Todo List App Technical Documentation

Introduction

In the todo list app, MVC (Model-View-Controller) infrastructure is used as a way to communicate data from the user to a local database, and from the local database to the user. In addition to the model view and controller components, a store component was used as a medium for the model to access the database, and the template component was used as a medium for the view to update the DOM. Below I have shown how MVC is used, and the specific functions included with each part of the MVC infrastructure. Furthermore, an audit of the todo list application found the app loaded solidly, but lacked some more advanced features which other todo list applications possess.

Use of MVC

MVC is a method for creating infrastructure for the communication of data in an application. The view is the part of the application which renders content to the page and captures events. The model updates the database with information from the view. The controller is the mastermind of the application. The controller takes data from the view, and updates the model. In addition, the controller takes data from the model and uses the data to render content to the view. This exchange of data is triggered by events captured in the view or data updated or read from the model.

Model

The Model executes functions called from the Controller. These functions perform updates to the local database (ie local storage), which is accessed through the storage component.

Model(storage): A constructor instantiates the model component. The Store component is also instantiated, and will hold the local storage.

Model.create(title, callback): The title is the todo in the form of a string. The callback is a function to execute after the todo has been added to the local storage. This function saves the todo title to the local storage by calling storage.save().

Model.read(query, callback): Model.read() uses the query to search and return a specific todo from the local storage using storage.find(), or if no query is given, will return all todos by calling storage.findAll().

Model.update(id, data, callback): Storage.save(id, data, callback) is called with three arguments including the id which is used to identify a todo, data which is the information that the todo will be updated with, and a callback to execute after the todo is updated in local storage.

Model.removeAll(callback): Removes all data from the storage by calling storage.drop(callback). The callback is the function to call after the storage is wiped.

Model.getCount(callback): A count of all the todos is returned by calling storage.findAll().

View

The View component renders content on the page and captures events to trigger functions in the Controller which update the database with data gathered from the View. The DOM can have new renders of content once the database is updated.

View(template): A contructor which instantiates the View component and the Template component. The view component in the view constructor is connected to the html being rendered in the template component.

View.render(viewCmd, parameter): A specific command is given from the controller as a string in the parameter viewCmd. The string is then used to identify a specific property representing a command within the viewCommands object. The identified command property is a function which is executed so the command can be enacted on the application.

showEntries(): Different list items on the application page are rendered through the Template.show() function whenever the todo status is changed by the user.

removeItem(id): The id is used to identify the todo. The identified todo is removed.

updateElementCount(): Every time a new todo is completed this function is called in order to update the counter showing the number of active todos remaining.

clearCompletedButton(): Updates the text in the clear completed button every time the number of completed todos changes to either an empty string if they're no completed todos or "clear completed" if there is one or more completed todos.

contentBlockVisibility(): This function determines if the block element representing the list of todos will continue to be displayed or removed on the basis of there still being list items remaining.

setFilter(): Updates which type of todo button is shown as selected every time a user selects a different list type.

clearNewTodo(): After an input has been entered to become a todo the input field is reset to an empty string.

elementComplete(): This function is triggered by checking off a todo as completed or active, and will cause the todo's checkbox and text to reflect the status of the todo.

editItem(id, title): The text of a double clicked on todo will be placed in an input element which has focus allowing for the editing of the todo list item.

editItemDone(): After editing an item this function will cause the input element to be removed, and the todo to appear as a regular list item.

Controller

The Controller uses events captured from the view to execute functions which either update the Model, or extract data from the Model to be used to render content to the View.

Controller(model, view): The Controller constructor holds the view and model components used to communicate data.

Controller.setView(locationHash): The locationHash is a string representing the status of the todo list, either "active", "completed", or "all". This string is passed to the updateFilterState() and is used to determine which todos will appear in the todo list.

Controller.showAll(): When showAll() is called Model.read() is executed with no query parameter, resulting in all the todos being returned from the database as an array. The returned todo list array is rendered with the "showEntries" viewCommand passed as an argument in a callback which is a parameter within Model.read().

Controller.showActive(): When showActive() is called Model.read() is executed and returns all active todos from the database. The active todos are rendered with the "showEntries" viewCommand passed as an argument in a callback passed as the last argrument into Model.read().

