

# PROG2053: Project description

**Deadline:** The deadline for the project is the 26<sup>th</sup> of November (11:59 PM).

**Where and what to deliver:** You deliver the project as a .zip (package) file in Blackboard on the Assignment page. Include the link to a 2 min. video of your application in a `submission.txt` file included in your project. NOTE: Only the third part of the project will be delivered for evaluation.

For submission details see the Section “*Submitting the Project for Grading*” on page 15 of this document.

**Description:** The project consists of three component parts. The first two give you the skills and knowledge required to complete the third one which is the one that you will be delivering for evaluation.

*The project is taken from the Stanford course, CS142, with the permission of the course leader Mendel Rosenblum.*

## Installing MERN Software

The programming projects for this class require you to install Node.js and MongoDB. Follow the instructions below to install them.

### Installing Node.js

Install the latest "Long Term Support (LTS)" version of Node.js (currently version 14.16.1). It can be downloaded from the URL <https://nodejs.org/en/download>. To verify you have Node.js and its package manager (npm), try running the commands:

```
node -v
```

and:

```
npm -v
```

which should run and print out the version numbers of your node and npm programs.

### Installing MongoDB

Install the [MongoDB Community Edition](https://docs.mongodb.com/manual/administration/install-community/) from the website <https://docs.mongodb.com/manual/administration/install-community/>.

Once you start the MongoDB server using the command

```
mongod (the exact arguments depend on where you placed the database)
```

you should be able to directly interact with the MongoDB database by running the command:

```
mongo
```

Type help at the command prompt to see the available commands.

For Windows users, you may need to add the location of where MongoDB was installed to your environment path variable in order to run the commands. This is usually located at

`C:\Program Files\MongoDB\Server\<version_number>\bin`

## Project – First Part

### Setup

Install Node.js and the npm package manager on your system, following the installation instructions above.

Create a directory `projectFirstPart` and extract the contents of the attached zip file `firstPart_projectFiles.zip` into the directory. The zip file contains the starter files for this assignment.

This assignment requires many node modules that contain the tools (e.g. [Webpack](#), [Babel](#), [ESLint](#)) needed to build a [ReactJS](#) web application as well as a simple Node.js web server ([ExpressJS](#)) to serve it to your browser. These modules can be fetched by running the following command in the `projectFirstPart` directory:

```
npm install
```

That command will fetch around 650 node modules using around 150 megabytes of space into the subdirectory `node_modules`.

We can use npm to run the various tools we had it fetch. As can be seen in the "scripts" property of the `package.json` file, the following run commands are available:

- `npm run lint` - Runs ESLint on all the project's JavaScript files. Your code should run ESLint without warnings.
- `npm run build` - Runs [Webpack](#) using the configuration file `webpack.config.js` to package all of the projects JSX files into a single JavaScript bundle in the directory `compiled`.
- `npm run build:w` - Runs [Webpack](#) like the "run build" command except it invokes webpack with `--watch` so it will monitor the React components and regenerates the bundle if any of them change. This option is useful for development so changes made to components can be picked up by simply refreshing the browser to load the newly updated bundle. Otherwise you need to remember to run "npm run build" after every change. You might get a deprecation warning `[DEP_WEBPACK_WATCH_WITHOUT_CALLBACK]` that you can safely ignore.

Your solutions for all of the problems below should be implemented in the `projectFirstPart` directory.

This project uses [ReactJS](#), a popular framework for building web applications. The project's goal is to get you enough up to speed with ReactJS and the course's coding conventions that you will be able to build a web application with it in the next part of the project.

In order to fetch our web app via the HTTP protocol, we use a simple Node.js web server that can be started with the command from the `projectFirstPart` directory:

```
node webServer.js
```

All the files in the `projectFirstPart` can be fetched using an URL starting with <http://localhost:3000>. Click on <http://localhost:3000> to verify your web server is running. It should serve the file `index.html` to your browser.

We recommend you configure your development environment to run webpack in watch mode so you will need to run the node webserver and webpack when building and testing your project. You could do this by running the programs in different command line windows. Syntax errors get detected and reported by Babel so the output of webpack is useful. If you are running on a system with a unix-like shell like MacOS. The command:

```
node webServer.js & npm run build:w
```

runs the web server in background and the webpack in foreground within a single window.

