# **Full stack development – Activity 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OVERVIEW** | | | | |
| In this activity, will build a full stack application using the MERN stack. They will be involved in writing the code for both components (back end and front end) and will have some time at the end to either change the style of the application or add new functionalities to it.  **Curriculum competencies**  **(Taken from (BC Curriculum) –** [**Applied Design, Skills, and Technologies**](https://curriculum.gov.bc.ca/curriculum/adst)**;** [**Mathematics**](https://curriculum.gov.bc.ca/curriculum/mathematics); [**BC Science**](https://curriculum.gov.bc.ca/curriculum/science)**)** | | | | |
| **Topic** | **Grade Level** | **Cost** | **Time** | **Complexity** |
| - Computer Science | Grades 8-12, all girls | $0.00/students | Intro: 00 min  Project: 00 min | Preparation: 3/5  Execution: 4/5  1-5 (5 = hardest) |
| **Comments from a past instructor:** Put in your pitch about how great this activity was! Or something that should be done differently if someone wants to resurrect it. | | | | |

|  |  |  |
| --- | --- | --- |
| **MATERIALS** | | |
| **Non-Consumable** | **Consumable** | **Other** |
| - Laptop (1 per pair)  - 1 Laptop for instructors   * Must have Node.js, Mongo db, VSCode, and Mongo db for VSCode installed in the system. |  |  |
| **PREPARATION** | | |
| Ensure that all laptops (Windows 10) have the following installed:   * Node.js (installation guide: <https://www.youtube.com/watch?v=__7eOCxJyow>)   + My node.js version is: v18.18.0 * Mongo db (installation guide: <https://www.youtube.com/watch?v=gB6WLkSrtJk>)   + My mongo db version is: db version v7.0.1 * VSCode (installation guide: <https://www.youtube.com/watch?v=CPmQwlycfGI>)   + My VSCode version is: 1.82.3 * Mongo db for VSCode (installation guide: <https://www.youtube.com/shorts/8MyvgOCb7wE>) | | |

|  |  |
| --- | --- |
| **PROCEDURE** | |
| **10-15 min**  **25 min**  **40-55 min**  **15-25 min** | 1. **Understand the structure of the folder and its sub-folders, which parts contribution to the front end and which contribute to the back end.** 2. **Write the code for the back end of the application.** 3. **Write the code for the front end of the application.** 4. **Deploy the application and test it out.** |
| **Extend** | Allow the participants to experiment with different style choices and add a new sort functionality based on ‘sort by year of manufacture’. |
| **SAFETY** | |
| General internet/computer safety | |
| **TALKING POINTS AND RESOURCES** | |
| **Activity) Building a full stack application**  *Purpose: In this hands-on activity, participants use the MERN stack to develop a car inventory application.*  **Activity Logistics**   * Place students in groups of 2. * Each group has a laptop with VS Code, MongoDB, and Node.js installed.   **Script + Procedure**   * The target user for this application is a car salesman whose duties would include:   + Maintaining a car inventory where you store the speed of each car and sort cars by their speed   + Crossing off cars that have been purchased   + Removing cars from the inventory once they leave the dealership * In a full-stack application, the organization of folders and subfolders can vary depending on the project's structure and technology stack. However, there are some common conventions that are often followed, and the use of "api" and "client" subfolders is one such convention:   + ‘api` Subfolder:     - The "API" subfolder typically contains server-side code or backend-related files.     - It is where you would typically find code responsible for handling HTTP requests, business logic, database interactions, and other server-side operations.     - Backend technologies like Node.js, Express.js, Ruby on Rails, Django, or other server frameworks are commonly used within the "api" subfolder.     - This is where you define your API routes, controllers, models, and any server-related functionality.   + `client` Subfolder:     - The "client" subfolder is usually reserved for client-side code or frontend-related files.     - It contains code responsible for rendering the user interface (UI), handling user interactions, making HTTP requests to the server API, and managing the presentation layer of your application.     - Technologies like React, Angular, Vue.js, HTML, CSS, and JavaScript are commonly used in the "client" subfolder.     - This is where you typically find components, views, templates, stylesheets, and frontend assets.   **Note: Always save a file (Ctrl + S) after editing it.**  Back end development:   * First, we work on the code in api/models/Car.js. We start by importing the 'mongoose' library using the `require('mongoose')` statement. This library is essential for working with MongoDB. * After importing mongoose, we create a Schema object by assigning it the value of `mongoose.Schema`. The Schema object is a critical component of Mongoose as it defines the structure of our data models. Think of it as a blueprint for how the documents in our MongoDB collection should be organized. By creating a Schema, we can specify the fields, their data types, and any validation rules that should be applied to the data when it is stored in the database. This structured approach ensures that our data adheres to a predefined format and helps maintain consistency in our application's data storage and retrieval processes.      * Next, we define a schema for a "todo" document that will be stored in a MongoDB database. To do this, we create a carSchema object using the previously imported mongoose.Schema. This schema specifies the structure of our "todo" documents. Each document is expected to have four fields: carName, choose, speed, and timestamp. * The carName field is defined as a String type and is marked as required, meaning that every "todo" document must have a value for carName. * The choose field is of Boolean type and has a default value of false. If this field is not explicitly provided when creating a "todo" document, it will default to false. * The speed field is defined as a Number type and, like carName, is also marked as required. * Finally, the timestamp field is set as a String type with a default value of Date.now(). This means that if the timestamp field is not specified when creating a "todo" document, it will automatically be populated with the current date and time.      * Finally, we create a model for our "Car" model using the mongoose.model method. This model is constructed based on the previously defined carSchema, which specifies the structure and validation rules for "Car". By creating this model, we're essentially defining how data for "Car" objects should be stored in the MongoDB database. The first argument to mongoose.model is the name of the model, which, in this case, is 'Car,' and the second argument is the schema (carSchema) that defines its structure. * Once the "Car" model is created, we export it using module.exports. This export statement makes the "Car" model available for use in other parts of the application, allowing us to perform various operations on the "Car" collection in the database, such as creating, reading, updating, and deleting data. This separation of concerns and modularization of code is a common practice in Node.js applications, promoting maintainability and code organization.      * Next, we go to api/server.js. Here, we're setting up the server-side of our application using Node.js and Express.js. To start, we import the necessary libraries and modules. First, we bring in the Express.js framework using `const express = require('express')`, which will help us create a web server and handle HTTP requests. Then, we import Mongoose, which is essential for connecting and interacting with a MongoDB database, using `const mongoose = require('mongoose')`. Finally, we incorporate the CORS (Cross-Origin Resource Sharing) middleware using `const cors = require('cors')`, which is crucial for handling cross-origin requests and allowing our server to communicate with clients from different origins. * After importing these dependencies, we create an Express application instance with `const app = express()`. This instance is the core of our web server and will handle incoming HTTP requests and responses. * To ensure our server is capable of processing JSON data, we set up middleware using `app.use(express.json())`. This middleware parses incoming JSON request bodies, making it easier to work with JSON data sent from clients. Additionally, we enable CORS support by using `app.use(cors())`, which allows our server to respond to requests from different domains, an important feature for client-side web applications that may run on separate servers or domains.      * Now, we are establishing a connection to a MongoDB database and importing a data model. First, we utilize Mongoose, which we previously imported, to connect to MongoDB. The mongoose.connect() method is called with the URL of the MongoDB database as its first argument, in this case, "mongodb://127.0.0.1:27017/cars." This URL specifies the location and name of the database we want to connect to. Additionally, we provide some configuration options as an object, including useNewUrlParser: true and useUnifiedTopology: true, which enable specific features and settings for the database connection. Once the connection is established, the code logs a success message to the console, indicating that it's "Connected to DB!" * Furthermore, we import the "Car" data model from the 'models' directory using const Car = require('./models/Car'). This step is crucial for our application to interact with the MongoDB database effectively. The "Car" model represents the structure and schema of documents stored in the 'cars' collection within the database. It enables us to create, read, update, and delete car data in a structured and organized manner. By importing the model, we can use it to perform database operations throughout the application.      * Next, we define routes and their corresponding request handlers for our Express.js application. These routes determine how the server should respond to specific HTTP requests. * First, we have a GET request route defined with app.get('/cars', async (req, res) => {...}). This route is designed to handle requests for retrieving a list of all cars. When a client makes a GET request to the '/cars' endpoint, the code asynchronously queries the MongoDB database using Car.find(), which retrieves all car documents from the 'cars' collection. Once the data is fetched, it's sent as a JSON response using res.json(cars), providing the client with a list of cars in JSON format. * Then, we have a POST request route defined with app.post('/car/new', (req, res) => {...}). This route handles requests to create a new car. When a client sends a POST request to '/car/new', the code extracts the car name and speed information from the request body. It then creates a new instance of the "Car" model with this data and saves it to the database using car.save(). Finally, the newly created car's data is sent back to the client as a JSON response with res.json(car), confirming the successful creation of the new car in the database. These route definitions are essential for providing the API endpoints necessary to interact with the car data in our application.      * Next, in this section of the code, we are defining additional routes and their corresponding request handlers in our Express.js application. * First, we have a DELETE request route defined with app.delete('/car/delete/:id', async (req, res) => {...}). This route is responsible for handling requests to delete a car from the database based on its unique ID. When a client sends a DELETE request to '/car/delete/:id', where :id represents a dynamic parameter containing the car's ID, the code uses Car.findByIdAndDelete(req.params.id) to find the car by its ID and remove it from the database. The result of this operation is then sent as a JSON response to the client, providing information about the deleted car. * Then, we have a GET request route defined with app.get('/cars/sortCarsBySpeed', async (req, res) => {...}). This route is designed to retrieve all cars from the database and sort them by speed in descending order. When a client makes a GET request to '/cars/sortCarsBySpeed', the code fetches all car documents from the 'cars' collection and uses the .sort('-speed') method to arrange them in descending order based on their speed property. The sorted list of cars is then sent back to the client as a JSON response. These route definitions enable clients to delete specific cars by ID and retrieve a sorted list of cars by their speed, enhancing the functionality of our car inventory application.      * Lastly, we define a GET request route, /car/choose/:id, and its associated request handler in our Express.js application. This route is responsible for toggling the 'choose' property of a car based on its unique ID. * First, we extract the car's ID from the request parameters using const carId = req.params.id. This ID is used to identify the specific car we want to update. * Inside a try-catch block, we attempt to find the car in the database using Car.findById(carId). If the car with the given ID exists, we retrieve it. However, if the car does not exist, we create a new entry with the provided ID, car name, and speed. This ensures that the car with the given ID is either updated or created as needed. * Next, we toggle the 'choose' property of the car using car.choose = !car.choose and save the updated car object to the database with await car.save(). This operation changes the 'choose' property's value from true to false or vice versa. * Finally, we send a JSON response containing the updated car object back to the client. In case of any errors during this process, we catch and log the error and respond with a 500 status code and an error message. This route allows clients to toggle the selection status of a car by its ID, providing a user-friendly way to mark cars as chosen or unchosen in our car inventory system.      * In this final part of the code, we are instructing our Express.js application to start the server and listen for incoming network requests on a specified port, which is 3001 in this case. We use the app.listen() method to achieve this. When the server is successfully started and begins listening on port 3001, a callback function is executed, and a message is logged to the console with the statement "Server is running on port 3001!". * This code is essential for the server to be operational and accessible to clients. By specifying the port number as 3001, we define where clients can send their HTTP requests to interact with our server. This code effectively "opens the doors" to our server, allowing it to handle incoming requests, route them to the appropriate request handlers, and send responses back to clients. The message logged to the console serves as a confirmation that the server is up and running, providing information to developers that the server is active and ready to process requests on port 3001.     Font end development:   * Now, for front end, we first look into App.js which is in the src folder of client. * First, we import two key elements from the 'react' library: useEffect and useState. These are React hooks that enable us to manage side effects and state within functional components. useEffect allows us to perform side effects in our components, such as data fetching, and useState allows us to manage and update the state of our component. * Additionally, we define a constant variable api\_base, which serves as the base URL for our API. This URL, in this case, is set to 'http://localhost:3001', indicating that our application is expecting to communicate with an API running on the local machine at port 3001. This base URL is crucial for making API requests to the server and is used as a prefix when constructing API endpoint URLs within our React application.      * Now, we are defining a functional component called App within a React application. Within this component, we utilize React's state management features. * First, we initialize several state variables using the useState hook. These variables include cars, which will hold an array of car data retrieved from the API; popupActive, a boolean indicating whether a popup for adding a new car is active; newCarName, which stores the name of a new car to be added; and newSpeed, which stores the speed of the new car. * The useEffect hook is employed to perform side effects within the component. In this case, when the component mounts (thanks to the empty dependency array []), it triggers the GetCars function, which sends an HTTP GET request to the API base URL concatenated with '/cars'. This request fetches a list of cars from the server, and the response data is then processed and used to update the cars state variable with the retrieved car data.      * Next, we define a function called chooseCar within our React component. This function is responsible for toggling the "choose" property of a car in response to a user action. * When called, chooseCar takes the id of the car as an argument. It then uses the fetch function to send a GET request to the API base URL concatenated with '/car/choose/' and the id parameter. This request is intended to toggle the "choose" property of the specified car on the server. * Once the server processes the request and returns the updated car data, the code uses the await keyword to asynchronously retrieve and parse the response data using res.json(). This data represents the modified car object with the "choose" property toggled. * Subsequently, the code updates the cars state variable by mapping over the current array of cars. For each car, it checks if its \_id matches the \_id in the retrieved data. If there's a match, it updates the "choose" property of that specific car with the new value from data. This ensures that the client-side state is synchronized with the server's data.      * Next, we define a function called `addCar` within our React component. This function is responsible for adding a new car to the inventory when invoked. * The function starts by using the `fetch` function to send an HTTP POST request to the API endpoint `api\_base + "/car/new"`. This endpoint is responsible for creating a new car in the server's database. The request includes a JSON payload containing the car's name (`newCarName`) and speed (`newSpeed`) provided by the user. * The request configuration includes the HTTP method set to "POST" and specifies that the content type of the request body is JSON with the `headers` option. * Once the server processes the request and successfully creates the new car, it responds with the updated car data, which includes a unique identifier (`\_id`) generated by the server. * The code uses the `await` keyword to asynchronously retrieve and parse the response data as JSON using `res.json()`. This data represents the newly created car object, including its `\_id`. * The `setCars` function is then used to update the `cars` state variable. It spreads the current array of cars (`[...cars]`) and adds the newly created car (`data`) to the end of the array. This ensures that the client-side state reflects the addition of the new car. * After adding the new car, several state variables are reset: `setPopupActive(false)` closes the popup for adding a new car, and `setNewCarName("Default Car Name")` and `setNewSpeed("100")` reset the input fields for the new car name and speed to their default values, preparing the form for the next car addition.      * Now, we define the last 2 important functions within our React component: * The `sortCarsBySpeed` function is responsible for sorting the cars by their speed property in descending order. It first sends an HTTP GET request to the API endpoint `api\_base + '/cars/sortCarsBySpeed'`. This endpoint is designed to retrieve the cars from the server and sort them by speed on the server side. Upon receiving the response, which contains the sorted car data, it updates the `cars` state variable with the sorted data using `setCars`. If any errors occur during this process, it catches and logs the error. * The `deleteCar` function is used to delete a car from the inventory based on its unique ID. It sends an HTTP DELETE request to the API endpoint `api\_base + '/car/delete/' + id`, where `id` is the identifier of the car to be deleted. Once the server processes the request and deletes the car, it responds with the deleted car's data. The code then updates the `cars` state variable by filtering out the deleted car from the array, ensuring that it no longer appears in the client-side inventory.      * The rest of the code in this file is complete, so you do not have to add anything to them. * We start by rendering a container div with the class name "App," which acts as the root element of our application. * Inside this div, we have an <h1> element displaying the title "Car Inventory," providing a clear heading for the application. * Following that, an <h4> element appears with the text "Your cars," serving as a subheading to indicate the section's purpose. * A button with the class name "sort-cars-by-speed-button" is displayed, which, when clicked, triggers the sortCarsBySpeed function to sort the cars by speed. * The car inventory itself is displayed as a collection of car items, each represented by a <div> element with the class name "car." These car items contain details such as the car's name, speed, and a checkbox. The class name is conditionally modified with "is-chosen" based on the car's "choose" property, which visually indicates whether the car has been selected. * When a car item is clicked, it invokes the chooseCar function to toggle the "choose" property of the car, allowing users to select or deselect cars interactively. * An "x" icon is provided in each car item, allowing users to delete a car from the inventory when clicked, triggering the deleteCar function. * A conditional check ensures that if there are no cars in the inventory (determined by cars.length > 0), a message is displayed informing the user that they currently have no cars in their inventory.      * Additionally, a button with the class name "addPopup" displays a "+" icon. When clicked, it activates a popup for adding a new car, setting the popupActive state to true. * The conditional rendering of the popup is controlled by the popupActive state. If popupActive is true, a popup dialog is displayed, allowing users to input the name and speed of a new car. * The popup includes input fields for the car's name and speed, along with a "New Car" button that triggers the addCar function when clicked. This function adds a new car to the inventory based on the provided information and resets the input fields and popup state to their initial values.      * Now, we move onto the final part of our full stack application - the styling! * In this segment of code, we define custom CSS variables and reset default styling for elements in a web application's styling. Here's an explanation of the code: * The :root selector is used to define custom CSS variables. These variables are typically used to store reusable values such as colors, fonts, or spacing that can be applied throughout the application. For example, --primary and --secondary are defined with specific color values, making it easy to maintain a consistent color scheme across the application. * After defining the custom variables, the \* selector is employed to reset default styling for all elements. This is a common practice to ensure a consistent starting point for styling. The following CSS properties are modified: * margin and padding are set to 0, removing any default spacing around elements. * box-sizing is set to "border-box," which ensures that an element's total width and height include padding and borders, making layout calculations more intuitive. * Additionally, a default font family of "Fira Sans" is specified for all text elements. This sets the default font for the entire application, which can be overridden as needed for specific elements.      * The body selector is used to set the styling for the entire page's body. It specifies two key properties: * background-color is set to var(--dark-alt), which assigns the background color using a custom CSS variable defined earlier. This allows for consistent theming and makes it easy to update the background color throughout the application by modifying the custom variable. * color is set to var(--light), which defines the text color using another custom CSS variable. Similar to the background color, this approach maintains consistency in text color across the application. * The .App selector is applied to style a specific container element with the class name "App.", where padding is assigned a value of 32px, which adds padding around the content within this container. This padding helps create spacing between the content and the container's edges, enhancing the layout and visual presentation.      * Now, we define the styling for specific text elements, namely top-level headings (`h1`) and subheadings with the class name "h4." * For `h1` elements (top-level headings):   + `font-size` is set to `40px`, specifying the font size to be applied to these headings. This ensures that top-level headings are visually prominent and larger than regular text.   + `font-weight` is set to `700`, indicating a bold font weight. This makes the headings stand out and appear bold.   + `margin-bottom` is set to `32px`, which adds space below the headings. This margin helps create separation between the headings and the content that follows, contributing to a well-structured layout. * For `h4` elements with the class name "h4" (subheadings):   + `font-size` is defined as `18px`, which sets a relatively smaller font size for subheadings compared to top-level headings.   + `color` is set to `var(--light-alt)`, using a custom CSS variable for text color. This ensures that the text color for subheadings is consistent with the defined color scheme.   + `text-transform` is set to "uppercase," converting the text to uppercase letters. This is a stylistic choice for subheadings, making them visually distinct from regular content.   + `font-weight` is set to `400`, which maintains the default font weight for subheadings, ensuring that they are not as bold as the top-level headings.   + `margin-bottom` is set to `16px`, adding space below subheadings to separate them from the content immediately following them.      * The rest of the code is for index.css completed code with self-explanatory comments, so you can play with it later. For now, it’s time to launch your application to see how it looks! * First, go to view -> terminal and then enter the following two commands sequentially:   + cd api   + npm start      * Next, click on ‘+’ to add a new terminal and then enter the following two commands sequentially:   + cd api   + npm start      * Now, it’s time for you to explore! Feel free to change the styling and display text, save the file, and then see how the application updates! * How would you go about adding a sort by manufactured year feature? (It would be virtually similar to the sort by speed functionality). You would have to edit the following files:   + api/models/Car.js   + api/server.js   + client/src/App.js   + client/src/index.js   **Adding the ‘Sort Cars by Year of Manufacture’ functionality**   1. api/models/Car.js      1. api/server.js      1. client/src/App.js              1. client/src/index.css | |
| **MODIFICATIONS AND ADAPTATIONS** | |
| This activity can be a little harder for those that have a hard time working with partners. Try to have the participants join into groups with their friends if they have them in the club or keep a close eye on those who have a harder time in a group setting | |
| **BACKGROUND INFORMATION** | |
| * + N/A | |

# 

# **Last Modified - 03/10/23**