

COMP3322A Modern Technologies on World Wide Web

Assignment 2 (20%)

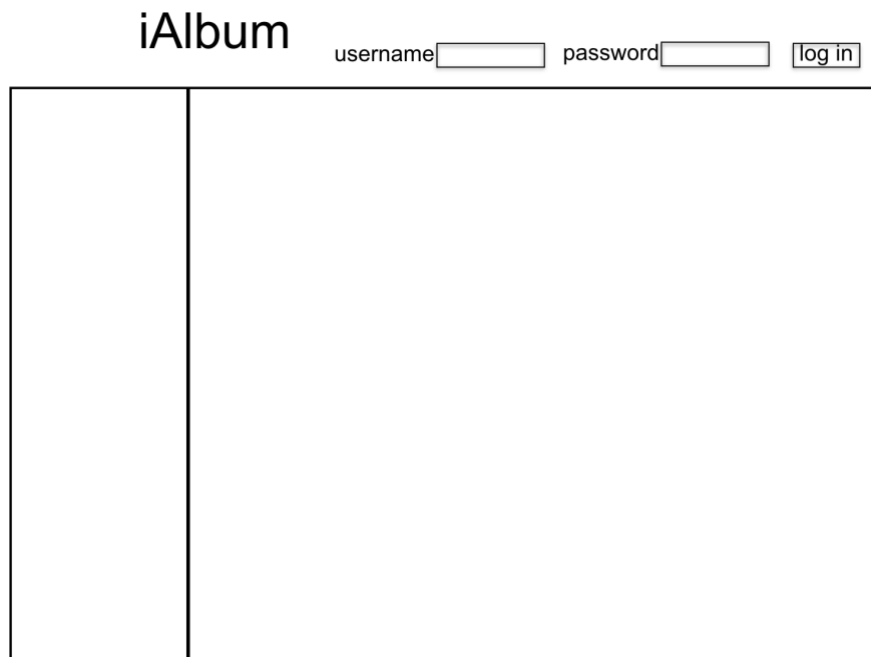
[Learning Outcomes 1, 2]

Due by: 23:55, Sunday Dec 15 2019

Overview

In this assignment, we are going to develop a simple single-page social photo sharing application **iAlbum** using the MERN stack (MongoDB, Express.JS, ReactJS and Node.js). The main work flow of **iAlbum** is as follows.

- Upon loading, the sketch of the page is shown in Fig. 1



The sketch shows a web page layout. At the top left is the text "iAlbum". To its right are two input fields labeled "username" and "password", followed by a "log in" button. Below this header is a large rectangular area divided into two vertical sections: a narrow left sidebar and a wider main content area.

Fig. 1

- After a user has logged in, the sketch of the page is in Fig. 2. A list of friend albums is shown in the left division.

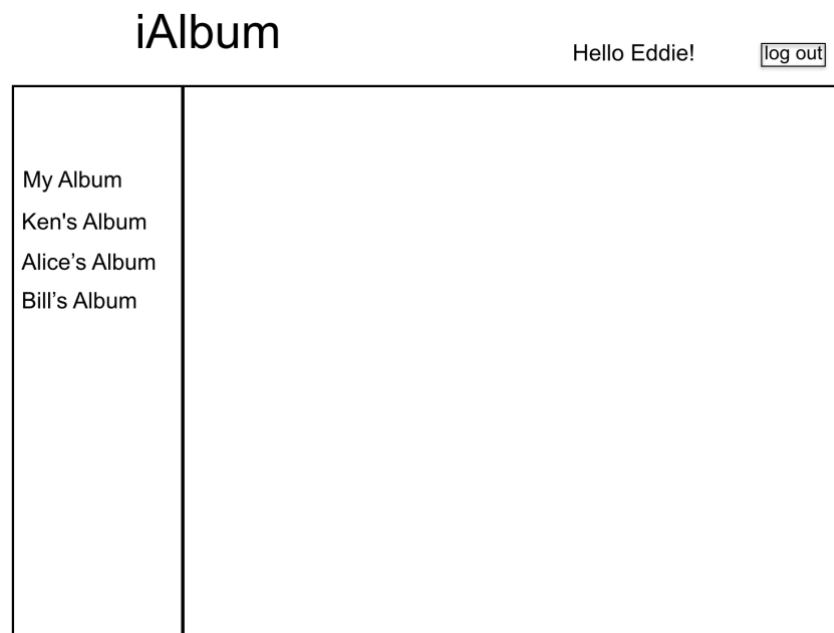


Fig. 2

- After clicking “My Album”, the sketch of the page is in Fig. 3. The photos in the user’s own album is displayed in the right division, together with messages of who liked a photo, deletion buttons, and new photo selection/upload buttons.