On Windows you can start the web server in background and webpack in foreground with the two Windows commands:

```
start /B node webServer.js  
npm run build:w
```

You can stop the background webserver with the command:

```
taskkill /IM node.exe /F
```

## Getting Started

In this part of the project we require that you use the model, view, controller pattern described in class. There are many ways of organizing code under this pattern so we provide an example that both demonstrates some basic ReactJS features as well as showing the file system layout and module pattern we would like you to follow in your projects.

You should start by opening the example in your browser by navigating to the URL <http://localhost:3000/getting-started.html>. The page displays examples of ReactJS in action. The HTML in `getting-started.html` provides a div for ReactJS to draw the app into and a script tag include the app's JavaScript bundle `compiled/gettingStarted.bundle.js`. The webpack config file [webpack.config.js](#) directs that this bundle be created from the ReactJS file `gettingStarted.jsx`, a JSX program that renders the ReactJS component named `Example` into the div in `getting-started.html`.

To support reusable components, we adopt a file organization that co-locates the ReactJS component and its associated CSS stylesheet in a subdirectory of a directory named `components`.

The `Example` component is located in the files `components/example/{Example.jsx, Example.css}`.

You should look through the files invoked in the `getting-started.html` view (`getting-started.html`, `gettingStarted.jsx`, `components/example/{Example.jsx}`) since it shows the JavaScript and JSX statements needed to run a ReactJS web application along with explanatory comments. You should use this pattern and file naming convention for the other components you build for the class.

Model data is typically fetched from the webserver which retrieves the data from a database. To avoid having to set up a database for this project we will give you an HTML script tag to load the model data directly into the browser's DOM from the local file system. The models will appear in the DOM under the property name `cs142models`. You will be able to access it under the name `window.cs142models` in a ReactJS component.

### Problem 1: Understand and update the example view

You should look through and understand the `getting-started.html` view and the `Example` component. To demonstrate your understanding do the following:

1. Update the model data for the `Example` component to use your name rather than "Unknown name". You should find where "Unknown name" is and replace it.
2. Replace the contents of the `div` region with the class `motto-update` in the `Example` component with some JSX statements that displays your name and a short (up to 20 characters) motto. Like the user's name, the initial value for motto should come in with the model data. You must include some styling for this display in `Example.css`.
3. Extend the display you did in the previous step so it allows the user to update the motto being displayed. The default value should continue to be retrieved from the model data.

### Problem 2: Create a new component – US states view

Create a new component view that will display the names of all US states containing a given substring. Your view must implement an input field that accepts a substring. The view will display in alphabetical order a **list** of all states whose names contain the given substring (ignoring differences in case). For example, the view for the substring of "al" should list the states Alabama, Alaska, and California. The page should also display the substring that was used to filter the states. If there are no matching states then the web page should display a message indicating that fact (rather than just showing nothing). All states should be displayed when the substring is empty.

As in Problem #1 we provide you the model data with states. It can be accessed via `window.cs142models.states` after it is included with:

```
<script src="modelData/states.js"></script>
```

See `states.js` for a description of the format of the states data.

To help you get started and guide you to the file naming conventions we want you to use we provided a file `p2.html` that will load and display the bundle `compiled/p2.bundle.js`

which is generated by webpack from `p2.jsx` which displays the React component `States`. You can open this file in your browser via the URL <http://localhost:3000/p2.html>.

The files you will need to implement are:

- `components/states/States.jsx` - The ReactJS Component of your states component.
- `components/states/States.css` - Any CSS styles your component needs. **Include some styling for your state list here.**

### Problem 3: Personalizing the Layout

Create a ReactJS component named `Header` that will display a personalized header at the top of a view. Add this header to all ReactJS web apps in your assignment (`gettingStarted.jsx`, `p2.jsx`, `p4.jsx`, `p5.jsx`). Note that you **should not** replace the section from part 1 (your name and motto). That section should be separate from your header. Use your imagination and creativity to create a header that is "uniquely you". This can include additional images, graphics, whatever you like. You can extend the JSX/JavaScript in the components but you should rather not use external ReactJS Components or JavaScript libraries such as JQuery. Be creative!

