**Lab Week 2**

You are required to build a web application for our unit FIT2095. It must include the following (but not limited to):

1. An HTTP server that serves clients requests on port 8080. **(4m)**
2. index.html that contains two hyperlinks to two different pages: **(2m)**
   1. Assessments.html: display the four assessments we have in the unit.
      1. example: <http://localhost:8080/assessments>
   2. Topics.html: display the 12 weeks topics
      1. example: <http://localhost:8080/topics>
3. 404.html: display an error message for unknown requests **(1m)**
   1. example: <http://localhost:8080/testerror>
4. a URL endpoint that accepts three parameters (day, month, year) and responds with: **(3m)**
   1. The current week since the first day of semester 2 which is the 26th of July 2021.
   2. An error message if the input date is before the first day of the semester.
   3. An error message if the input date is after week 14.
   4. example:<http://localhost:8080/whichweek/?d=5&m=8&y=2020>

All four pages must have a link to the home page (index.html)

Note: You should not use Express.js

**You can use the following function for task 4.**

1. /\*\*
2. \*
3. \* @param {day} d
4. \* @param {month} m
5. \* @param {year} y
6. \* @returns week number since August 3,2020; returns -1 if the input is before 3rd of August 2020
7. \*/
8. function getDaysDiff(d, m, y) {
9. let returnValue = -1;
10. let currentDay = new Date();
11. currentDay.setDate(parseInt(d));
12. currentDay.setMonth(parseInt(m) - 1); // months start from 0
13. currentDay.setYear(parseInt(y));
14. let firstDay = new Date("8/3/2020"); // first day in semester 2
15. **if** (currentDay >= firstDay) {
16. var diffDays = parseInt((currentDay - firstDay) / (1000 \* 60 \* 60 \* 24)); //gives day difference
17. returnValue = (Math.floor(diffDays / 7) + 1);
18. }
19. **return** (returnValue);
20. }

**Lab Week 3**

You are required to build a web application that represents a bookstore management system.

Specifications are:

1. The database is implemented as an array of items (books)
2. It uses Express.js as a middleware
3. Each item (book) has five attributes: id, title, author, topic,cost
4. the id for the new items is auto-generated.
   * HINT: use Math.random() method.
   * Example: let newId= Math.round(Math.random()\*1000)
5. it offers the following operations:
   * add a new book to the bookstore through the URL.
     + E.g: http://localhost:8080/addbook?title=Harry Potter&author= J. K. Rowling&topic=fiction&cost=15
     + Object {id:789, author:’J. K. Rowling’, topic:’fiction’,cost:15} should be saved in the database
   * List all books
     + The output should have five columns: id, title, author, topic, cost
       - HINT: use function **generateList** as a reference
       - URL: http://localhost:8080/getallbooks
   * delete a book by id
     + URL: http://localhost:8080/deleteid/938, where 938 is the id of the book that should be deleted
   * get bookstore total value:
     + URL: http://localhost:8080/getbookstorevalue
     + where the bookstore value is equal to *[Math Processing Error]∑0nbook.cost* for all n items in the array DB

**Bonus:  use express.Router() to separate the routes in a different file.**

## **Lab  Week 4**

Develop a simple bookstore management system, such that:

1. It has at least five pages  
   1. Homepage
   2. New Book
   3. List Books
   4. Invalid Data
   5. 404 (Page not found)
2. The home page shows a logo (i.e. an image of your choice) at the header of the page.
3. The home page must have two links to two pages: add a new book and list books (HINT: [*refer to static assets section*](https://www.alexandriarepository.org/module/week-4-advanced-express-js/#staticassetssec))
4. add new book page has a form that sends a POST request to the server with the following items: 
   * 1. title
     2. author
     3. topic
     4. cost
   1. The server should not accept a title, topic or author name with less than 3 characters and a negative cost. If any of the conditions cannot be met, the server should respond with the 'invalid data' page.
5. If invalid data arrives at the server, the server must respond with the invalid page (fourth page) that shows an error message (or image)
6. The list book page displays all the books in a table format (HINT: [*refer to this app*](https://www.alexandriarepository.org/module/week-4-advanced-express-js/#tablerenderingex))
7. Change the background of the pages into any colour you prefer using a CSS file  (HINT: [*refer to static assets section*](https://www.alexandriarepository.org/module/week-4-advanced-express-js/#staticassetssec))
8. The server should respond with a 404 page for any wrong/invalid pathname (i.e. URL). The error message on the 404 page should be an image.
9. Users must be able to navigate through the app pages using hyperlinks placed in the header of each page.

