by clicking on Code and then Download ZIP and then unzipping the file. In either case you'll end up with a new folder called ***db-design-orm-assignment***. We'll refer to this new folder as the ***root of the project***. Do all your work in the new folder.

# Reviewing the data model

Let's review the data model we are working with. Open ***User.java*** and confirm it is an entity mapped to a ***users*** table. Check the definition of the ***users*** table in Workbench and verify the schemas match. If the User class is missing the annotations, copy them from the code sample below.

```
@Entity
@Table(name="users")
public class User {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Integer id;
   private String firstName;
   private String lastName;
   private String username;
   private String password;
   private String profilePicture;
   private String handle;
   // setters and getters
}
```

```
CREATE TABLE `users` (

 `id` int(11) NOT NULL AUTO_INCREMENT,
 `first_name` varchar(45) DEFAULT NULL,
 `last_name` varchar(45) DEFAULT NULL,
 `username` varchar(45) DEFAULT NULL,
 `password` varchar(45) DEFAULT NULL,
 `profile_picture` varchar(45) DEFAULT NULL,
 `handle` varchar(45) DEFAULT NULL,
 PRIMARY KEY (`id`)
)
```

To interact with the users table we provide a repository configured to create, read, update and delete (CRUD) instances of ***User*** as records in the ***users*** table. Open the ***UserRestRepository.java*** interface and verify it is written in terms of ***CrudRepository*** and configured to map the ***User*** to the ***users*** table.

```
public interface UserRestRepository
      extends CrudRepository<User, Integer> {
   @Query(value = "SELECT * FROM users",
         nativeQuery = true)
   public List<User> findAllUsers();
   @Query(value = "SELECT * FROM users WHERE id=:userId",
         nativeQuery = true)
   public User findUserById(@Param("userId") Integer id);
}
```

```
// implements ORM access to database
// based on existing implementation
// override SQL query used to retrieve
// all users
// wrap query in highlevel method
// override SQL query to retrieve user
// by their ID
// wrap query in highlevel method
```

Finally review the ***UserRestOrmDao.java*** that implements the CRUD operations in terms of the repository and makes the operations available as HTTP end points accessible to clients that want to integrate with the data model.

```
@RestController                          // make this available through HTTP
@CrossOrigin(origins = "*")              // allow access from any internet domain
public class UserRestOrmDao{             // implements database access using ORM
   @Autowired                            // automatically instantiate
   UserRestRepository userRepository;    // implements ORM access to database
   ...
}
```

# Reviewing user interface dependencies

The user interface for this assignment is implemented in ***src/main/webapp/react/social/index.html*** shown below. The user interface is dynamically rendered in the ***<DIV>*** with ID ***root***. JavaScript files *.js work together to retrieve data from the server and update the content the user sees.

```
<div id="root"></div>                                // where application renders
<script src="users/user-service.js"></script>        // communication with server
<script src="users/user-form-editor.js"></script>    // screen for editing a user with a form
<script src="users/user-list.js"></script>           // screen listing all users
<script src="index.js"></script>                     // entry point of applications
<script type="text/jsx">
   import App from "index";                          // load application
   ReactDOM.render(                                  // render
      React.createElement(App),                      // the application
      document.getElementById('root')                // in
   );
</script>
```

It all starts with the last ***SCRIPT*** above that imports the ***index.js*** script, creates an instance of the ***App*** and then renders it in the ***DIV*** element with ID ***root***. The dependencies have already been taken care of, so you don't need to do anything here.

# Review index.js

The root of the application is implemented in ***src/main/webapp/react/social/index.js*** in the ***App*** React component shown below. The ***App*** component implements navigation between two screens implemented in React components ***UserList*** and ***UserFormEditor*** as shown below.

```
import UserList from "./users/user-list";                    // load UserList screen
import UserFormEditor from "./users/user-form-editor";       // load UserFormEditor screen
const {HashRouter, Route} = window.ReactRouterDOM;           // libraries for screen navigation
const App = () => {                                          // root component
```

```
  return (
    <div>
      <HashRouter>                                  // screen navigation definitions
        <Route path={["/users", "/"]} exact={true}>  // if browser URL matches "/users" or "/" ...
          <UserList/>                               // then show screen UserList
        </Route>
        <Route path="/users/:id" exact={true}>      // if browser URL matches "/users/:id"
          <UserFormEditor/>                         // then show screen UserFormEditor
        </Route>                                    // where ":id" is a placeholder for a user's ID
      </HashRouter>
    </div>
  );
}
export default App;
```