The files you will need to implement are:

- `components/header/Header.jsx` - The ReactJS Component of your header component. This is defined as a class `Header` of type [React.Component](#).
- `components/header/Header.css` - Any CSS styles your component needs. **Include some styling for your header here.**

Note: `gettingStarted.jsx` should have a personalized header from Problem 3 at the top of the page and the section with the motto from Problem 1.2 right below it. All other page views (`p2.html`, `p4.html` and `p5.html`) should have your personalized header from Problem 3.

### Problem 4: Add dynamic switching of the views

Create a `p4.html` and a corresponding JSX file `p4.jsx` that includes both view components (the `Example` and `States` components). The `p4.jsx` needs to implement an ability to switch between the display of the two components. When a view is displayed there should be a button above it that switches to display the other view. For example, when the `States` view is displayed the button above it should read "Switch to Example," and when pushed the `States` should disappear and the `Example` view should be displayed.

For this problem you will need to create the files above as well as modify the webpack configuration file `webpack.config.js` to build a file `compiled/p4.bundle.js` that you can use in `p4.html` file. Note that if you are using Webpack with `--watch` (i.e. `npm run build:w`), you will need to restart it after changing code `>webpack.config.js`.

## Problem 5: Single page app

Although the approach taken in Problem 4 allows you to switch between the two views, it does not allow you to bookmark or share a URL pointing at a particular view. Even doing a browser refresh event causes the app to lose track of which view was being displayed.

We can address this deficiency by storing the view information into the URL. [React Router](#) provides this functionality for ReactJS. For this problem make a copy of your `p4.html` solution into a file named `p5.html` and copy your `p4.jsx` into a file named `p5.jsx`. Convert the code to use [React Router](#) to switch between the two component views. You may **style your toolbar-like control** (not just simple plain text links) that will allow the user to switch between the example and states component views.

Since this is the first extension from the core ReactJS we import, we're providing you with step-by-step instructions.

1. The project's `package.json` specifies `react-router` so the `npm install` command already fetched it for us. We do need to explicitly import it into our `p5.jsx` file. Add the following import line:

```
import { HashRouter, Route, Link } from "react-router-dom";
```

The line uses the JavaScript [import](#) statement to bring in the ReactJS components from React Router: [HashRouter](#), [Route](#), and [Link](#). The `HashRouter` module of React Router uses the fragment portion of the URL for storing information. So we can make `p5.html#/states` mark showing the States view while `p5.html#/example` specifies the Example component view.

2. The most common way of using React Router is to conditionally render the view we want based on the current URL. It is the [Route](#) component that implements this condition rendering when placed inside a `HashRouter` element like:

```
<HashRouter>
  ...
  <Route path="/states"
component={States} />
  <Route path="/example" component={Example}
/>
  ...
</HashRouter>
```

which would render the States component if the URL had `#/states` and the Example component if the URL had `#/example`.

3. Although we could use hyperlinks (i.e. `<a>` tags) to switch views `react-router` recommends using the `Link` component to generate the hyperlinks.

```
<Link to="/states">States</Link>
```

generates a hyperlink with `href="#/states"` and the strings "States" in it.

## Style

Your solutions should have proper MVC decomposition and should be styled as well. **Note that you should not directly manipulate the DOM in your code.** In addition, your code and templates must be clean and readable. Remember to run ESLint on your code. ESLint should raise no errors.

## Project – Second Part

In this project you will use [ReactJS](#) with [Material-UI](#) to create the beginnings of a photo-sharing web application. In the third part of this project, you'll also explore retrieving data from a server.

### Setup

You should already have installed Node.js and the npm package manager to your system. If not, follow the instructions at the start of this document (Installing MERN Software section).

Create a directory `projectSecondPart` and extract the contents of the attached file `secondPart_projectFiles.zip` into the directory. The zip file contains the starter files for this assignment.