## **Week5: Lab tasks**

Develop a simple bookstore management system, such that:

1. It should have the ability to add new books, update, delete and list the available books.
2. It uses MongoDB to store all the entries instead of arrays
3. Each book must have the following fields:  
   1. title
   2. author
   3. topic
   4. date of publication
   5. summary
4. Pages  
   1. add new book: adds a new document (i.e. new book) to the DB
   2. Get all books: shows all the books in a table format
   3. Delete all books by a topic: the page takes a topic as input and deletes all documents from the DB for that given topic
   4. Update a book by title: the page takes a title and all other fields as input. The server must update all the fields for that given title
5. redirect the client to the get all books page after the insert, update and delete operations.

Prefered Specs:

* Add background to all pages
* Style your pages' links, colours, and fonts.

## **Week6: Lab Tasks**

Develop an application that is responsible for managing a clinic and its patients. The app must include the following features:

1. It uses Mongoose to store all the data instead of plain MongoDB
2. It has two collections: Doctors and Patients
3. Each Doctor has the following fields (Doctor Schema):
   1. full name: an object has
      1. first name: cannot be empty
      2. last name
   2. date of birth (date)
   3. Address: Object has
      1. State: min 2, max 3 characters
      2. Suburb
      3. Street
      4. Unit
   4. numPatients: a positive number represents the number of patients a doctor has seen so far
4. Each Patient has the following fields (Patient Schema):
   1. full name (string): cannot be empty
   2. doctor (ObjectID): \_id of the patient's doctor
   3. age (number): a number between 0 and 120
   4. date of visit (date): default to the current date
   5. case description (string): a string with at least 10 characters
5. Pages
   1. Insert a new Doctor: adds a new doctor to ‘doctors’ collection
   2. Get all doctors page: shows all the doctors in a table format **(including the \_id field)**
   3. Insert new patient page: adds a new patient to the Patient collection. (Hint: Get the Doctor’s \_id from the  ‘Get all doctors page’ manually (copy&paste)). The server has to increment (update) the numPatients of that doctor by one.
   4. Get all patients page. This page must show all the patients in a table format including the first and last names of their doctors.
   5. Delete patient by fullName: the page takes a fullName as input and deletes first patient with specified fullName from the DB.
   6. Update Doctor numPatients by \_id: the page takes as input the doctor's \_id and the numPatients. It sets the new number of patients to the doctor with the given \_id.
   7. Invalid Data: if an error occurs, redirect the user to invalid data page.
6. Add a logo for the home page
7. Change the background of all pages using a CSS file
8. Redirect the client to the get all patients page after the insert, update and delete operations.
9. Deploy your application to a VM in your GCP account.

## **Lab Week 7**

Implement and run the Movie Library application ([see week 7 material](https://www.alexandriarepository.org/module/week-7-create-a-restful-application/)) and add the following features:

1. Delete a movie by its ID
2. Delete an actor by its ID and all its movies from the 'Movie' collection.
3. Remove a movie from the list of movies of an actor
   * Example: http://localhost:8080/actores/1234/987
   * where 1234 is the actor's ID
   * and 987 is the movie's ID
4. Remove an actor from the list of actors in a movie
   * Example: http://localhost:8080/movies/567/2234
   * where 567 is the movie ID
   * and 2234 is the actor ID
5. Add an existing actor to the list of actors in a movie
6. Retrieve (GET) all the movies produced between year1 and year2, where year1>year2.
7. The current implementation of getAll actors function retrieves the list of actors, where each actor has an array of IDs that represents his/her movies.  Update the implementation such that the array of movies should contain the details of the movies instead of IDs.
8. Like point (7), reimplement getAll movies such that it retrieves the details of all actors for each individual movie.
9. Delete all the movies that are produced between two years. The two years (year1 & year2) must be sent to the backend server through the request's body in JSON format.
10. Deploy your app into a VM instance in your GCP account.

***Note: No front-end (HTML) is required for this lab***

## **Lab Week 8**

#### **Note: This lab has no backend (Node.js) side.**

Using Angular, develop a frontend app that represents a clinic management system. In this week, you will develop a page that is responsible for entering the doctors' details. Each entry (doctor) consists of:

1. Unique ID
2. First Name
3. Last Name
4. Date of Birth
5. Address:
   * Suburb
   * State:  one of (ACT, NSW, NT, QLD, SA, TAS, WA, VIC)
   * Post Code
6. Number of Patients

Features

1. The frontend app uses an array as a database to hold doctors data
2. Doctors data should be displayed in a table such that each row ends with a button to delete the doctor's entry
3. Button 'ADD Doctor' saves a doctor and shows the total number of available doctors
4. Button 'Delete Docs without Patients' deletes all the doctors with zero patients. The button must show the number of doctors with 0 patients and it should be disabled if there are no such doctors.

