MONASH University

ENG1003 - Engineering mobile apps - S1 2021

Assignment 1b Instructions

2. Tasks

You will be building a small web app onto an existing MDL skeleton site to allow student registration for drop-in consultation session(s) for an engineering unit.

The system is set up to have two queues by default, and to automatically assign a student to the shortest queue.

This system works by having the app open on the AV machine in the room for the consultation session. The system allows arriving students to add themselves to the queue by providing some information, and then see where they get placed in the queue itself. When staff are ready to take on another student, they will go to the machine and mark the 'current' student that they have just seen as 'done', which removes said student from the queue. They can then view details about the next student in their queue before calling for the student and assisting them.

Currently, the app has three HTML pages. **index.html** is the main page of the app, and shows a live clock, as well as the current queue. It allows the viewing of the student details, as well as marking of a student as 'done'. It also has a link to the 'add student' functionality.

The **add.html** page allows a student to be added to the session, and is accessed by clicking on the 'Add student' link from the main page.

The **view.html** page shows all the details of a student in the session, and is accessed by clicking on the relevant 'details' button for a given student on the main page.

Here's the folder structure provided to you. You can **download the files here [click]**

- A1b/
  - index.html
  - add.html
  - view.html
  - js/
    - shared.js
    - main.js
    - add.js
    - view.js

The **shared.js** file is loaded on all pages and should contain the definition for classes, as well as *shared code* that will execute on all pages such as the loading of data from local storage when the page loads. In the file skeleton, you are provided with **three constants** pre-defined for you to use, which are the `STUDENT_INDEX_KEY` to store the selected student's index (position) in the queue, the `STUDENT_QUEUE_KEY` to store the selected student's queue index in the list of queues, and the `APP_DATA_KEY`, to store the Session instance for this app.

The **main.js** file is loaded second on the **index.html** page, and contains code that runs only on that page.

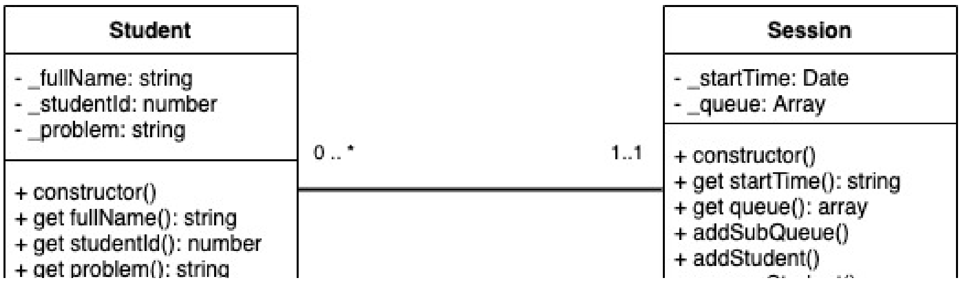
The **add.js** file is loaded second on the **add.html** page, and contains code that runs only on that page.

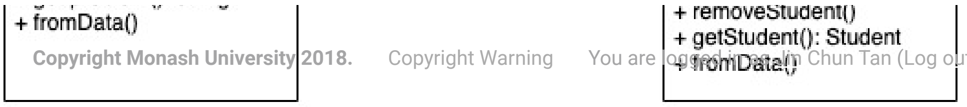
The **view.js** file is loaded second on the **view.html** page, and contains code that runs only on that page.

Task 1

First, you will need to write up the main classes that we will use in this app. You should implement this task in the **shared.js** file.

Class Diagram





You should define the classes, along with the attributes (in the constructor), accessors and methods. You can find a description of what each method does below.

Student Class constructor()

This constructor should accept three parameters for the student's full name, student id and problem description. It should assign these to the appropriate attributes.

Student Class get fullName()

This is the accessor for the `_fullName` attribute.

Student Class get studentId()

This is the accessor for the `_studentId` attribute.

Student Class get problem()

This is the accessor for the `_problem` attribute.

Student Class fromData()

This is the method responsible for restoring the data state retrieved from local storage for a single student instance. It takes one parameter which is the student data object. It should assign the object properties to the appropriate attributes.

Note: This method shouldn't directly interact with local storage, but simply process the provided object in the parameter/argument.

Session Class constructor()

This constructor does not take any parameters. It should set the `_startTime` attribute to the current date/time using a Date object, and initialise the `_queue` attribute with an empty array.

Session Class get startTime()

This is the accessor for the `_startTime` attribute. It should return a formatted string matching the locale date/time format.

Session Class get queue()

This is the accessor for the `_queue` attribute.