***HashRouter*** defines two routes for navigating to either of two screens implemented by React components ***UserList*** and ***UserFormEditor*** implemented under ***src/main/webapp/react/social/users/user-list.js*** and ***src/main/webapp/react/social/users/user-form-editor.js***. If the user types a URL that matches "***/***" or "***/users***", then screen ***UserList*** is rendered. If the user types a URL that matches "***/users***" followed by a user ID, then screen ***UserFormEditor*** is rendered. The ***UserList*** component defines the default screen that displays when the user first loads the application. As the names suggest, the ***UserList*** screen will render a list of users and the ***UserFormEditor*** screen allows editing a user using a form. These two screens will allow full CRUD interaction with the ***users*** table through an HTTP API. Communication with the server will be implemented in ***src/main/webapp/react/social/users/user-service.js***. The navigation has already been setup for you so you don't have to do anything here either.

# Retrieving all records from the database

Let's first work on getting a list of the users stored in the users table. Review the ***findAllUsers()*** already implemented in the ***UserRestOrmDao.java*** shown below. Note that the method is mapped to an HTTP GET URL that matches "***/api/users***".

```
  @GetMapping("/api/users")                 // map this method to an HTTP GET request
  public List<User> findAllUsers() {        // execute this method is URL matches /api/users
    return userRepository.findAllUsers();   // retrieve all records from users table and return as list of users
  }
```

Start the server and point your browser to http://localhost:8080/api/users. Confirm that the user records are displayed. In ***users/user-services.js***, let's try retrieving the users from within our Web application as shown below. Use the code below to complete the relevant code in users/user-services.js marked as ***TODO***. The ***USERS_URL*** constant declares where the server is listening for HTTP requests. The ***findAllUsers*** function retrieves users from the server.

```
const USERS_URL = "http://localhost:8080/api/users"    // where the server is listening for requests

export const findAllUsers = () =>
  fetch(USERS_URL)                                     // send GET request to server
```

```
    .then(response => response.json())          // parse response from server

export default {
  findAllUsers                                  // export functions in this file for others to import
}
```

In the ***users/user-list.js*** React component, implement a screen that retrieves users from the server and renders them as a list. Let's use ***user-service***'s ***findAllUsers*** function to retrieve all users from the server and store them in a ***users*** state variable initialized as an empty array. Use the highlighted coded below as a guide to retrieve all users from the server.

```
import userService from "./user-service"    // import userService to talk to the server
const { useState, useEffect } = React;      // import state management React hooks
const UserList = () => {
  const [users, setUsers] = useState([])    // create a users state variable
  useEffect(() => {                         // on load
    findAllUsers()                          // call local function findAllUsers()
  }, [])
  const findAllUsers = () =>                // local function findAllUsers
    userService.findAllUsers()              // use userService.findAllUsers() to retrieve users from server
      .then(users => setUsers(users))       // store them in local users state variable
  return( ... )
}
```

In ***user-list.js***, in the <UL> tag, iterate over the users array and add a line item tag <LI> for each user in the array. Remove any static <LI> that might already be there. Render the user's first name, last name, and username for each item as shown below.

```
  return(
    <div>
      <h2>Users</h2>
      <ul>
      {
        users.map(user =>           // in the <ul> tag
          <li key={user.id}>
            {user.firstName},       // iterate over the users array, for each user
            {user.lastName},        // add a line item tag <li>, for each user
            {user.username}         // render user's first, last name, and username
          </li>)
      }
      </ul>
    </div>
  )
```

To test, start the server and then open the ***/src/main/webapp/react/social/index.html*** file with your browser. To run the file, right click it and select ***Run index.html***. Verify that a list of users is rendered and it matches the records in the users table. To make the output prettier you can optionally add the "***list-group***" class to <UL> and "***list-group-item***" class to <LI>

# Retrieving a record by their id

Let's now work on selecting one of the users and then editing the user. First let's navigate to a form screen to CRUD user records. In ***user-list.js***, use the highlighted coded below as a guide to navigate to a user form screen.

```
const {Link} = window.ReactRouterDOM;          // import Link component to navigate to other screens
const UserList = () => {
        ...
        <ul className="list-group">
          {
            users.map(user =>
              <li className="list-group-item"     // added optional class
                key={user.id}>
                <Link to={`/users/${user.id}`}>   // navigate to /users/<ID> where ID is the user's ID
                  {user.firstName},
                  {user.lastName},
                  {user.username}
                </Link>
              </li>)
          }
        </ul>
        ...
}
```

