# <u>OpenClassrooms Project 8</u>
# <u>Modify an Existing Project</u>

**INTRODUCTION**

The project requiring modification is a fairly common system called a *Todo App*. In particular the application has been programmed using an MVC architecture. It is therefore important to explain briefly about MVC to give a basis for the detailed discussion about the specific details of the *Todo App* code.

**WHAT IS MVC?**

MVC is a common design architecture for computer programming. It originates from the 1970s, however, it has become popular in web-based applications and some javascript frameworks. MVC stands for *Model-View-Controller*, and is a programming pattern for separating the architecture of a program into 3 interconnecting parts.

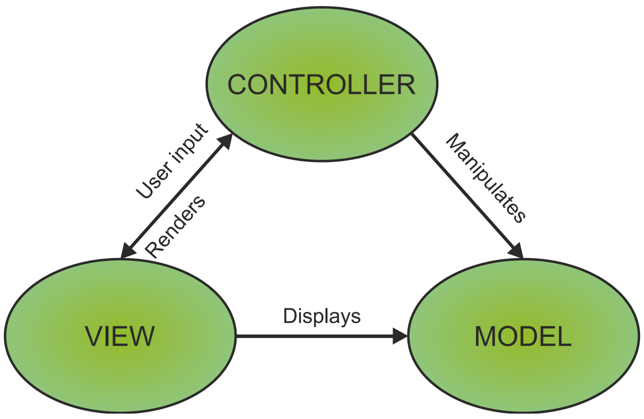
Model — Represents the data of the application. This matches up with the type of data an application is dealing with, such as a user, video, picture, or comment. Changes made to the model notify any subscribed parties within the application.

View — The user interface of the application. In the MVC architecture the view component is treated as dumb, that is, it is should be naive with respect to the data it is being sent — it has no control over it, it simply displays what it is given. In terms of web applications, the view should be a thin layer that sits just on top of the DOM.

Controller — The controller sits between the Model and the View components and is used to handle any user input, such as clicks or browser events. The controller both updates the view, and it updates the model when necessary.

Schematically this translates in most textbooks to Fig 1.

*Fig 1.*



There are, of course many variations on the MVC model which are beyond the scope of this text.

The MVC model has many advantages for programmers, data is independent of the user interface, and, in theory, it decouples the various components of an application so that developers are able to work in parallel on different components without impacting or blocking one another.

There is a caveat to the “in theory” statement above — which is — that’s what *should* happen. However, a problem with MVC patterns in that while it is easy to see and abstract the View and Model layers into well defined objects — the View controls the view, the Model handles the data — it is not as easy to define what should go in the Controller class. For example, what if something isn’t a View or Model component, but also isn’t really a Controller? This question highlights the problem with the MVC system. Inevitably, everything that isn’t easily defined as View or Model ends up in the Controller. So we end up with a complex, bloated, and ill-defined Controller that is difficult to abstract (and reuse) with well defined and smaller View and Model components. Alternatively, if instead of putting ill-defined components in the Controller, one puts them as standalone components, the result is dozens of satellite components that are handled in a spaghetti-like fashion. This makes it very complex for programmers to follow the logic of a program and results in difficult to debug applications and almost no ability to reuse code.

This is perhaps why modern frameworks, such as ReactJS, have attempted to move away from the rigidity of the MVC paradigm and have created a system mainly revolving around V (View), and individual components each handling their own state or data.

Notwithstanding the above criticisms, the MVC paradigm can still be a highly useful tool for developers, and when utilized carefully can produce highly reusable code. This is perhaps best achieved through what I would term MVC++ (borrowing terminology from C++) where the controller layer is not one system, but a multitude of objects with specific and well defined roles within the system.

**TODO LIST APP DESCRIPTION**

The Todo list app on test is built around the MVC architecture with three main classes, *Controller.js, Model.js, View.js*. The *Controller.js* class is passed the context of *Model* and *View* in its constructor thus giving it access to both *View* and *Model* as one would expect in the MVC app framework.

Classes are defined as prototypes in the normal javascript style. The custom methods of each class are detailed in optimizations.

Additionally, data is stored in a *localStorage* file, which allows up to 10MB of storage, more than enough to store hundreds-of-thousands of list items during a session.