Controller.showCompleted(): When showCompleted() is called Model.read() is executed and returns all todos from the database which are completed. The completed todos will be rendered with the "showEntries" viewCommand passed as an argument in a callback passed as the last argument into Model.read().

Controller.addItem(title): The title, the text of the todo, is added as a part of a new todo object to the todos in the database through Model.create(). A callback function passed as the final argument in Model.create() causes the input to be cleared, and the re-rendering of all the todos, including the todo just added to the database.

Controller.editItem(id): When editItem() is call Model.read() will be executed with the id of a todo the user double clicked on passed as the first argument. Model.read() will return a todo from the database which has the same id. In a callback passed as the last argument to Model.read() the returned todo will be rendered in the editing mode.

Controller.editItemSave(id, title): Any blank spaces at the beginning or end of the string representing the edited todo are removed using slice(). The edited todo string and the todo id are passed to Model.update(). Model.update() will match the todo with the argument id with a todo of the same id in the database, and change the todo's string value in the database to the edited string value. A callback, the final argument in Model.update(), will cause the view to eliminate the editing input field, and render the newly edited todo as a regular todo.

Controller.editItemCancel(id): The id for the todo being edited is passed to Model.read(), and a todo with the matching id from the database is returned. In a callback, the last argument passed to Model.read(), the editing input field is removed and the returned todo is rendered as the todo.

Controller.removeItem(id): The id of the todo to be removed is passed to Model.remove(), and the todo in the database which matches the id passed into Model.remove() will be removed from the database. Additionally, a callback, passed as the second argument to Model.remove(), will cause the todo with the id passed into Model.remove() to be removed from the DOM.

Controller.removeCompletedItems(): Model.read() will be triggered and return all completed todos. The returned todos will be removed from the database and the DOM by triggering removeItem() from a callback passed to Model.read().

Controller.toggleComplete(id, completed, silent): Model.update() will be called with the todo in the database which matches the id passed to Model.update() having it's completed boolean value change to match the completed parameter's boolean value from toggleComplete(). The third argument passed to Model.update(), a callback function, will change the state of the checkbox to either checked or unchecked, also depending on the boolean value passed into Model.update().

Controller.toggleAll(completed): The status of all todos will be altered from active to completed or vice versa by passing in the current status of completed as a boolean, obtaining all the todos from the database in an array by calling Model.read(), and changing the status of all the todos in the returned array to the opposite. The resutling array, with the altered todo status', is iterated through, and for each iteration, the todo in the iteration is passed to Controller.toggleComplete() to have the changed status updated in the database via the model, and in the DOM via the view.

Controller._updateCount(): Model.getCount() is called and returns the number of total todos as the argument for the callback passed into Model.getCount(). The total number of todos is used to update various parts of the page including the todo count, the clearCompletedButton, toggleAll and contentBlockVisibility.

Controller._filter(force): The todos will be filtered based on the current active route. If the last active route isn't "all", or the routes are switching, the todos will be rendered under the correct route by calling show[All|Active|Completed]().

Controller._updateFilterState(currentPage): currentPage is a string representing the current status of the list. The navigation button shown as selected will be changed to match the list status the user has chosen by passing the currentPage parameter to view.render() with "setFilter" as the first argument.

Store

The Store contains the function declarations which alter the local storage database. These functions are executed in the Model on the basis of events from the View triggering functions in the Controller which trigger the functions from the Store to be called in the Model.

Store(name, callback): A new database is created with the same name as the name passed into the constructor method. The database will contain an array of objects with each object having a todo's text and id.

Store.find(query, callback): Locates specific todos of the same status by matching an iteration of an array of selected todos of a certain status with a particular todo of the same status from an array of todos from the database.

Store.findAll(callback): An array of all the todos from the database, regardless of their status, is returned.

Store.save(updateData, callback, id): Store.save() will either update a todo if the id passed through matches the id of an already existing todo from the database or else a new todo will be added to the database. The callback will execute code which will be a reaction to what was saved in the database.

Store.remove(id, callback): The passed in id will be matched with an id of an iteration of a todo from the database array of todos. The matched todo from the database will be removed.

Store.drop(callback): All data in the database will be removed when this function is called by replacing the data in the database with an empty array.

Template

The Template component establishes the html for the todos and the todo list itself. The Template has a series of function declarations which are called in functions in the View. Those functions are called in the Controller if data is either read, created, updated or removed from the model.

