TODO APP DOCUMENTATION 1.0

Overview

The project is a To-do Web App with basic features such as: Adding, Editing, Deleting Filtering and Toggling To-dos as well as To-do data being persisted across browser sessions

Getting Started

Start by downloading the App's project files from here and extract the contents of the downloaded zipped folder.

From the project folder's root directory run `npm install` to install the project's dependencies (specifically Jasmine).

Files/Folder Structure

`index.html`

`package.json`

`.gitignore`

`node_modules/`

- `node_modules/jasmine-core/`
- `node_modules/todomvc-app-css/`
  - `index.css`
- `node_modules/todomvc-app-css/`
  - `package.json`
  - `readme.md`
- `node_modules/todomvc-common/`
  - `base.css`
  - `base.js`
  - `package.json`
  - `readme.md`

`js/`

- `app.js`
- `controller.js`
- `helpers.js`
- `model.js`
- `store.js`
- `template.js`
- `view.js`

`test/`

- `ControllerSpec.js`
- `SpecRunner.html`

Launching the App

Launch the `index.html` file from the project's root directory using the browser of your choice with JavaScript enabled.

App Features

Adding New Todos

To add a todo, type the todo text at the text input at the top of the App and press the enter key. Extra white spaces are trimmed before the todo is saved.

Editing Existing Todos

Double Click on a todo text to enable editing mode. Press the enter key when done.

Toggle Todos

Toggle todos (complete/active) by clicking on the checkbox beside (left side) each todo. To toggle all, click on the toggle all button (grey pointing down arrow head) at the left side of the todo input

Filter Todos

At the bottom of the todo list are the following items:

- Counter that show the number of active todos
- `All` button to display all todos
- `Active` button to show only active todos
- `Completed` button to show only completed todos
- `Clear completed` button to delete only completed todos. Visible only when there are completed todos.

Delete Todos

Hover over a todo item and click on the red 'x' button at the right side of the toto item.

Language and Dependencies

The App is written in HTML, CSS and JavaScript (ES5).

HTML for a basic structure, CSS for styling, and JavaScript for the App's interactivity.

The Jasmine library is used for testing.

Other dependencies are listed in the package.json file at the projects root directory

Code Architecture

The App utilizes an MVC (Model – View – Controller) Pattern.

Object Constructors

The App utilizes the following constructors, which are all exported to the `window` object as properties of an `app` object when the App is first loaded.

Store

Defined in `store.js` (`line 13`).

It instantiates a client side storage object (utilizes the browser's localStorage Web API) to be used as a todo database.

Parameters

It can take 2 parameters:

- `name`: A `string` that represents the name of the database
- A `callback` function.

Methods

The Store constructor has five (5) methods.

- **Store.prototype.find**: Finds items in the todos database based on a query given as a JavaScript object. For example, when editing or deleting a todo item.

  It takes up two(2) parameters

  - `query` A JavaScript object e.g `{title: "My todo"}`
  - `callback` A function invoked on the result of the query (`true || false`)
- **Store.prototype.findAll**: Retrieves all data from the todos database.

  It takes a single parameter, a `callback` which is invoked upon retrieving data.
- **Store.prototype.save**: Saves the given data to the todos database. If no item exists it will create a new item, otherwise it'll simply update an existing item's properties. For example, adding a new todo item or saving an edited todo item.

  It takes up to three(3) parameters

  - `updateData` A JavaScript object. The data to save back into the database.
  - `callback` The function to be invoked after saving
  - `id` The ID of an item to update (*optional*)
- **Store.prototype.remove**: Will remove an item from the database based on its ID. For example, deleting a todo item

  It takes up two(2) parameters

  - `id` The ID of the item you want to remove
  - `callback` The function to be invoked after removal
- **Store.prototype.drop**: Will drop (delete the database) all storage.

  It takes a single parameter, a `callback` which is invoked after the database has been dropped.

Model

Defined in `model.js` (`line 10`).

Creates a new Model instance and hooks up the client side storage(database) created by the `Store` constructor.

Parameters

It takes a single parameter

- `storage` A JavaScript object that is a reference to the client side storage.

Methods

- **Model.prototype.create**: Creates a new todo model (Basically a todo item task).

  It takes up two(2) parameters.

  - `title`. A `string` that represents the title of the todo task
  - `callback` A function invoked after the model/todo item is created
- **Model.prototype.read**: Finds and returns a model/todo item in storage. If no query is given it returns everything.

  It takes up two(2) parameters.

  - `query` which can be a `string || number || object`

    Example

    ```
    model.read(1, func); // Will find the model with an ID of 1
    model.read('1'); // Same as above
    model.read({ title: "My todo task" }); //Will find a model with title
    equalling "My todo task".
    ```

  - `callback` A function invoked after the model/todo item is found
- **Model.prototype.update**: Updates a model/todo item.

  It takes up three(3) parameters.

  - `id` The id of the model/todo item to update
  - `data` The properties to update and their new value
  - `callback` A function invoked after the model/todo item is updated.
- **Model.prototype.remove**: Removes a model/todo item from storage

  It takes up two(2) parameters.