At program start-up, or after a change to the data list, the *view.render* method is called between 6 to 8 times depending on what changes have been made. The following table shows the possible arguments passed to the *view.render* method.

| **Command (String)** | **Parameter** |
|---|---|
| “updateElementCount” | Number: of entries e.g. 4 |
| “removeItem” | Object — todo list item e.g. { id:100, title:””, completed:boolean} |
| “clearCompletedButton” | Object {completed: num , visible: boolean} |
| “toggleAll” | Object {checked: boolean} |
| “contentBlockVisibility” | Object {visible:boolean} |
| “showEntries” | Objects — todo list items e.g.[ { id:100, title:””, completed:boolean}, ...] |
| “setFilter” | String: ‘’, ’active’, ‘completed’ |
| “clearNewTodo” | Undefined |
| “elementComplete” | Object {id:#, completed:boolean} |
| “editItem” | Object(id:#, title:””} |
| “editItemDone” | Object(id:#, title:””} |

Lastly, numerous helper programs have been written on top of the core MVC architecture, in many cases obfuscating the MVC architecture and defeating its purpose — that of separation and decoupling of components.

As an example, below is an examination of how the View component handles the adding of a new list item following an *“on change”* event.

1) The program author has added the following objects to the global object in *helper.js*, these are accessed by *view.js*:

*qs, qsa, $on, $parent, $delegate*

2) View overwrites the inherited “bind” prototype method with a custom method which calls *$on* in the global object defined by the *helper.js* program:

```
View.prototype.bind = function (event, handler) {
var self = this;
if (event === 'newTodo') {
$on(self.$newTodo, 'change', function () {handler(self.$newTodo.value);});
```

3) Checking the *helper.js* program shows that the *$on* function does the following:

```
window.$on = function (target, type, callback, useCapture) {
target.addEventListener(type, callback, !!useCapture);
};
```

4) Thus is (2) above, what we actually have is the following event handler:

```
self.$newTodo.addEventListener(“change”,  function(){$self.$newTodo.value});
```

5) We must then follow the code to discover that *$newTodo* is defined somewhere else and actually calls the global *qs* function (unhelpfully defined without the $dollar symbol).

```
this.$newTodo = qs('.new-todo');
```

6) Thus (2) above is actually the following code:

```
qs('.new-todo'). addEventListener(“change”,  function(){qs('.new-todo').value)};
```

7) *qs* is then further defined in *helper.js* to actually be a *querySelector* call for the css class *“.new-todo”* and the value is supplied by Controller as *“function(title){self.addItem(title)}”.*

Thus the event listener is actually defined by the global *“qs”* object which is defined in the *helper.js* program. Controller then declares these functions through a custom call to bind. Essentially, the program author is defining the event listeners in Controller, rather than in View, and is using Helper to pretend they were defined in View.  So Helper is really a sub-Controller of View which is in the global context, thus, partly, so is View. Essentially Controller and View are not as separate as they should be in an MVC framework.

The impact of this spaghetti-like obfuscation will be discussed in the optimizations section of this document.

