**Technical Documentation**

**MVC**

The Todo App is based on an **MVC framework** built from JavaScript. MVC stands for *Model-View-Controller*. In MVC, the concerns of the application are separated into three main components: the *model*, the *view* and the *controller*. Each component handles its own tasks and functionality, communicating with the other two components to form a complete system.

In our app, the main files that perform our tasks are outlined below.

**view.js**

The **view.js** component is what renders the view to the user and captures events and changes in the UI, triggering functions in the *controller.js* component.

**View(template)**

A constructor that creates references to all areas of the HTML view, and attaches the HTML from template.js for DOM rendering.

**View.render(viewCmd, parameter)**

This function defines the viewCommands object, which contains methods to update the view as needed. These methods are what are passed in as viewCmd, which is then passed to viewCommands[viewCmd] as the index, as shown below.

**showEntries()**

Runs every time the list view needs to change the to-do items showing. Updates the items in visible list.

**removeItem()**

Runs every time a to-do item needs to be removed from the list. Removes the item from the view.

**updateElementCount()**

Runs every time a change is made to the list items, and updates the active count in the bottom of the UI.

**clearCompletedButton()**

Runs each time a change on the total number of completed occurs. Determines whether the clear completed button will display or not.

**contentBlockVisibility()**

Runs each time a change occurs with the list items. Determines if the list block will display or not. Returns true until there are no to-do items to show.

**toggleAll()**

When the toggle all button is clicked, toggles the checkbox (hidden) for toggle all, which in turn triggers the Controller.

**setFilter()**

Runs whenever the view status changes, to determine which to-do items match the current status, and will display those items.

**clearNewTodo()**

Will remove the text from the input field after a new to-do has been entered.

**elementComplete()**

Updates the checkbox and text of to-do to match the status of the item.

**editItem()**

Changes the to-do item clicked on, to become editable with focus on it and makes it editable.

**editItemDone()**

Returns the state of the list item being edited, to a non-editable state.

**controller.js**

The **controller.js** component is what communicates between the view and model. It will trigger methods in both *view.js* and *model.js* as needed, to keep the view and database up to date with user input and requests.

**Controller(model, view)**

A constructor that creates a controller to communicate between the model and view components passed in by reference.

**Controller.setView(locationHash)**

Initializes the view on load, and set the route upon view change, passed in as locationHash. Then triggers Controller.\_updateFilterState. Options to pass in are all, active and complete.

**Controller.showAll()**

Triggers Model.read() on the database of to-do items to be read, with no query parameter. Resulting in returning the entire database. Also passes the callback function view.render() which runs the viewCommand “showEntries” with the returned data .

**Controller.showActive()**

Triggers Model.read() on the database of to-do items to be read, with the query parameter of {completed: false}. Resulting in returning the active to-do objects. Also passes the callback function view.render() which runs the viewCommand “showEntries” with the returned data.

**Controller.showCompleted()**

Triggers Model.read() on the database of to-do items to be read, with the query parameter of {completed: true}. Resulting in returning the completed to-do objects. Also passes the callback function view.render() which runs the viewCommand “showEntries” with the returned data.

**Controller.addItem(title)**

Triggers Model.create() passing the trimmed value of the input field as the title of the new to-do item. It then clears the input field, and re-filters the database so that the new to-do item is included in the list view, if applicable.

**Controller.editItem(id)**

Triggers Model.read() passing the ID of the to-do object to be edited, and View.render(‘editItem’) as a callback function. Will record any changes to an exiting to-do item in both the model and update the view to match.

**Controller.editItemSave(id, title)**

Trims any white space from the beginning or end of the title after editing is complete. If there are any characters in edited title, triggers Model.update() passing the ID of the to-do object to be edited, and new title. Also triggers View.render(‘editItemDone’) as a callback function to update the view to match the updated model.

**Controller.editItemCancel(id)**

Triggers Model.read() to return the original value of the to-do title, then triggers View.render(‘editItemDone’) to reset the view of the to-do to match the original value.

**Controller.removeItem(id)**

Finds the to-do item matching the ID passed, and triggers Model.remove() and View.render(‘removeItem’) to remove it from the database and view.

**Controller.toggleComplete(id, completed)**

For each ID passed in, triggers Model.update() to update the completed status of the item to match the completed status passed in. Then triggers the View.render(‘elementComplete’) to match view to database.

**Controller.toggleAll(completed)**

Triggers Model.read() passing in a completed status opposite to the current completed state, and a callback function of .forEach() to apply the new status to each to-do item in the database.

**model.js**

The **model.js** component is what accesses and updates the local database via the *store.js* component.

**Model(storage)**

Constructor to create a Model instance, hooking up to the database passed. Triggers Store() in store.js component.

**Model.create(title, callback)**

Creates a new to-do item and writes it to the database by triggering Store.save() in store.js component.

**Model.read(query, callback)**

Finds and returns an object(s) from the database, and performs the callback function passed, if any is passed. Done by triggering Store.find() in store.js component.

**Model.update(id, data, callback)**

Updates an existing to-do in the database, by passing an ID and triggering Store.save() in the store.js component.

**Model.remove(id, callback)**

Finds an existing to-do object by the passed ID, and removes it from the database by triggering Store.remove() in the store.js component.

**Model.getCount(callback)**

Returns a count of completed, active and all to-do items in the database. This is done by triggering Store.findAll() in the store.js component.

**store.js**

Todo App uses *local storage* as it’s default database. However, it is also possible to access a remote database through AJAX calls. Access to either of these is through the **store.js** component. All data written to local storage from store.js is converted to a string prior saving. All data passed forward from the database is parsed back into an object.

**Store(name, callback)**

Creates a new local storage database matching the name passed, if no local storage database already exists. If a callback function is passed, it should be an AJAX call to a remote database to use in the app.

**Store.find(query, callback)**

Find results via query passed. This could be all completed to-dos, active to-dos, or an individual to-do via ID. The callback passed is a function that determines what is done with the objects found.

**Store.findAll(callback)**

Returns all objects in the database and preforms the callback function passed in on them.

**Store.save(updateData, callback, id)**

Will create a new to-do object and save to the database, via object passed in as updateData. Alternatively, if an ID is passed in it will update the matched existing object with the updateData passed in.

**Store.remove(id, callback)**

Remove the to-do item that matches ID passed in. Fires for individual removal as well as when using “clear Completed”.

**template.js**

In **template.js** the basic HTML structure is laid out for individual to-do items, the to-do list, the active items counter, and the clear completed button.

**Template()**

Template() is a constructor that outlines the HTML for each individual to-do item. Each to-do is a list an <li> that has placeholders for the items unique ID, completed/active status and title.

**Template.show(data)**

The to-do items are generated from the template here, with a for loop which runs over each to-do object in the database passed in. As each to-do is created from the template, replacing the placeholder values with its own values.

**Template.itemCounter(activeTodos)**

This is the method that creates the counter at the bottom of the app, which displays the number of active items in the list. In addition, if the activeTodos passed in equal 1 then no “s” is applied to the string. Otherwise an “s” is applied to pluralize the string.

**Template.clearCompletedButton(completedTodos)**

This method determines if the “clear completed” button is generated or not. If there are any completed to-da items in the passed in completedTodos, then the button is generated. Else, an empty string is returned resulting in no button.