Session Class addSubQueue()

This method is responsible for adding a new sub-queue to the `_queue` attribute. A sub-queue starts as a new empty array.

Note: The queue will contain an array of sub-queues, which are in turn, arrays.

Session Class addStudent()

This method is responsible for adding a new student to the *shortest queue* for the session. If all the existing queues are of the same length, it will add the student to the first queue.

The method should accept three parameters, the student's full name, student id and problem description. It will create a new instance of the Student class using the data provided, determine which queue the student should join, and add the student to the queue.

Note: We refer to the sub-queues as the individual queues inside the `_queue` array, which stores all the queues for the session.

Session Class removeStudent()

This method is responsible for removing (deleting) a student from the queue. The method should accept two parameters - the student's index (position) in the queue, along with the queue's index in the `_queue` attribute.

Session Class getStudent()

This method is responsible for accessing (and returning) a specific Student instance in the queue. The method should accept two parameters - the student's index (position) in the queue, along with the queue's index in the `_queue` attribute. It should return the student instance (object) using the `return` statement.

Session Class fromData()

This is the method responsible for restoring the data state retrieved from local storage for a single session instance. It takes one parameter which is the session data object. It should assign the object properties to the appropriate attributes. You will need to iterate over the `_queue` property in the data object and re-create *each* Student instance using the Student Class's `fromData` method for each queue in the data's `_queue` property.

Note: This method shouldn't directly interact with local storage, but simply process the provided object in the parameter/argument.

Task 2

Write a function to check if data exists in localStorage. You should implement this in the **shared.js** file. This function should take one parameter, which is the **key**.

The function should check to see if any data is stored in localStorage at the provided key, and return `true` if there is data, and `false` if there is no data.

Task 3

Write a function to update data into localStorage. You should implement this in the **shared.js** file. This function should take two parameters, which is the **key** and the **data** to be stored.

The function should stringify the data provided before storing it in localStorage at the provided key.

Task 4

Write a function to retrieve / get data from localStorage. You should implement this in the **shared.js** file. This function should take one parameter, the **key**.

The function should retrieve the data from localStorage at the provided key, then determine if it is necessary to parse the data back into an object, and do so if required before returning it using the `return` statement.

Note: you can make use of `try{...}catch(error){...}finally{...}` here if you want to do so. (Refer to the demo shown in the workshop in the LS example)

Task 5

To wrap up your work in the **shared.js** file, you should write some code **at the bottom of the file that will run when the file is loaded**.

- Create a new Session instance stored in a variable `consultSession`
- Check if any data exists in local storage for the `APP_DATA_KEY` using your Task 2 function, and
  - If data exists, retrieve it using your Task 4 function, and restore it using the Session class `fromData` method into the `consultSession` variable
  - If data doesn't exist, then create (by default) two queues using the Session class `addSubQueue` method, and then update localStorage using your work from Task 3. Note: you should be storing the `consultSession` Session instance into localStorage using the key stored in `APP_DATA_KEY`.

Task 6

Write a function **addStudent** that will be responsible for adding a student based on the user input provided. You should implement this task in the **add.js** file.

To begin with, you should take a look at the HTML code in the **add.html** file. Try to understand what the code on the page is responsible for and doing, especially after line 31.

You are provided with input elements and their corresponding error message fields:

- student's full name (id: fullName)
  - error message field (id: fullName_msg)
- student's id (id: studentId)
  - error message field (id: studentId_msg)
- problem description (id: problem)
  - error message field (id: problem_msg)

You will need to write some validation for the user input provided:

- check that the input **is not blank**
- check that the student id is valid (i.e. in the right format)

Tips:

- You can check if the student id is in the correct format by using a *regular expression*(regex) check. The regex looks like this:
  `const regex = /^[1-3]{1}[0-9]{7}$/;`
  You can use the string method `match` to see if a string matches a defined regex (See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match).

  Sites such as https://regexr.com/ help explain and let you build regexes for your requirements. Briefly though, here's an explanation of the parts that make up the regex above.
  - `/` The start of the regular expression
  - `^` Defines the 'start of the string' that we are checking
  - `[1-3]{1}` States that the **first** character must be a value from 1-3
  - `[0-9]{7}` States that the **next seven** characters must be a value from 0-9
  - `$` Defines the 'end of the string' that we are checking (i.e. the string must match exactly the contents between `^` and `$`
  - `/` The end of the regular expression

If any part of the input is **invalid**, you should display a message in the field's corresponding error message field to show the user what the problem is, and then end the execution of this function using a `return` statement.