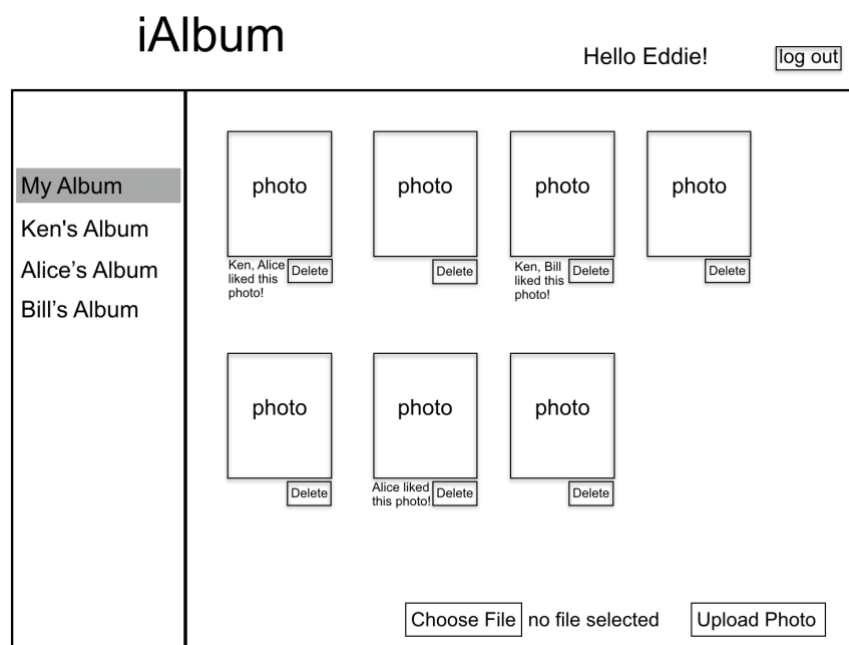


Fig. 3

- Each photo is clickable. After clicking a photo, the photo will be enlarged as in Fig. 4, together with the “liked” message and the deletion button. When the cross is clicked, the page returns to the view shown in Fig. 3.

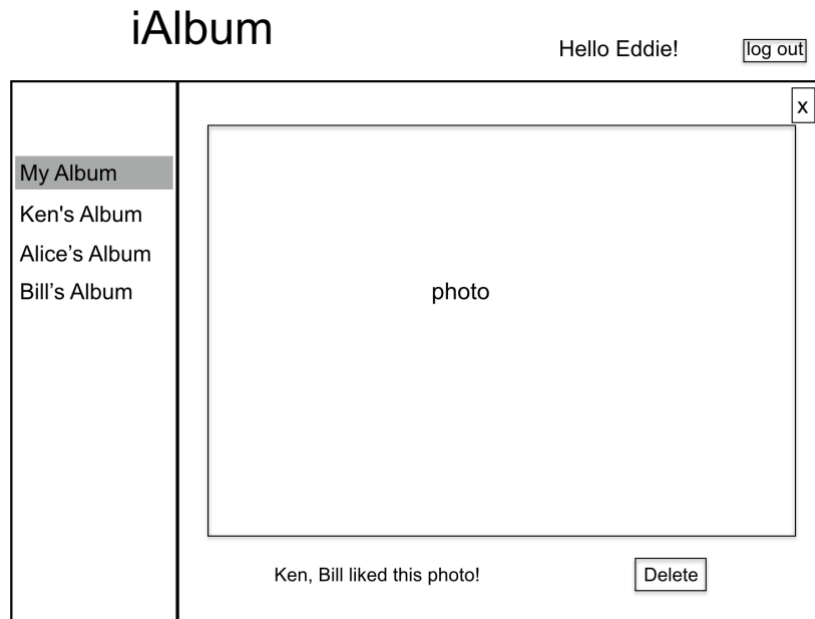


Fig. 4

- When a friend's album is selected in the left-hand list, the page view becomes one in Fig. 5. Photos of the friend are shown in the right division, together with messages of who liked a photo and the like buttons.