This assignment requires many node modules that contain the tools (e.g. [Webpack](#), [Babel](#), [ESLint](#)) needed to build a [ReactJS](#) web application as well as a simple Node.js web server ([ExpressJS](#)) to serve it to your browser. It also fetches [Material-UI](#) which contain the React components and style sheets we will be using. These modules can be fetched by running the following command in the `projectSecondPart` directory:

```
npm install
```

[ReactJS](#) and [Material-UI](#) are fetched into the `node_modules` subdirectory even though we will be loading it into the browser rather than Node.js.

Like the previous assignment, we can use npm to run the various tools we had it fetch. The following npm scripts are available in the `package.json` file:

- `npm run lint` - Runs ESLint on all the project's JavaScript files. Your code should run ESLint without warnings.
- `npm run build` - Runs [Webpack](#) using the configuration file `webpack.config.js` to package all of the projects JSX files into a single JavaScript bundle in the directory `compiled`.
- `npm run build:w` - Runs [Webpack](#) like the "run build" command except it invokes webpack with `--watch` so it will monitor the React components and regenerates the bundle if any of them change.

Your solutions for all of the problems below should be implemented in the `projectSecondPart` directory. As was done with on the previous project you will need to run a web server we provide for you by running a command in your `projectSecondPart` directory:

```
node webServer.js
```

As in the previous project, you can use the command:

```
node webServer.js & npm run build:w
```

to run the web server and webpack within a single command line window.

## Problem 1: Create the Photo Sharing Application

As starter code for your PhotoApp we provide you a skeleton (`photo-share.html` which loads `photoShare.jsx`) that can be started using the URL ["http://localhost:3000/photo-share.html"](http://localhost:3000/photo-share.html). The skeleton:

- Loads a [ReactJS](#) web application that uses [Material-UI](#) to layout a Master-Detail pattern. It has a header made from a [Material-UI App Bar](#) across the top, places a `UserList` component along the side, and has a content area beside it with either a `UserDetail` or `UserPhotos` components.
- Uses the [React Router](#) to enable deep linking for our single page application by configuring routes to three stubbed out components:
  1. `/users` is routed to the component `UserList` in `components/userList/`
  2. `/users/:userId` is routed to the component `UserDetail` in `components/userDetail/`
  3. `/photos/:userId` is routed to the component `UserPhotos` in `components/userPhotos/`

See the use of [HashRouter](#), and [Route](#) in `photoShare.jsx` for details. For the stubbed out components in `components/*`, we provide an empty CSS file and a simple render function that includes some description of what it needs to do and the model data to use.

For this problem, we will continue to use our magic `cs142models` hack to provide the model data so we display a pre-entered set of information. As before, the models can be accessed using `window.cs142Models`. The schema of the model data is defined below.

Your assignment is to extend the skeleton into a working web app operating on the fake model data. Since the skeleton is already wired to either display components `UserList`, `UserDetail`, and `UserPhotos` with the appropriate parameters passed by React Router, most of the work will be implementing the stubbed out components. They should be filled in so that:

- `components/userList` component should provide navigation to the user details of all the users in the system. The component is embedded in the side bar and should provide a list of user names so that when a name is clicked, the content view area switches to display the details of that user.
- `components/userDetail` component is passed a `userId` in the `props.match` by React Router. The view should display the details of the user in a pleasing way along with a link to switch the view area to the photos of the user using the `UserPhotos` component.
- `components/userPhotos` component is passed a `userId`, and should display all the photos of the specified user. It must display all of the photos belonging to that user. For each



photo you must display the photo itself, the creation date/time for the photo, and all of the comments for that photo. For each comment you must display the date/time when the comment was created, the name of the user who created the comment, and the text of the comment. The creator for each comment should be a link that can be clicked to switch to the user detail page for that user.

Besides these components, you need to update the `TopBar` component in `components/topBar` as follows:

- The left side of the `TopBar` should have your name.
- The right side of the `TopBar` should provide app context by reflecting what is being shown in the main content region. For example, if the main content is displaying details on a user the toolbar should have the user's name. If it is displaying a user's photos it should say "Photos of " and the user's name.

The use of `ReactRouter` in the skeleton we provide allows for deep-linking to the different views of the application. Make sure the components you build do not break this capability. It should be possible to do a browser refresh on any view and have it come back as before. Our standard approach to building components handles deep-linking automatically. Care must be taken when doing things like sharing objects between components. A quick browser refresh test on each view will show when you broke something.