If all of it is valid, you should then add the student using the `addStudent` method from the Session class, and then update the session data in localStorage using your Task 3 function. Finally, use an `alert` to notify the user that the student has been added to the session or queue, and then redirect the user to the **index.html** page.

Note: You can redirect a user in JavaScript using `window.location` by *assigning* a string containing the relative path to the file, to the property.

Task 7

Write a function that will display a 'live' clock on the main page (index.html). You should implement this task in the **main.js** file.

A span element has been defined with the id: currentTime on the index.html page for you.

You should use this space to display the current time to the user. This needs to be a 'live' clock.

Task 8

Write a function **view** that takes two parameters, **index** (student position in queue) and **queueIndex** (index of queue in the array of queues). You should implement this task in the **main.js** file.

This function is responsible for passing the data in the parameters provided to the view.html page via localStorage. You can do this by storing the relevant values using the keys `STUDENT_INDEX_KEY` and `STUDENT_QUEUE_KEY` and then redirecting the user to the **view.html** page.

Task 9

Write a function **markDone** that takes two parameters, **index** (student position in queue) and **queueIndex** (index of queue in the array of queues). You should implement this task in the **main.js** file.

You will need to confirm with the user that they intend to mark this student as 'done'. If the user agrees, you will need to remove the student from the queue using the `removeStudent` method from the Session class, and then update localStorage with the latest data.

Task 10

Write a function that will display the current queue status to the user. You should implement this task in the **main.js** file.

This function should take one parameter, **data**, which is the **queue data** that will be provided as an argument when the function is called.

Hint: you can access the queue data via `consultSession.queue`.

Before starting, take a look at the marked up code in the **index.html** file that shows the queue 'list' and each 'item' in the queue. A div has been defined (with examples, you don't have to delete this as it gets replaced by your code when it runs) with the id: queueContent.

You will need to generate some HTML code that displays the queues and the students (name) in them into this area. You are provided with the template of how the queue and items within the queue should look like in the index.html file.

Note that you will need to dynamically insert the values for the onclick attributes for view and markDone inside the `<a>` elements with the actual values for the student's position in the queue (index) and the index of the queue itself in the array of queues (queueIndex). This is so that your Task 8 and Task 9 function will work properly, for the student selected.

Once done, you will need to edit the function **markDone** from Task 9 and add a call to **this** function that you have written in Task 10 at the very bottom with the appropriate data. You should also add a call to the function you wrote for this task at the bottom of the main.js file so that the queue is displayed on the page when the index.html page loads.

Task 11

Task 11

This task will require you to write some HTML code using the MDL library Components (getmdl.io) in the file **view.html**.

The **view.html** page is responsible for showing the user the information available about a selected student (You'll write the function to show this data in Task 12).

The HTML file is provided to you with an **empty div with the class of page-content**. You should write some HTML using the MDL components in here, and ensure that you use the **mdl-grid** layout https://getmdl.io/components/index.html#layout-section/grid to position your elements on the page.

Hints:

- a div with `class="mdl-grid"` creates a new **row** of content on the page.
- divs inside the mdl-grid div using the classes `mdl-cell mdl-cell--x-col` are positioned as cells in the 'row' and will show up side by side.
  - Desktop screens have a maximum 'width' of 12 x
  - Tablet screens have a maximum 'width' of 8 x
  - Phone screens have a maximum 'width' of 4 x

You will need to show data on the student on this page, which consists of:

- Student's full name
- Student's ID
- Student's problem description

It is up to you on how you want to format this display - there are marks for how 'nice' it looks on the page.

Task 12

This task will require you to write some global code that will run when the **view.js** file is loaded.

You will need to retrieve the stored indexes, using the keys `STUDENT_INDEX_KEY` and `STUDENT_QUEUE_KEY` into appropriate variables. Then, retrieve the selected student using the indexes via the `getStudent` method in the Session class.

You should then create references to the HTML elements you defined to display the information on the student, and display the information using the `innerText` or `innerHTML` property.

Task 13

Finally, clean up your code and document your work. Make sure that you

- have appropriate code Indentation and Code Brackets (per the style guide).
- have appropriate variable and function naming (per the style guide).
- have appropriate variable and function scoping (smallest possible).
- have appropriate function header documentation (name, purpose of function, parameter and return value description).
- do not have magic numbers in your code.
- have appropriate file header documentation (name, purpose/description of file) for all JS files.
- start each JS file with `"use strict";`

◄ Assignment 1a submission (Due W5 Thu 8PM)

Jump to...

Assignment 1b submission (Due W8 Fri 8PM) ►