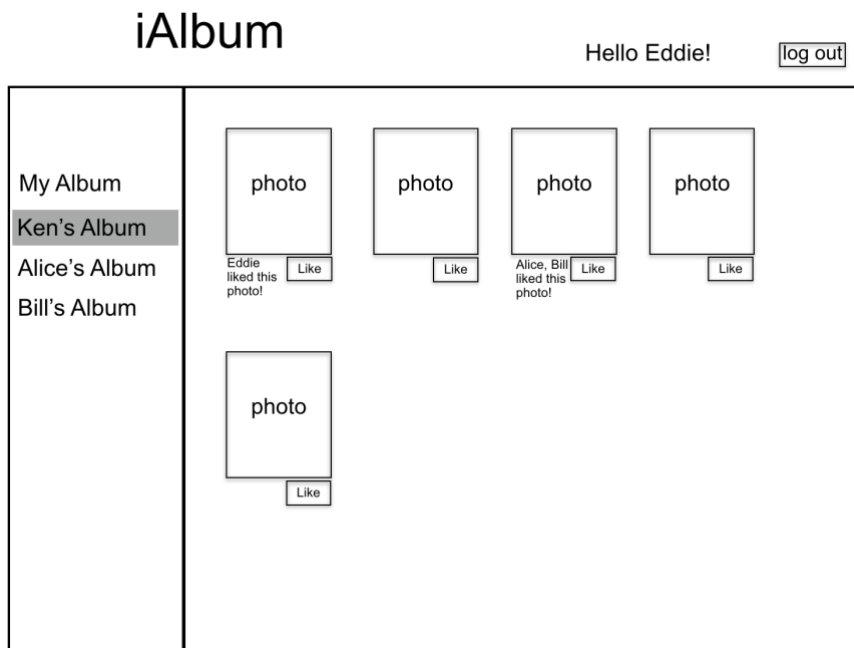


Fig. 5

- Similarly, when a photo is clicked, the photo will be enlarged as in Fig. 6, together with the "liked" message and the like button. When the cross is clicked, the page returns to the view shown in Fig. 5.

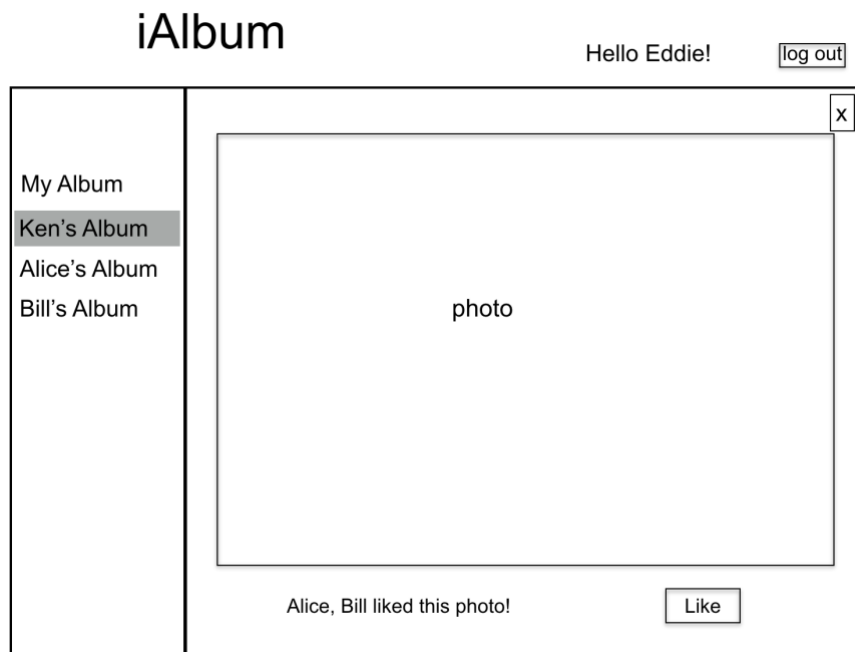


Fig. 6

We are going to achieve this application by implementing code in a backend Express app and a frontend React app.

- Express app:
[app.js](#)
[./routes/ albums.js](#)
- React app:
[./src/App.js](#)
[./src/index.js](#)
[./src/App.css](#)

Task 1. Backend Web Service

We implement the backend web service logic using Express.js. The web service is accessed at <http://localhost:3002/xx>.