  - `id` The id of the model/todo item to remove
  - `callback` A function invoked after the model/todo item is removed.
- **Model.prototype.removeAll**: Will remove ALL data from storage

  It takes a single parameter.

  - `callback` A function invoked after ALL data is removed.
- **Model.prototype.getCount**: Returns a count of all todos (active, completed and total todos)

  It takes a single parameter.

  - `callback` A function invoked after all todos after count is done.

Template

Defined in `template.js` (`line 32`).

Sets up defaults for all the Template methods such as a default template

Parameters

It is called without any parameters

Methods

- **Template.prototype.show**: Creates an `<li>` HTML string and returns it for placement in the app.

  It takes a single parameter.

  - `data` A JavaScript object containing the properties you want to find in the template to replace.

    Example

    ```
    template.show({id: 1,
    title: "My to-do task",
    completed: 0,
    });
    ```

- **Template.prototype.itemCounter**: Displays a counter of how many to dos are left to complete and returns a `string` containing the count.

  It takes a single parameter.

  - `activeTodos` The number of active todos.
- **Template.prototype.clearCompletedButton**: Updates the text within the "Clear completed" button and returns a `string` containing the count

  It takes a single parameter.

  - `completedTodos` The number of completed todos.

View

Defined in `view.js` (`line 15`).

It instantiates a view object that abstracts away the browser's DOM.

Parameters

It takes a single parameter

- `template` The template object

Methods

The View Constructor has two(2) 'public' methods. The remaining methods are indicated as 'private' methods called within the two main public methods.

- **View.prototype.render**: Renders a given command with the options.

  It takes up to two(2) parameters

- `viewCmd` A string that matches a property of the `viewCommands` object defined within the method.
- `parameter` A JavaScript object with todo details that the `viewCmd` acts on.

  Example

  ```
  // Check the Toggle all button
  view.render('toggleAll', {
        checked: true
      });

  // Edit an existing todo
  view.render('editItem', {
        title: "My to-do task"
      });
  ```

The following 'private' methods are called within the `View.prototype.render` method under corresponding properties within the `render.viewCommands` object:

- **View.prototype._removeItem**: render.viewCommands.removeItem
- **View.prototype._clearCompletedButton**: render.viewCommands.clearCompletedButton
- **View.prototype._setFilter**: render.viewCommands.setFilter
- **View.prototype._elementComplete**: render.viewCommands.elementComplete
- **View.prototype._editItem**: render.viewCommands.editItem
- **View.prototype._editItemDone**: render.viewCommands.editItemDone

- **View.prototype.bind**: Registers the handler for a todo application event.

  It takes up to two(2) parameters

  - `event` A `string` describing application the event and determines the type of event listener (`click || dblclick || change`) to bind to the corresponding DOM node.
  - `handler` The function that is called for the corresponding event.

  The following 'private' methods are called within the `View.prototype.bind` method for corresponding events.

  - **View.prototype._itemId**: Called where

    ```
    event === 'itemEdit' ||
    event === 'itemRemove' ||
    event === 'itemToggle'
    ```

  - **View.prototype._bindItemEditDone**: Called where `event === 'itemEditDone'`
  - **View.prototype._bindItemEditCancel**: Called where `event === 'itemEditCancel'`

Controller

Defined in `controller.js` (`line 11`).

It instantiates a controller object that takes a model and view and acts as the controller between them

Parameters

It takes two(2) parameters.

- `model` The model instance
- `view` The view instance

Methods

The Controller Constructor has two(12) 'public' methods.

- **Controller.prototype.setView**: Loads and initializes the view. It takes a single parameter
- `locationHash` A string used to create the route
- **Controller.prototype.showAl**l: Called when the App loads. Will get all to-do items and display them in the todo-list. It has no parameters.
- **Controller.prototype.showActive**: Renders all active tasks. It has no parameters.
- **Controller.prototype.showCompleted**: Renders all completed tasks. It has no parameters.
- **Controller.prototype.addItem**: Called when adding a todo item. It takes a single parameter
- `title` A `string` representing the to-do task text
- **Controller.prototype.editItem**: Triggers the item editing mode. It takes a single parameter
- `id` A `number` representing the ID of the to-do task to edit
- **Controller.prototype.editItemSave**: Finishes the item editing mode successfully.

  It takes a two (2) parameters.

  - `id` A `number` representing the ID of the to-do task to edited
  - title A string representing the to-do task text edited
- **Controller.prototype.editItemCancel**: Cancels the item editing mode.

  It takes a single parameter

  - `id` A `number` representing the ID of the to-do task that was being edited.
- **Controller.prototype.removeItem**: Remove/Delete a to-do Item from the DOM and storage/database.

  It takes a single parameter

  - `id` A `number` representing the ID of the to-do item to be removed.
- **Controller.prototype.removeCompletedItems**: Will remove/delete all completed items from the DOM and storage/database. It has no parameters.
- **Controller.prototype.toggleComplete**: Will update a to-do items state (on/off) on the DOM {checked: true|| false} and on the database/storage {completed: true || false}.