LAB Week 9

Implement and run the application provided in [Week 9 Angular II article](https://www.alexandriarepository.org/module/week-9-angular-js-ii/) and add the following new requirements:

1. a new section to add a movie
2. a new section to delete a movie by **aTitle**
   1. the new section has to have one text box to enter the value of **aTitle**
3. a new section to delete all movies produced between **aYear1 and aYear2**  
   1. the new section has to have two text boxes to enter two integer values that represent **aYear1 and aYear2**
4. a new section to add an actor to a movie  
   1. the section has to show the list of available actors to select one of them
   2. and, the list of available movies to select one of them
   3. a button to insert the selected actor to the selected movie
   4. NOTE: can be verified via Compass
5. a new section to list all the available movies
6. Deploy your application to a VM in your GCP account

**Lab 10**

Run the source code provided in [Week 10 article](https://www.alexandriarepository.org/module/week-10-single-page-application-using-mean-stack/)and implement the following requirements:

1. a new component to add a new movie
2. a new component to delete a movie
3. a new component to list all movies
4. a new component to add an actor to a movie  
   1. the section has to show the list of available actors to select one of them
   2. and, the list of available movies to select one of them
   3. a button to insert the selected actor into the selected movie
   4. NOTE: can be verified via Compass
5. a new component that shows 'View not Found' if the requested URL doesn't match any paths for routes defined earlier in the configuration.
6. Add a pipe that transforms the actor's year of birth into age. The age must be shown as a column in the list actors template
7. Make your app installable (PWA)
8. deploy the app to your GCP account

## **Lab Week 11**

In this week, you will develop a ticketing manager for the Australian Football league AFL. In this application, the server sends to all the connected clients a text and a set of teams. The client, then, selects one of the teams and the number of tickets they want to purchase and click on the 'send' button to send their data back to the server. The server counts the number of tickets for each team and broadcasts the updated results to all the connected clients. The clients and server are connected bi-directionally in real-time using socket.io. The clients must show the results in a pie or bar chart.

The backend server should have the following duties (4m):

* Maintain the teams' object
* For each new connection, the server must respond with the teams' object
* For each new purchase that arrives:
  + add the number of tickets to the team's counter
  + broadcast the updated object (or the updated counter) to all the connected clients (i.e. emit to all).

The frontend duties (6m):

* Sends a Socket.io connection request
* Listens and receives teams objects
* Extracts the text, teams' names and their counters.
* Dynamically build and display the text and the list of teams
* Sends the number of the tickets along with the selected team to the backend server using socket.io
* Use the chart.js package to plot the results in a graphical way.
* shows the total number of tickets sold so far

Sample of the teams object that can be implemented at the server side

**let** teamsObj = {

theText: "Select your team and enter the number of Tickets:",

teams: [

{ text: "Melbourne", value: 0, count: 0 },

{ text: "Port Adelaide", value: 1, count: 0 },

{ text: "Geelong Cats", value: 2, count: 0 },

{ text: "Brisbane Lions", value: 3, count: 0 },

{ text: "Western Bulldogs", value: 4, count: 0 },

{ text: "Sydney Swans", value: 5, count: 0 },

{ text: "GWS Giants", value: 6, count: 0 },

{ text: "Essendon", value: 7, count: 0 },

],

};

## **Week 12: Lab Tasks**

#### **Lab Week 12**

Synthesize and Translate Chat Application

Design and implement an application that converts clients messages in a chat application to speech in English and translates it to a target language.

1. the clients and the server should communicate using socket.io (1m)
2. a client sends a text and the target language to the server via a socket.io event (1m)
3. the server translates the text to the target language (2m), then converts the text to English speech (2m)
4. the server saves the mp3 file that is generated by Google service locally. (2m)
5. the server then emits the translation back to all clients using a socket.io event (1m)
6. the server should be able to serve multiple clients concurrently (1m)

Lab resources:

* Google Translate: <https://github.com/googleapis/nodejs-translate>
* Google Text-To-Speech: <https://www.alexandriarepository.org/module/week-11-real-time-bidirectional-event-based-communication/>
* Bootstrap forms: <https://getbootstrap.com/docs/5.1/forms/overview/>
* HTML 5 Audio Player: <https://www.w3schools.com/html/html5_audio.asp>
* HTML Audio/Video load: <https://www.w3schools.com/tags/av_met_load.asp>
* randomstring package that can be used to generate random names for the mp3 files: <https://www.npmjs.com/package/randomstring>