**TODO LIST APP MODIFICATION**

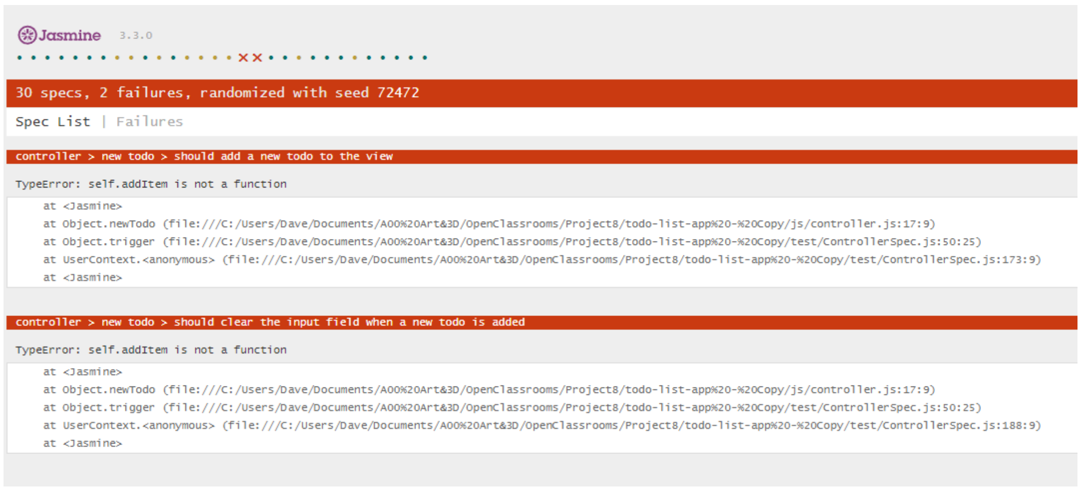
## Bug Fixes

Having taken over the project, and with the installation of JASMINE program testing tools, it became clear that there were a number of bugs, optimizations, and further tests, that needed to be implemented. The bugs are listed individually below:

### 1) controller.js — line 95

*Controller.prototype.adddItem = function (title) {*

CORRECTION: Line should read “addItem” – see Jasmine test failure below.



```
Jasmine 3.3.0
• • • • • • • • • • • • • • • • ✕✕ • • • • • • • • • • • •
30 specs, 2 failures, randomized with seed 72472
Spec List | Failures
controller > new todo > should add a new todo to the view
TypeError: self.addItem is not a function
    at <Jasmine>
    at Object.newTodo (file:///C:/Users/Dave/Documents/A00%20Art&3D/OpenClassrooms/Project8/todo-list-app%20-%20Copy/js/controller.js:17:9)
    at Object.trigger (file:///C:/Users/Dave/Documents/A00%20Art&3D/OpenClassrooms/Project8/todo-list-app%20-%20Copy/test/ControllerSpec.js:50:25)
    at UserContext.<anonymous> (file:///C:/Users/Dave/Documents/A00%20Art&3D/OpenClassrooms/Project8/todo-list-app%20-%20Copy/test/ControllerSpec.js:173:9)
    at <Jasmine>

controller > new todo > should clear the input field when a new todo is added
TypeError: self.addItem is not a function
    at <Jasmine>
    at Object.newTodo (file:///C:/Users/Dave/Documents/A00%20Art&3D/OpenClassrooms/Project8/todo-list-app%20-%20Copy/js/controller.js:17:9)
    at Object.trigger (file:///C:/Users/Dave/Documents/A00%20Art&3D/OpenClassrooms/Project8/todo-list-app%20-%20Copy/test/ControllerSpec.js:50:25)
    at UserContext.<anonymous> (file:///C:/Users/Dave/Documents/A00%20Art&3D/OpenClassrooms/Project8/todo-list-app%20-%20Copy/test/ControllerSpec.js:188:9)
    at <Jasmine>
```

### 2) index.html — “label for”

The “label for” is not associated with an *“Id”* therefore the label will not function when clicked — line 16.

*<input class="toggle-all" type="checkbox">*
*<label for="toggle-all">Mark all as complete</label>*

CORRECTION: Since toggle all is unique, add an ID — Line should read:

*<input class="toggle-all" id="toggle-all" type="checkbox">*

Index.css —label doesn’t display — Line 120.

```
label[for='toggle-all'] {
        display: none;
}
```

CORRECTION: css needs setting to *“display:block;”* or one could simply delete the css.

**3) index.css — check boxes**

The list item custom styled checkboxes are not showing in Google Chrome — Line 187-188
This is because the colours defining stroke and fill in the inline svg in the css use “#” symbol.

```
.todo-list li .toggle:after {
content: url('data:image/svg+xml;utf8,<svg xmlns="http://www.w3.org/2000/svg"
width="40" height="40" viewBox="-10 -18 100 135"><circle cx="50" cy="50" r="50"
fill="none" stroke="#ededed" stroke-width="3"/></svg>');
}

.todo-list li .toggle:checked:after {
content: url('data:image/svg+xml;utf8,<svg xmlns="http://www.w3.org/2000/svg"
width="40" height="40" viewBox="-10 -18 100 135"><circle cx="50" cy="50" r="50"
fill="none" stroke="#bddad5" stroke-width="3"/><path fill="#5dc2af" d="M72 25L42 71 27
56l-4 4 20 20 34-52z"/></svg>');
}
```

CORRECTION: Alter css so HTML colour values are rgb values eg. *stroke="rgb(200,200,200)”*

**4) Missing *“learn.json”* file**

missing file ERR 404 file not found. Unknown significance??? not corrected.