Preparations

1. Following steps in Lab 6, install the Node.js environment and the Express framework, create an Express project named **AlbumService**, and install dependencies for MongoDB/monk.
2. Following steps in Lab 6, install MongoDB, run MongoDB server, and create a database "[assignment2](#)" in the database server.

Insert a number of records to a [userList](#) collection in the database in the format as follows.

We will assume that user names are all different in this application.

```
db.userList.insert({'username': 'Eddie', 'password': '123456', 'friends': ['Ken', 'Alice', 'Bill']})
```

Create a folder “**uploads**” under the **public** folder in your project directory. Copy a few photos to the **uploads** folder (due to our simplified implementation to be outlined later, we will only use .jpg photo files in this assignment). Insert a few records to a **photoList** collection in the database in the format as follows, each corresponding to one photo you have copied to the **uploads** folder. Here **userid** should be the value of **_id** generated by MongoDB for the photo owner’s record in the **userList** collection, which you can find out using **db.userList.find()**.

```
db.photoList.insert({'url': 'http://localhost:3002/uploads/1.jpg', 'userid': 'xxxxxxx', 'likedby': ['Ken', 'Alice']})
```

Note that copying photos to **uploads** folder and inserting photo records into the **photoList** collection, as we do above, are only for testing purpose. We will implement the functionality of uploading photos and inserting their records to the database using **iAlbum**.

For implementation simplicity, in this assignment, we do not store uploaded photos in MongoDB. Instead, we store them in the harddisk under the **./public/uploads/** folder, and store the path of a photo in the **photoList** collection only, using which we can identify the photo in the **uploads** folder.

Task 1. Implement backend web service logic (AlbumService/app.js,

AlbumService/routes/albums.js)

app.js (10 marks)

In **app.js**, load the router module implemented in **./routes/albums.js**. Then add the middleware to specify that all requests for **http://localhost:3002/** will be handled by this router.

Add necessary code for loading the MongoDB database you have created, creating an instance of the database, and passing the database instance for usage of all middlewares.

Also load any other modules and add other middlewares which you may need to implement this application.

Add the middleware to serve requests for static files in the **public** folder.

We will let the server run on the default port 3002 and launch the Express app using command “node app.js”.

./routes/albums.js (32 marks)

In **albums.js**, implement the router module with middlewares to handle the following endpoints:

1. **HTTP GET requests for <http://localhost:3002/init>.** The middleware checks if the “userID” cookie has been set for the client. If not, send an empty string back to the client. Otherwise, retrieve **username** of the current user (according to the value of the “userID” cookie), and **username** and **_id** of friends of the current user from the **userList** collection in the database; send all retrieved information as a JSON string to the client if the database operation is successful, and the error message if failure. You should decide the format of the JSON string and retrieve data accordingly in the client-side code to be implemented in Task 2.
2. **HTTP POST requests for <http://localhost:3002/login>.** The middleware should parse the body of the HTTP POST request and extract the username and password carried in request body. Then it checks whether the username and password match any record in the **userList** collection in the database. If no, send “Login failure” in the body of the response message. If yes, set a cookie with the key of “userID”, the value of this user record’s **_id**, and a maxAge of 1 hour; send the **username** and **_id** of all friends of the logged-in user as a JSON string in the body of the response message if database operation is successful and the error message if failure. Again, you should design the format of the JSON string.
3. **HTTP GET requests for <http://localhost:3002/logout>.** The middleware should unset the “userID” cookie, and send an empty string back to the user.
4. **HTTP GET requests for <http://localhost:3002/getalbum/:userid>.** If the userid in the URL is “0”, the middleware should retrieve **_id**, **url** and **likedby** array of all photos of the current user from the **photoList** collection in the database, based on the userid stored in the “userID” cookie. Otherwise, retrieve **_id**, **url** and **likedby** array of all photos of the friend from the **photoList** collection based on the userid in the URL. Send all retrieved information as a JSON string in the body of the response message if database operation is successful, and the error message if failure. You should decide the format of the JSON string to be included in the response body.
5. **HTTP POST request for <http://localhost:3002/uploadphoto>.** Store the photo carried in the request body in **./public/uploads/** folder and insert a record of the photo in the **photoList** collection in the database.

To read and write files in the Express/Node.js framework, you will need the **fs** module (<https://nodejs.org/api/fs.html>). Load the module in **albums.js**. To store the received photo in a **path** (e.g., **./public/uploads/1.jpg**), a simple approach is to use **req.pipe(fs.createWriteStream(path));** .