Template(): A constructor which contains the default template for each todo list item. The default template is a list element which is given a unique id, status and title.

Template.show(data): The data parameter represents an array of objects, where each object contains a specific todo's id, title and status. This array of todo objects is iterated through and for each iteration the todo item's id, title and status are used to correctly render the todo to the page by replacing the default id, title and status values in the template with the values specific to the todo of the current iteration.

Template.itemCounter(activeTodos): Every time the number of active todos changes this function is executed causing the active todo counter to be altered. The parameter of activeTodos represents the number of active todo's and is returned as a part of a string to be rendered as the active todo counter. Also, if the activeTodo's is greater than 1 then a 's' is added to the end of the string to pluralize the active todo counter.

Template.clearCompletedButton(completedTodos): Every time the status of a todo changes this function is executed to determine if the number of completed todos is greater than zero. If greater than zero, the string "Clear completed" is returned causing the "Clear completed" button to be rendered, or else an empty string is returned, and no "Clear completed" button is generated.

JavaScript Errors Fixed

1. Type Error

Location: controller.js line 96

Description: The error was a type error involving a misspelling of the word addItem.

Solution: The misspelled word was spelled correctly to avoid the type error.

2. Possible Duplicate Id's Error

Location: store.js line 88

Description: The id was created by randomly generating a number, and multiplying the number by the length of a string (10), and then rounding the resulting number down to the nearest whole number. The process of generating a random number was repeated using a loop and the random numbers were all added to the same id variable on each iteration. The error is that if this application is used on a large scale it is likely that two or more list items will have the same random numbers generated, causing an error where both list items of the same id have the same behaviour since they are represented in DOM manipulation by the same id.

Solution: The second error was fixed by ensuring each newly generated list item has an initial id value of 'list' followed by concating a unique string number created by taking the length of the todos array, and adding one, thus creating a unique id for each list item by increasing the numerical value of each list id by one as each new list item is added to the todos array (i.e. var newId = `item ${todos.length + 1}`;).

3. Extra Logging Statements

Location: controller.js lines 168 - 172

Description: It is not necessary to have a logging statement to the console for each list item's id every time a list item is removed since this is only excess code which serves no functional or organization purposes.

Solution: The solution is to remove all the code resulting in the unnecessary logging statement of a list item's id to the console each time a list item is removed.

4. Unoptimized Loops 1

Location: controller.js - line 126 - 132

Description: The two while loops on line 126 -132 are not necessary since there is only one iteration looking at a single list item, requiring there to be no need for loops which should be used to iterate through an array of multiple iterations.

Solution: The two while loops should be turned into if conditional statements.

5. Unoptimized Loops 2

Location: store.js lines 122 -132

Description: There are two for loops beside each other which each have the same number of iterations, with each iteration starting at 0. Thus, both for loops are unnecessary.

Solution: One for loop should be removed and both if conditional statements should be placed into the same for loop.

6. Unoptimized Loops 3

Location: template.js line 65

Description: The for loop is set to iterate through a variable which is made equal to the array length of the data array when no variable is necessary, and the for loop should iterate directly based on the length of the data array.

Solution: The for loop should iterate through the data array's length directly instead of through a variable made equal to the data array's length since this reduces unnecessary code.

Todo List Application Audit

The developer tools feature of lighthouse was used to audit the todo list app. The results of the audit can be found at this link: https://github.com/mbdev95/OpenClassrooms_Project_8_Debugging_todoListApp/blob/master/Lighthouse_Audits/Todo%20List%20Audit.pdf. In summary, the audit illustrated the todo list app had an overall fast load time. However, the load time could still be improved by reducing the JavaScript file sizes, as well as modernizing the JavaScript code. Moreover, the audit found that accessibility could be improved by ensuring form input elements have labels to make the input elements more readable to screen readers. Overall, the todo list app audit yielded few issues in comparison to the competitor app, however the competitor app had a more advanced user interface.

Conclusion

In conclusion, MVC is effectively used to execute JavaScript functions which either alter or read the database on the basis of an event or input being captured. On the other hand, MVC is also effectively used to execute JavaScript functions which render content from the database. Finally, the audit showed the app overall works well in terms of load time, but could have a more advanced user interface in order to better compete with other todo list applications.