**5) store.js — potentially conflicting “data-id” of list items**

Although unlikely, it is possible that two randomly generated data-ids could conflict — Line 83.

```
// Generate an ID
 var newId = "";
 var charset = "0123456789";
 for (var i = 0; i < 6; i++) {
newId += charset.charAt(Math.floor(Math.random() * charset.length));
}
```

There is a 1 in $1X10^6$ (1 in 1 million) chance. This tiny chance of conflict can be corrected by two methods.

i) check all previous IDs and regenerate number if a match is found. This method has the problem that if there were thousands of IDs, it would be very slow. It must be recognised that, given 1 in 1 million chance, the “issue” would only really be an issue with thousands of items, therefore this solution is not preferred.

ii) Optimize the code by removing the randomization, and instigate a counter so that every new ID is one higher than the item with the current highest ID. This routine can be further optimized by placing it inside the else statement of *“if(id)”* just above *“updateData.id”* thus preventing the unnecessary generation of IDs which happens with the original code.

CORRECTION:

```
//Generate unique ID based on the todo item with the current highest ID (start at 100 to avoid zero)
if (todos.length === 0){
newId = 100;
}
else {newId = todos[todos.length-1].id+1;}

// Assign an ID  (now no need for parseInt(newId) as it is a number already);
updateData.id = newId;
```

**TODO LIST OPTIMIZATIONS**

**1)Modifiy functions to prevent unnecessary calculation.**

As mentioned above, the code in the *Store.js* function defined in *Store.prototype.save* can be modified to prevent calling the newId routine when an Id has been passed to the function. Furthermore, removal of the conversion of a string to a number, iterated in a loop 6 times, and then back to a string for the randomly generated Id is a significant optimization.

It would also be far preferable to dispense with the need to have *"Ids"* in the dataset as the data is an array — it has its own far superior Id system with in-built methods and properties for speedy data manipulation. This is, however, a major undertaking to correct due to the reliance throughout the code on the *"id"* object property in the Model, View, Controller and many of the satellite components (Store.js, template.js etc). This is an example of poorly considered code causing deeply embedded problems (see above discussion on spaghetti MVC).

**2) Optimization of function parameters/arguments**

Another problem with the code, which can be found throughout the code, is the use of the dominant argument, passed to functions, appearing last in the list. The results in the program checking whether the first argument is actually of the second argument type, setting the second argument to either itself or an empty function call, and finally conditionally checking if the first argument is of the type is was supposed to be. See below.

```
Model.prototype.read = function (query, callback) {
var queryType = typeof query;
callback = callback || function () {};

if (queryType === 'function') {
callback = query;
return this.storage.findAll(callback);
} else if (queryType === 'string' || queryType === 'number') {
query = parseInt(query, 10);
this.storage.find({ id: query }, callback);
} else {
this.storage.find(query, callback);
}
};
```

Again, this kind of poor optimization would require a radical rewrite to solve.

**3) Optimization of function calls**

The Todo App has numerous parameter-less functions essentially doing very similar things. I have listed these in groups below where a single function could be written, with the addition of a parameter, to encapsulate the desired functionality thus resulting in much more readable and efficient code.

***Controller.js***
*setView/*
*showAll/showActive/showCompleted*
*addItem/editItem/editItemSave/editItemCancel/*
*removeItem/removeCompletedItems/*
*toggleComplete/toggleAll*
*_updateCount*
*_filter*
*_updateFilterState*