Although you don't need to spend a lot of time on the appearance of the app, it should be neat and understandable. The information layout should be clean (e.g., it should be clear which photo each comment applies to).

### Photo App Model Data

For this problem we keep the magic DOM loaded model data we used in the previous project. The model consists of four types of objects: `user`, `photo`, `comment`, and `SchemaInfo` types.

- Photos in the photo-sharing site are organized by user. We will represent users as an object `user` with the following properties:
  - `_id`: The ID of this user.
  - `first_name`: First name of the user.
  - `last_name`: Last name of the user.
  - `location`: Location of the user.
  - `description`: A brief user description.
  - `occupation`: Occupation of the user.The DOM function `window.cs142models.userModel(user_id)` returns the `user` object of the user with id `user_id`. The DOM function `window.cs142models.userListModel()` returns an array with all `user` objects, one for each the users of the app.
- Each user can upload multiple photos. We represent each photo by a `photo` object with the following properties:
  - `_id`: The ID for this photo.

`user_id`: The ID of the `user` who created the photo.  
`date_time`: The date and time when the photo was added to the database.  
`file_name`: Name of a file containing the actual photo (in the directory `projectSecondPart/images`).  
`comments`: An array of the `comment` objects representing the comments made on this photo.

The DOM function `window.cs142models.photoOfUserModel(user_id)` returns an array of the `photo` objects belonging to the user with id `user_id`.

- For each photo there can be multiple comments (any user can comment on any photo). `comment` objects have the following properties:

`_id`: The ID for this comment.  
`photo_id`: The ID of the `photo` to which this comment belongs.  
`user`: The `user` object of the user who created the comment.  
`date_time`: The date and time when the comment was created.  
`comment`: The text of the comment.

- For testing purposes we have `SchemaInfo` objects have the following properties:

`_id`: The ID for this `SchemaInfo`.  
`__v`: Version number of the `SchemaInfo` object.  
`load_date_time`: The date and time when the `SchemaInfo` was loaded. A string.

## Problem 2: Fetch model data from the web server

After doing Problem 1, our photo sharing app front-end is looking like a real web application. The big barrier to be considered real is the fakery we are doing with the model data loaded as JavaScript into the DOM. In this Problem we remove this hack and have the app fetch models from the web server as would typically be done in a real application.

The `webServer.js` given out with this project reads in the `cs142Models` we were loading into the DOM in Problem 1 and makes them available using [ExpressJS](#) routes. The API exported by `webServer.js` uses HTTP GET requests to particular URLs to return the `cs142Models` models. The HTTP response to these GET requests is encoded in JSON. The API is:

- `/test/info` - Returns `cs142models.schemaInfo()`. This URL is useful for testing your model fetching method.
- `/user/list` - Returns `cs142models.userListModel()`.
- `/user/:id` - Returns `cs142models userModel(id)`.
- `/photosOfUser/:id` - Returns `cs142models.photoOfUserModel(id)`.

You can see the APIs in action by pointing your browser at above URLs. For example, the links ["http://localhost:3000/test/info"](http://localhost:3000/test/info) and ["http://localhost:3000/user/list"](http://localhost:3000/user/list) will return the JSON-encoded model data in the browser's window.

To convert your app to fetch models from the web server you should implement a `FetchModel` function in `lib/fetchModelData.js`. The function should be declared as follows:

```

/*
 * FetchModel - Fetch a model from the web server.
 *   url - string - The URL to issue the GET request.
 * Returns: a Promise that should be filled
 * with the response of the GET request parsed
 * as a JSON object and returned in the property
 * named "data" of an object.
 * If the requests has an error the promise should be
 * rejected with an object contain the properties:
 *   status: The HTTP response status
 *   statusText: The statusText from the xhr request
 */

```

Although there many modules that would make implementing this function trivial, we want you to learn about the low-level details of AJAX. You may not use other libraries to implement FetchModel; you must write Javascript code that creates XMLHttpRequest DOM objects and responds to their events.