The users are now going to render as hyperlinks that navigate to user-edit-form.js screen when you click them. In ***user-edit-form.js***, we'll implement a screen with a form to edit the selected user. The URL that navigates to the form screen contains the ID of the user we clicked on. We can use the ID to retrieve the user from the server using the HTTP GET endpoint defined in UserRestOrmDao.java as shown below.

```
  @GetMapping("/api/users/{userId}")          // map this method to HTTP GET request
  public User findUserById(                   // execute this method when URL matches /api/users/ID
      @PathVariable("userId") Integer id) {   // parse user ID from path variable userId
    return userRepository.findUserById(id);   // retrieve single user by ID and return as instance of User
  }
```

In user-service.js we previously implemented findAllUsers. Let's now implement findUserById to retrieve the single user. Use the highlighted code below to implement findUserById.

```
const USERS_URL = "..."
export const findAllUsers = () => ...
export const findUserById = (id) =>           // send HTTP GET request to server
  fetch(`${USERS_URL}/${id}`)                 // encode user ID at end of path
    .then(response => response.json())        // parse HTTP response body as JSON

export default {
  findAllUsers,
  findUserById                                // export this function for others to import
}
```

In user-form-editor.js, get the ID of the user from the browser's path

```
const {useParams} = window.ReactRouterDOM;      // import userParams to parse parameters from URL
const UserFormEditor = () => {
   const {id} = useParams()                     // parse "id" from URL
   ...                                          // as defined in URL pattern in index.js
}
```

In user-form-editor.js, import the user service and use it to retrieve the user for the ID

```
import userService from "./user-service"        // import user-service so we can fetch a single user
const {useParams} = window.ReactRouterDOM;
const UserFormEditor = () => {
   const {id} = useParams()
   const findUserById = (id) =>                 // fetch a single user using their ID
      userService.findUserById(id)              // use user service's new findUserById

   ...
}
```

In user-form-editor.js, use React's useState and useEffect hooks to load a user identified by the ID encoded in the browser's path

```
import userService from "./user-service"
const {useState, useEffect} = React;            // import React's hooks
const {useParams} = window.ReactRouterDOM;
const UserFormEditor = () => {
   const {id} = useParams()
   const [user, setUser] = useState({})
   useEffect(() => {                            // on load
      findUserById(id)                          // find the user by their ID encoded from path
   }, []);
   const findUserById = (id) =>
      userService.findUserById(id)
         .then(user => setUser(user))           // store user from server to local user state variable
   ...
}
```

In user-form-editor.js, render the user in the form. Replace the labels and input fields if you have to

```
   return (
      <div>
         <h2>User Editor</h2>
         <label>ID</label>
         <input value={user.id}/><br/>
         <label>First Name</label>
         <input value={user.firstName}/><br/>
         <label>Last Name</label>
```

```
        <input value={user.lastName}/><br/>
        <label>Username</label>
        <input value={user.username}/><br/>
        <label>Password</label>
        <input value={user.password}/><br/>
        <button>Cancel</button>
        <button>Delete</button>
        <button>Create</button>
        <button>Save</button>
      </div>
  )
```

To test, open index.html with your browser and verify the list of users is rendered. Now click on one of the users and confirm that the application navigates to user-edit-form.js and it renders the user you clicked.

# Navigating back from an editor component to a list component

Let's add a cancel button so we can return from the form back to user list. In user-form-editor.js, add a cancel button to go back to the user-list.js screen. Add the highlighted code shown below.

```
const {useParams, useHistory} = window.ReactRouterDOM;    // import useHistory

const UserFormEditor = () => {
  ...
  return (
    <div>
      <h2>User Editor</h2>
      …
      <br/><label>Password</label>
      <input value={user.password}/>
      <button                                              // add a button
        onClick={() => {                                   // when you click
        history.back()}}>                                  // use history to go back
        Cancel                                             // Cancel label
      </button>
      <button>Delete</button>
      <button>Create</button>
      <button>Save</button>
    </div>
  )
```

To test, navigate to a user and then click on the new Cancel button. Confirm that you navigate back to list of users