  It takes three(3) parameters

  - `id` A `number` representing the ID of the to-do item to be toggled complete/incomplete.
  - `completed` Boolean passed as the value of the model/to-do item `completed` property to update the state of the model/to-do item.
  - `silent` Optional Boolean/undefined to prevent re-filtering to-do items
- **Controller.prototype.toggleAll**: Will toggle ALL checkboxes' on/off state and completeness of models.

  It takes a single parameter.

  - `completed` Boolean passed as the value of ALL the models/to-do items `completed` property to update the state of the models/to-do items.

The Controller also has the following 'private' methods

- **Controller.prototype._updateCount**: Updates the pieces of the page which change (active, completed) depending on the remaining number of todos. It takes no parameter.
- **Controller.prototype._filter**: Re-filters the todo items, based on the active route.

  It takes a single paramater

  - `force` A Boolean or undefined value to determine a forced repainting of to-do items
- **Controller.prototype._updateFilterState**: Simply updates the filter nav's selected states

  It takes a single paramater

  - `currentPage` A `string` to determine the active route

Todo

The Todo constructor is defined in the `app.js` (`line 10`) file.

Instantiates a todo list object when the app is launched.

Parameters

It takes a single parameter

- `name` A `String` which is the name of the todo list object created.

Methods

None.

Helper Methods

A group of helper methods (attached) to the `window` object are defined in the `helpers.js` file

- `window.qs`: Grab an element by using a CSS selector
- `window.qsa`: Grab all matching elements by using a CSS selector
- `window.$on`: addEventListener wrapper
- `window.$delegate`: Attach a handler to event for all elements that match the selector
- `window.$parent`: Find the element's parent with the given tag name

  Example:

  ```
  $parent(qs('a'), 'div');
  ```

- `NodeList.prototype.forEach`: Allow for looping on nodes by chaining

  Example:

  ```
  qsa('.foo').forEach(function () {})
  ```

App Performance

Testing Environment

Operating System: Windows 10 (Version 1809) Browser: Version 79.0.3945.79 (Official Build) (64-bit)

Performance Metrics

First Contentful Paint(FCP)

First Contentful Paint marks the time at which the first text or image is painted.

First Meaningful Paint(FCP)

First Meaningful Paint measures when the primary content of a page is visible.

FCP/FMP time (in seconds)

0–2 (fast) 2–4 (average) Over 4 (slow)

Speed Index

Speed Index shows how quickly the contents of a page are visibly populated.

Speed Index (in seconds)

0–4.3 (fast) 4.4–5.8 (average) Over 5.8 (slow)

First CPU Idle

First CPU Idle marks the first time at which the page's main thread is quiet enough to handle input.

First CPU Idle metric (in seconds)

0–4.7 (fast) 4.8–6.5 (average) Over 6.5 (slow)

Time to Interactive(TTI)

Time to interactive is the amount of time it takes for the page to become fully interactive.

TTI metric (in seconds)

0–5.2 (fast) 5.3–7.3 (average) Over 7.3 (slow)

Max Potential First Input Delay

The maximum potential First Input Delay that your users could experience is the duration, in milliseconds, of the longest task.

Results

Throttling 4x CPU Slowdown					
First Contentful Paint	**First Meaningful Paint**	**Speed Index**	**First CPU Idle**	**Time to interactive**	**Max Potential First Input Delay**
1.9s	1.9s	1.9s	1.9s	1.9s	23.33ms

No Throttling					
First Contentful Paint	**First Meaningful Paint**	**Speed Index**	**First CPU Idle**	**Time to interactive**	**Max Potential First Input Delay**
0.56ms	0.56ms	0.46ms	0.56ms	0.56ms	20ms

Runtime Performance (No Throttling)			
	Total Time	**Scripting**	**Rendering**
Adding Todos	21ms	5ms	1ms
Deleting Todos	32ms	7ms	0ms

Runtime Performance (Throttling 4x CPU Slowdown)			
	Total Time	**Scripting**	**Rendering**
Adding Todos	129ms	31ms	3ms
Deleting Todos	121ms	34ms	2ms

Recommendations

1. Serve images in next-gen formats. Image formats like JPEG 2000, JPEG XR, and WebP often provide better compression than PNG or JPEG, which means faster, downloads and less data consumption.
2. Preconnect to required origins. Consider adding `preconnect` or `dns-prefetch` resource hints to establish early connections to important third-party origins
3. Eliminate render-blocking resources. Todolistme.net has resources are blocking the first paint of your page. Consider delivering critical JS/CSS inline and deferring all non-critical JS/styles
4. Reduce the impact of third-party code. Todolistme.net has third-party code that blocked the main thread for **4,150ms.**

Third-party code can significantly impact load performance. Limit the number of redundant third-party providers and try to load third-party code after your page has primarily finished loading. In this case **Google/Doubleclick Ads** had the most impact with a **size of 223KB** and a **Main-Thread Blocking Time of 1,460ms**.

Other recommendations to remove performance Include:

1. Properly size images. Continue with using SVG for complex icons where possible
2. Defer off-screen images. Consider lazy-loading these images after all critical resources have finished loading to lower Time to Interactive.
3. Minify CSS
4. Minify JavaScript
5. Remove unused CSS

License