Refer to https://nodejs.org/api/fs.html#fs_fs_createwritestream_path_options for **fs.createWriteStream()** and

https://nodejs.org/api/stream.html#stream_readable_pipe_destination_options for pipe().

Since in our simplified client-side implementation (detailed in Task 2), the client does not send the file name of the uploaded photo but only the image file, you can create a name of your choice for the newly uploaded photo. For example, you can generate a random number `x` and name the file `x.jpg` (recall that we assume only `.jpg` files are uploaded). Then the photo record to be inserted to the `photoList` collection should be `{'url': 'http://localhost:3002/uploads/x.jpg', 'userid': 'xxxxxxx', 'likedby':[]}`, where `userid` is the value of the “`userID`” cookie.

The middleware should send a JSON string containing the `_id` and `url` of the photo in the `photoList` collection to the user if success, and the error message if failure.

6. **HTTP DELETE requests for** <http://localhost:3002/deletephoto/:photoid>. The middleware should delete the photo record from the `photoList` collection, corresponding to the `_id` value in the URL, and remove the photo file from `./public/uploads/` folder (**Hint**: you may find `fs.unlink()` useful https://nodejs.org/api/fs.html#fs_fs_unlink_path_callback). Return an empty string to the client if success and the error message if failure.

7. **HTTP PUT requests for** <http://localhost:3002/updatelike/:photoid>. The middleware should update the `likedby` array of the photo record in the `photoList` collection, corresponding to the `_id` value in the URL. Especially, use `_id` of the current user in the “`userID`” cookie to find out the `username` of the current user from the `userList` collection, and then use the `db.collection.update` method to update the photo record in the `photoList` collection. Return the updated `likedby` array to the client if success and the error message if failure.

8. Add the code for enabling CORS in `albums.js`, similar to what you did in Lab 8.

Task 2 Front-end React App

Implement the front-end as a React app. The app is accessed at `http://localhost:3000/`.

Preparations

Following steps in Lab 8, create a React app named **MyApp** and install the jQuery module in the React app.

Implement the React app (MyApp/src/index.js,

MyApp/src/App.js, MyApp/src/App.css)

index.js (3 marks)

Modify the generated **Index.js** in the **./src** folder of your react app directory, such that it renders the component you create in **App.js** in the “root” division in the default **index.html**.

App.js (40 marks)

App.js should import the jQuery module and link to the style sheet **App.css**.

Design and implement the component structure in **App.js**, such that the front-end page views and functionalities as illustrated in Figures 1-6 can be achieved. You can decide the components to create, in order to implement the React app.

Hints:

1. You can use conditional rendering to decide what component to be displayed on the page view. For example, in the *root* component you are creating in **App.js**, you may use a state variable to indicate if a cookie has been set for the current user (i.e., the user has logged in) or not, and then render the component presenting the respective page view accordingly. Initialize the state variable to indicate that no cookie has been set, and update it when the user has successfully logged in and logged out, respectively.
2. **Page load.** When the browser loads the page from the React App, send an AJAX HTTP GET request for <http://localhost:3002/init>: (1) if an empty string is received from the server, a page as shown in Fig. 1 should be displayed; (2) if a JSON string carrying username of the logged-in user and `_id` and username of the friends is received, a page as shown in Fig. 2 should be displayed. Especially, the username of the logged-in user is to be shown in the message “Hello username!” in the header area.
3. **Log in.** When handling the click event on the “log in” button, check if both the username and password input textboxes are non-empty: if so, send an AJAX HTTP POST request for <http://localhost:3002/login>, carrying the input username and password in the request body; otherwise, show an alert popup box stating “You must enter username and password”. If a “Login failure” response is received, the page view remains the one in Fig. 1, except for displaying “Login failure” in the header area in addition. If a JSON string containing the username and `_id` of friends is received, display the page as shown in Fig. 2.
4. **Log out.** When handling a click event on the “log out” button, send an AJAX HTTP GET request for <http://localhost:3002/logout>. Display the page view as in Fig. 1 when an empty string is received from the server.
5. When handling a click event on “My Album” in the album list, send an AJAX HTTP GET request for <http://localhost:3002/getalbum/0>. When the JSON string containing `_id`, url and likedby array of the current user’s photos is received, display the photos following the sketch in Fig. 3. Especially, each photo is an `` element with a “src”

attribute pointing to the url of the respective photo, and an “id” attribute recording _id of the photo. If the likedby array of a photo is not empty, display a message “xx liked the photo!” underneath the photo; otherwise, do not display the message. Display a “Delete” button under each photo. You can decide the display size of the photos and the number of photos to display in each row on the page. Add a file upload input element (http://www.w3schools.com/jsref/dom_obj_fileupload.asp) and an “Upload Photo” button at the bottom of the right division. Note that the texts to display on and behind the file upload input button may differ depending on the browser you use to render the file upload input element.