Your solution needs to be able to handle multiple outstanding FetchModel requests. To demonstrate your FetchModel routine works, your web application should work so that visiting `http://localhost:3000/photo-share.html` displays the version number returned by sending an AJAX request to the `http://localhost:3000/test/info` URL. The version number should be displayed in the TopBar component of your app.

After successfully implementing the FetchModel function in `lib/fetchModelData.js`, you should modify the code in

- `components/userDetail/UserDetail.jsx`
- `components/userList/UserList.jsx`
- `components/userPhotos/UserPhotos.jsx`

to use the FetchModel function to request the data from the server. There should be no accesses to `window.cs142models` in your code and your app should work without the line in `photo-share.html`:

```
<script src="modelData/photoApp.js"></script>
```

## Style

Your problem solutions should have proper MVC decomposition. In addition, your code and components should be clean and readable, and your app must be at least "reasonably nice" in appearance and convenience.

Note that we are using [Material-UI](#), React components that implement Google's [Material Design](#). We have used Material-UI's [Grid component](#) to layout the Master-Detail pattern, and a [AppBar](#) header for you. Although you don't need to build a fully Material Design compatible app, we recommend you to use [Material-UI](#) components when possible.

In addition, remember to run ESLint before submitting. ESLint should raise no errors.

## Extra problems

The `userPhotos` component specifies that the display should include all of a user's photos along with the photos' comments. This approach doesn't work well for users with a large numbers of photos. As an extra challenge, you can implement a photo viewer that only shows one photo at a time (along with the photo's comments) and provides a mechanism to step forward or backward through the user's photos (i.e. a stepper).

If you would like to challenge yourself, your solution should:

- Introduce the concept of "advanced features" to your photo app. On app startup "advanced features" is always disabled. The toolbar on the app should have a checkbox labelled "Enable Advanced Features" that displays the current state of "advanced features" (checked meaning advanced features is enabled) and supports changing the enable/disable state of the advanced features.
- Your app should use the original photo view unless the "advanced features" have been enabled by the checkbox. If enabled, viewing the photos of a user should use the single photo with stepper functionality.
- The user interface for stepping should be something obvious and the mechanism should indicate (e.g. a disabled button) if stepping is not possible in a direction because the user is at the first (for backward stepping) or last photo (for forward stepping).
- Your app should allow individual photos to be bookmarked and shared by copying the URL from the browser location bar. The browser's forward and back buttons should do what would be expected. When entering the app using a deep linked URL to individual photos the stepper functionality should operate as expected.

## Project – Third Part (to be delivered)

In this project you will start up a database system and convert your Photo Sharing App you built in the Second Part of the Project to fetch the views' models from it. We provide you a new `webServer.js` supporting the same interface as the Second Part of the Project's web server but it also establishes a connection to a database. This allows you to make your app into a legitimate *full stack* application.

### Setup

You should have MongoDB and Node.js installed on your system. If not, follow the instructions at the start of this document (Installing MERN Software section).

#### **\*IMPORTANT!\***

The setup for this Third Part is different from the previous parts. You start by making a copy of your `projectSecondPart` directory files into a directory named `projectThirdPart`. Into the `projectThirdPart` directory extract the contents of the attached file `thirdPart_projectFiles.zip`. This zip file will overwrite the files `package.json`, `webServer.js`, `.eslintrc.json`, and `index.html` and add several new files and

directories. In the unlikely event you had made necessary changes in any of these files in your `projectSecondPart` directory you will need to reapply the changes after doing the unzip.

Once you have the `projectThirdPart` files, fetch the dependent software using the command:

```
npm install
```

For this we will be running all three tiers of the web application (browser, web server, database) on your local machine.

## Start and initialize the MongoDB database

Once you have installed MongoDB and created the directory for the database as described in the instructions at the start of this document (Installing MERN Software section), you can start MongoDB by running the command:

```
mongod (the exact arguments depend on where you placed the database)
```

Since this command doesn't return until the database is shutdown you will want to either run it in a separate window or as a background process (e.g. `mongod (args) &` on Linux/macOS).

Once the MongoDB server is started you can load the photo app data set by running the command:

```
node loadDatabase.js
```

This program loads the fake model data from previous projects (i.e. `modelData/photoApp.js`) into the database. Since our app currently doesn't have any support for adding or updating things you should only need to run `loadDatabase.js` once. The program erases whatever is in the database before loading the data set so it is safe to run multiple times.