# Deleting records from the database

Now let's implement deleting a user. Let's add a delete button at the end of the user edit form and use the server's deleteUser method to permanently remove the use record from the database. In UserRestOrmDao, review the deleteUser method shown below.

```
  @DeleteMapping("/api/users/{userId}")      // map this method to HTTP DELETE request
  public void deleteUser(                    // execute this method is URL matches /api/users/ID
        @PathVariable("userId") Integer id) {  // parse user's ID from path variable
    userRepository.deleteById(id);           // use repository to permanently remove the user by their ID
  }
```

Now let's try to send an HTTP DELETE request to the server from our user interface. In ***user-service.js***, add deleteUser function as shown below.

```
const USERS_URL = "..."
export const findAllUsers = () => …
export const findUserById = (id) => ...
export const deleteUser = (id) =>          // deleteUser function accepts user's ID
  fetch(`${USERS_URL}/${id}`, {            // encode user's ID at the end of the URL
    method: "DELETE"                       // send an HTTP DELETE request to the server
  })

export default {
  findAllUsers,
  findUserById,
  deleteUser                               // export method for others to import
}
```

Now let's use the deleteUser we just declared in user-form-editor.js. Declare a deleteUser event handling function we can invoke when we click on a new Delete button as shown below

```
const UserFormEditor = () => {
  ...
  const deleteUser = (id) =>                       // deleteUser event handler accepts user's ID
    userService.deleteUser(id)                     // invokes user service's deleteUse passing ID
      .then(() => history.goBack())                // if successful, navigate back to user list
  return (
    <div>
      <h2>User Editor</h2>
      …
      <button                                      // new Delete button
        onClick={() => deleteUser(user.id)}>       // notifies deleteUser event handler on click
        Delete                                     // Delete label
      </button>
      <button>Create</button>
      <button>Save</button>
    </div>
  )
```

To test, select a user in the user list and then in the user edit form click on the new Delete button. Confirm that the user is removed from the database and the user interface navigates back to the user list.

# Creating new records in the database

Now let's implement creating new users by sending an HTTP POST request to the server including the new user encoded in the BODY of the HTTP request. In UserRestOrmDao, review the createUser method shown below.

```
  @PostMapping("/api/users")                          // map this method to an HTTP POST
  public User createUser(@RequestBody User user) {    // parse new user from HTTP Request BODY
    return userRepository.save(user);                 // insert new user into users table
  }
```

Now let's try to communicate with the server by sending an HTTP POST message. In user-service.js, add a function called createUser() that POSTs a new user to the server formatted as a string representation of a JSON user object. Use the highlighted code below as a guide.

```
const USERS_URL = "..."
export const findAllUsers = () => …
export const findUserById = (id) => ...
export const deleteUser = (id) => ...
export const createUser = (user) =>                   // accept a user object
  fetch(USERS_URL, {                                  // send user object to server
    method: 'POST',                                   // using HTTP POST
    body: JSON.stringify(user),                       // encode object as a JSON string
    headers: {'content-type': 'application/json'}     // tell server to interpret this as an object
  })
  .then(response => response.json())                  // parse response from server

export default {
  findAllUsers,
  findUserById,
  deleteUser,
  createUser                                          // export so others can import
}
```

In the user-list.js screen, let's add a button to add a new user. It'll navigate to the same user edit form we've been using, but we'll pass "new" as the user ID, meaning we don't want to edit an existing user, but create a new one instead.

```
const {Link, useHistory} = window.ReactRouterDOM;
const UserList = () => {
  const history = useHistory()
  …
  return(
    <div>
```

```
        <h2>Users</h2>
        <button onClick={() => history.push("/users/new")}>
          Add User
        </button>
        <ul>
        ...
        </ul>
      </div>
    )
}
```