6. When handling a click event on a friend’s “xx’s Album” entry in the album list, send an AJAX HTTP GET request for <http://localhost:3002/getalbum/userid>, where the **userid** is the _id of the friend in the userList collection in the database. When the JSON string containing _id, url and likedby array of the friend’s photos is received, display the page view as shown in Fig. 5. Note that the album entry in the list in the left division, whose photos are being displayed in the right division, should be highlighted.
7. When handling a click event on the “Upload Photo” button, check if a file has been selected with the file upload input element (**Hint:** you may find “files” property of the `input` `FileUpload` object useful http://www.w3schools.com/jsref/dom_obj_fileupload.asp). If no, do nothing. If yes, send an AJAX HTTP POST request for <http://localhost:3002/uploadphoto>, enclosing the file object as a whole in the body of the request message. If a JSON string containing the _id and url of the uploaded photo is received, display the new photo as the last photo on the page, together with a “Delete” button. If an error message is received, show the error message in an alert box.
8. When handling a click event on a “Delete” button on the page view in Fig. 3, a confirmation box will be popped up, showing the message “Are you sure you want to delete this photo?”. If “Cancel” is clicked, the box disappears and the previous page view remains; if “OK” is clicked, send an AJAX HTTP DELETE request for <http://localhost:3002/deletphoto/photoid>, where the **photoid** should be _id of the photo in the photoList collection in the database. Upon receiving an empty string in the response, the photo together with the “liked” message and the “Delete” button should be removed from the page, and the following photo entries should be moved forward to occupy its space, if it is not the last photo. If an error message is received, show the error message in an alert box.
9. When handling a click event on a “Delete” button on the page view in Fig. 4, similar event handling as the above should be implemented, except for the following: upon receiving an empty string in the response, the page view in Fig. 3 should be displayed, where the photo entry corresponding to the deleted photo is removed and the following photo entries are moved forward to occupy its space (if it is not the last

photo).

10. When handling a click event on a “Like” button (either on the page view in Fig. 5 or on the page view in Fig. 6), send an AJAX HTTP PUT request for <http://localhost:3002/updatelike/photoid>, where `photoid` should be `_id` of the photo in the `photoList` collection in the database. Upon receiving the updated `likedby` array in the response, display the updated “liked” message underneath the photo on the page. If an error message is received, show the error message in an alert box.
11. When handling a click event on a photo on the page view in Fig. 3 or Fig. 5, display the photo together with the “liked” message, the “Delete” or the “Like” button, and the “x” in the right division (Fig. 4 or Fig. 6). When “x” is clicked, the page view returns to Fig. 3 or Fig. 5. An AJAX HTTP GET request should be sent for <http://localhost:3002/getalbum/0> or <http://localhost:3002/getalbum/userid> (depending on whose album entry is highlighted in the left division), and response should be handled similarly as in points 5 and 6 above, to retrieve the photo information for display.

App.css (10 marks)

Style your page views nicely using CSS rules in **App.css**.

Other marking criteria:

(5 marks) Good programming style (avoid redundant code, easy to understand and maintain). You are encouraged to provide a **readme.txt** file to let us know more about your programs.

Submission:

You should zip the following files (in the indicated directory structure) into a **yourstudentID-a2.zip** file

[AlbumService/app.js](#)
[AlbumService/routes/albums.js](#)
[MyApp/src/App.js](#)
[MyApp/src/index.js](#)
[MyApp/src/App.css](#)

and submit the zip file on Moodle:

- (1) Login Moodle.
- (2) Find “Assignments” in the left column and click “Assignment 2”.
- (3) Click “Add submission”, browse your .zip file and save it. Done.
- (4) You will receive an automatic confirmation email, if the submission was successful.
- (5) You can “Edit submission” to your already submitted file, but ONLY before the deadline.