We use the [MongooseJS](#) Object Definition Language (ODL) to define a [schema](#) to store the photo app data in MongoDB. The schema definition files are in the directory `schema`:

- `schema/user.js` - Defines the User collection containing the objects describing each user.
- `schema/photo.js` - Defines the Photos collection containing the objects describing each photo. It also defines the objects we use to store the comments made on the photo.
- `schema/schemaInfo.js` - Defines the SchemaInfo collection containing the object describing the schema version.

These files are loaded both into the `loadDatabase.js` program where they are used to create the database and the `webServer.js` where they are used to access the database. Note: The object schema stored in the database is similar to but necessarily different from the `cs142models` JavaScript objects used in the previous assignment. Familiarize yourself with these schema definitions.

## Start the Node.js web server

Once you have the database up and running you will need to start the web server. This can be done with the same command as the previous assignments (e.g. `node webServer.js`). Start your web server with the command from your `projectThirdPart` directory:

```
node webServer.js
```

If you use the above command, remember to **restart the web server after each change you make to the server code**. You can also use `nodemon`, which will watch for any changes to the server code and automatically restart the web server:

```
nodemon webServer.js
```

After updating your Photo Share App with the new files from `projectThirdPart` and starting the database and web server make sure the app is still working before continuing on to the assignment.

## Problem 1: Convert the web server to use the database (40 points)

The `webServer.js` we give you in this project is like the one in the Second Part of the Project `webServer.js` in that the app's model fetching routes use the magic `cs142models` rather than a database. Your job is to convert all the routes to use the MongoDB database. There should be no accesses to `cs142models` in your code and your app should work without the line:

```
var cs142models = require('./modelData/photoApp.js').cs142models;
```

in `webServer.js`. Note that any `console.log` statements in `webServer.js` will print to the terminal rather than the browser.

### Web Server API

As in the Second Part of the Project the web server will return JSON encoded model data in response to HTTP GET requests to specific URLs. We provide the following specification of what URLs need to be supported and what they should return. Your web server should support the following model fetching API:

- **/test** - Return the schema info (`/test/info`) and object counts (`/test/counts`) of the database. This interface is for testing and as an example for you, we provide an implementation that fetches the information from the database. You will not have to change this one.
- **/user/list** - Return the list of users' models appropriate for the navigation sidebar list. Since we anticipate a large numbers of users, this API should only return an array of the user properties needed by the navigation side bar (`_id`, `first_name`, `last_name`). It replaces the `cs142models.userListModel()` call in the provided code.
- **/user/:id** - Return the detail information of the user with `_id` of `id`. This should return the information we have on the user for the detail view (`_id`, `first_name`, `last_name`, `location`, `description`, `occupation`) and replaces the

`cs142models userModel()` call. If something other than the id of a User is provided the response should be an HTTP status of 400 and an informative message.

- **/photosOfUser/:id** - Return the photos of the user with `_id` of `id`. This call generates all the model data needed for the photos view including all the photos of the user as well as the comments on the photos. The photos properties should be (`_id`, `user_id`, `comments`, `file_name`, `date_time`) and the comments array elements should have (`comment`, `date_time`, `_id`, `user`) and only the minimum user object information (`_id`, `first_name`, `last_name`). This replaces the `cs142models.photoOfUserModel()` call. If something other than the id of a User is provided the response should be an HTTP status of 400 and an informative message. Note this API will need some assembling from multiple different objects in the database. The assignment's `package.json` file fetches the [async](#) module to make the assembling the multiple photos easier.

To help you make sure your web server conforms to the proper API we provide a test suite in the sub-directory `test`. **Please make sure that all of the tests in the suite pass before submitting.** See the Testing section below for details.

Your GET requests do not return exactly the same thing that the `cs142models` functions return but they do need to return the information needed by your app so that the model data of each view can be displayed with a single `FetchModel` call. You will need to do subsetting and/or augmentation of the objects coming from the database to build your response to meet the needs of the UI. For this assignment you are not allow to alter the database schema in anyway.