In user-edit-form.js, we can check for the user ID so that if the ID is "new" then we should not load the non existing user. If the ID is different then "new", then we should load the existing user. We'll also add a new Create button that will send the new user to the server. Finally, we'll need to make the input fields editable by keeping track of any changes and updating the local user instance so we can send the latest user data to the server. Make the highlighted changes shown below to user-edit-form.js.

```
const UserFormEditor = () => {

  useEffect(() => {
    if(id !== "new") {                          // only load the user by ID if the ID is not "new"
      findUserById(id)
    }
  }, []);
  const createUser = (user) =>                  // handle Create button click to send user
    userService.createUser(user)                // to the server
      .then(() => history.goBack())             // then return to user list
  return (
    <div>
      <h2>User Editor</h2>
      <label>First Name</label>
      <input
        onChange={(e) =>                        // update local user object's first name
          setUser(user =>                       // as the user types in the
            ({...user, firstName: e.target.value}))}  // input field
        value={user.firstName}/>
      <label>Last Name</label>
      <input
        onChange={(e) =>                        // update local user object's last name
          setUser(user =>                       // as the user types in the
            ({...user, lastName: e.target.value}))}   // input field
        value={user.lastName}/>
      <label>Username</label>
      <input
        onChange={(e) =>                        // update local user object's username
          setUser(user =>                       // as the user types in the
            ({...user, username: e.target.value}))}   // input field
        value={user.username}/>
      <label>Password</label>
      <input
        onChange={(e) =>                        // update local user object's password
```

```
                setUser(user =>                          // as the user types in the
                  ({...user, password: e.target.value}))}  // input field
            value={user.password}/>
          ...
          <button                                        // new Create button to
            onClick={() => createUser(user)}>            // create new user
            Create                                       // label
          </button>
          <button>Save</button>
        </div>
    )
}
```

To test, refresh your index.html page on the browser, click on the new Add User button, confirm that it navigates to the user editor with ID "new". Then enter a new user's first name, last name, username, and password and click on the new Create button. Confirm the new user is added to the database

# Updating a record in the database

Finally we'll implement updating an existing user. Take a look at the updateUser method implemented in the UserRestOrmDao class shown below. The method is mapped to an HTTP PUT request, accepts a user ID in the path, and the new user data embedded in the request's BODY.

```
  @PutMapping("/api/users/{userId}")                          // map method to HTTP PUT
  public User updateUser(
      @PathVariable("userId") Integer id,                     // parse user's ID from URL
      @RequestBody User userUpdates) {                        // parse user object from BODY
    User user = userRepository.findUserById(id);              // retrieve user from database
    user.setFirstName(userUpdates.getFirstName());            // update
    user.setLastName(userUpdates.getLastName());              // user fields
    user.setUsername(userUpdates.getUsername());              // with new
    user.setPassword(userUpdates.getPassword());              // values from
    user.setProfilePicture(userUpdates.getProfilePicture());  // user interface
    user.setHandle(userUpdates.getHandle());
    return userRepository.save(user);                         // save changes to database
  }
```

On the user interface, let's send an HTTP PUT request to the server encoding the user's ID in the URL and the new user's data in the request's BODY. In user-service.js, make the changes highlighted below.

```
const USERS_URL = "..."
export const findUserById = (id) => ...
export const deleteUser = (id) => ...
export const createUser = (user) => ...
export const updateUser = (id, user) =>          // update a user whose ID is id and new values are in user
  fetch(`${USERS_URL}/${id}`, {                  // send request to server with ID embedded in URL
    method: 'PUT',                               // send an HTTP PUT request
    body: JSON.stringify(user),                  // embed user data in the BODY as JSON string
```

```
    headers: {'content-type': 'application/json'}    // tell server to interpret object as JSON
  })
  .then(response => response.json())                 // parse response

export default {
  findAllUsers,
  findUserById,
  deleteUser,
  createUser,
  updateUser                                         // export for others to import
}
```

The user form editor can now use the new update user service to send updates to the server. In user-edit-form.js add a new Save button that will send the changes to the server as shown below.

```
const UserFormEditor = () => {
  ...
  const updateUser = (id, newUser) =>                // update user with ID with new user data
    userService.updateUser(id, newUser)              // send new user to server
      .then(() => history.goBack())                  // then go back to user list
  return (
    <div>
      <h2>User Editor</h2>
      ...
      <button                                        // new Save button
        onClick={() => updateUser(user.id, user)}>   // sends updates to server
        Save
      </button>
    </div>
  )
}
```

To test, click on a user, change their password or last name, save and confirm the changes are saved to the database.

# Deliverables

As a deliverable, zip up your project and upload it to Canvas. Before you zip the whole directory, remove temporary directories that contain unnecessary files. Make sure to remove the target directory or any other build directories created by your IDE. Your code should contain the implementation for the following JavaScript files:

1. user-service.js
    a. createUser()
    b. findAllUser()
    c. findUserById()
    d. deleteUser()
    e. updateUser()
2. user-list.js

    a. findAllUser() event handler
    b. Add User button
3. user-edit-form.js
    a. Event handlers
        i. createUser()
        ii. findUserById()
        iii. deleteUser()
        iv. updateUser()
    b. Buttons
        i. Create
        ii. Save
        iii. Cancel
        iv. Delete