### **\*IMPORTANT!\***

Implementing these Express request handlers requires interacting with two different "model" data objects. The Mongoose system returns [models](#) from the objects stored in MongoDB while the request itself should return the data models needed by the Photo App views. Unfortunately since the Mongoose models are set by the database schema and front end models are set by the needs of the UI views they don't align perfectly. Handling these requests will require processing to assemble the model needed by the front end from the Mongoose models returned from the database.

Care needs to be taken when doing this processing since the models returned by Mongoose are JavaScript objects but have special processing done on them so that any modifications that do not match the declared schema are tossed. This means that simply updating a Mongoose model to have the properties expected by the front end doesn't work as expected. One way to work around this is to create a copy of the Mongoose model object. A simple way of doing the copy is to translate the model into JSON and back to an JavaScript objects. The following code fragment does this object cloning:

```
JSON.parse(JSON.stringify(modelObject));
```

by taking `modelObject` converting into a JSON string and then converting it back to a JavaScript object, this time without the methods and special handling done on Mongoose models.

## Testing

Testing a full web application is challenging. In the directory `test` we provide a test of just the backend portion of your application. The test uses [Mocha](#), a popular framework for writing Node.js tests. To setup the test environment, from inside the `test` subdirectory do an `npm install` to fetch Mocha and all the related dependencies. Once you have done this, you can run the test by running the command inside the `test` directory: `npm test`.

The `npm test` command runs the file `test/serverApiTest.js` which is a program written in the Mocha language (e.g. `describe()` and `it()`) testing the three Photo App backend URLs (`/user/list`, `/user/ID`, `/photosOfUser/ID`). **In order to be reasonably sure that the functionality of the backend routes conforms to spec, please check that all our provided tests pass before submitting.**

## Style

Your project should have proper MVC decomposition. In addition, your code and templates must be clean and readable, and your app must be at least "reasonably nice" in appearance and convenience.

In addition, your code and templates must be clean and readable. Remember to run `npm run lint` before submitting. The linter should raise no errors.

## Deliverables

Use the standard class [submission mechanism](#) to submit the entire `projectThirdPart` directory.

## Submitting the Project for Grading

### TL;DR

Upload your project as a [zip file](#) containing your cleaned project directory on the Assignment page in Blackboard. The zip file should contain a single directory named `projectN` where `N` is the group number.

### Cleaning up before submitting

Please delete any unnecessary files from your project directory before submitting. The web tools used by the projects can generate hundreds of megabytes worth of files in your project directory that we don't want and you won't be able to submit. The contents of the following directories contain generated files that can safely be deleted since we can regenerate them while evaluating your project:

- `node_modules` - Contains the modules fetched by npm based the specification in `package.json` file.
- `compiled` - Contains the bundled JavaScript product by the React.js tool chain.



## Zip your project directory

By having you submit the entire project directory we can be sure to get all the files and subdirectories you add or modify when building your project. This includes the multiple hidden files (i.e. files that start with a period) that are not normally displayed. You can create a zip file of the project directory either from the graphical user interface or a terminal using the zip command.

From the Finder program on MacOS you can control-click on the project folder and select the "Compress" option on the pop-up menu. On Windows, you can right click on the folder, select "Send to..." and "Compressed (zipped) folder."

From a shell running in a terminal program with the current working directory being the directory/folder containing your project directory, run the command:

```
zip -r projectN.zip projectN
```

where `projectN` is the name of your project directory.

Both of the above approaches generated a ".zip" file you can upload into Canvas.

**IMPORTANT: For a PASS grade you will also need to make sure your code is free of any lint warnings and passes the provided test suite.**

## Video

In addition you must submit a short video tour of your photo app you implemented in the last part of the project. The video length should be no longer 2 minutes. This should be a simple screen recording with audio. On Mac OSX you can use Quicktime and on Windows/Linux you can use VLC to take screen recordings. Please do not spend too much time creating the video. You will not be graded on production quality.

Videos will likely be too large for our submission process so we need you to provide us a link in a `submission.txt` file (that you include in your project directory). Uploading the video to a video sharing website such as [YouTube](#) or [Vimeo](#) is an easy way of getting such a link.