***Model.js***
*create*
*read*
*update/ remove/removeAll*
*getCount*

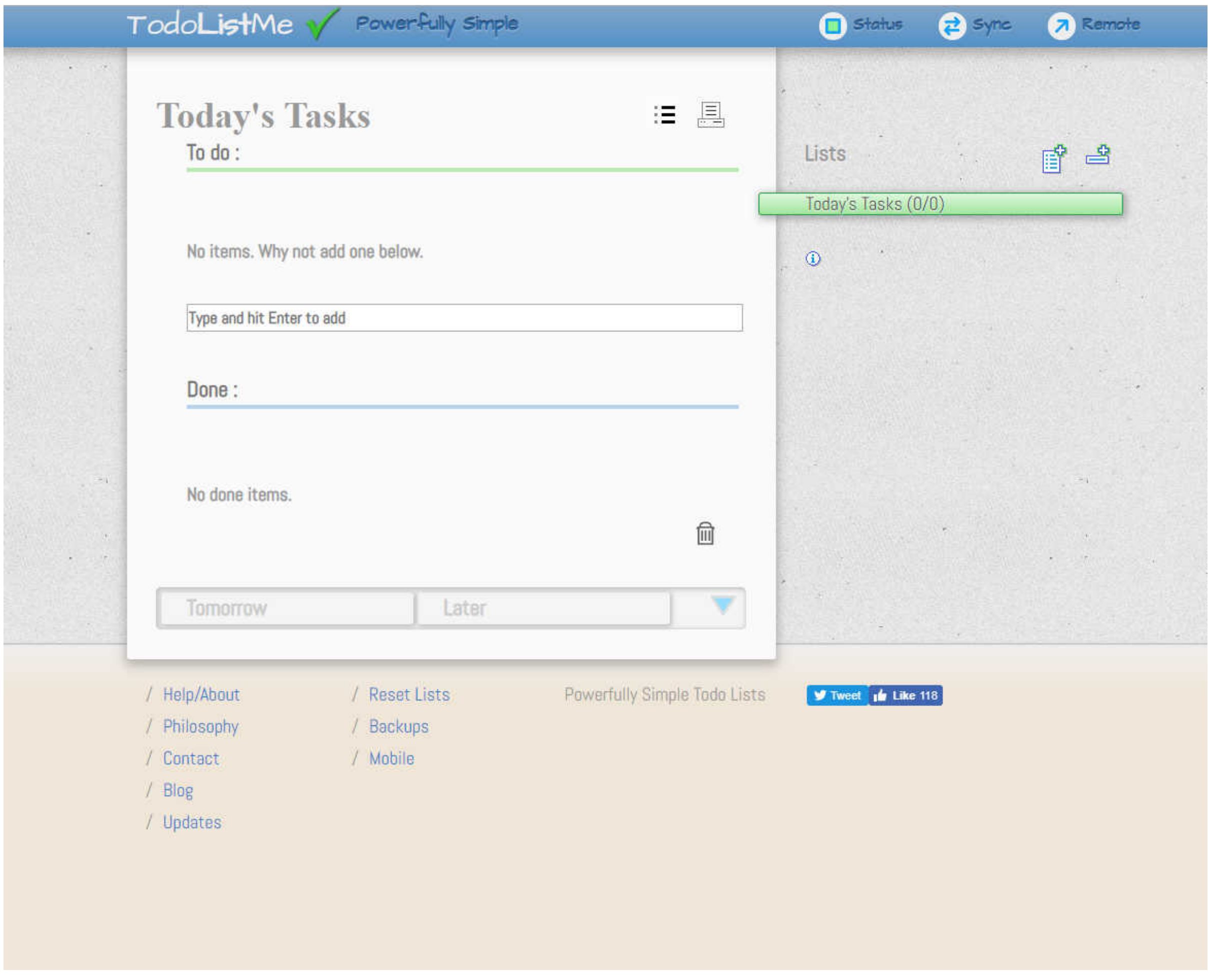
***View.js***
*_removeItem*
*_setFilter*
*_elementComplete*
*_editItem/_editItemDone*
*render*
*_itemId*
*_bindItemEditDone/_bindItemEditCancel*
*bind*

My general comments about the code reflect the above comments. The core program in total (i.e. excluding HTML files and testing files) comprises 8 separate programs totalling over 1200 lines of code. This is for a program that merely performs a basic CRUD operation (create/read/update/destroy) on an array of simple objects. For comparison, a Chess game, including all the rules of chess, can be coded in less than 500 lines of code, or 900 lines of code with complex custom canvas animations.

This is an example of a poorly devised program design in an MVC architecture resulting in a spaghetti pattern of sprawling and bloated code. I have therefore focused only on the most significant flaws, that of data handling, where the majority of program calculation occurs, to optimize the code. Performance will be discussed below.

## PERFORMANCE

Performance is relative, therefore benchmarks will be provided in comparison with a competitor website called *“TodoListMe”* see site screen grab below.



No items. Why not add one below.

No done items.

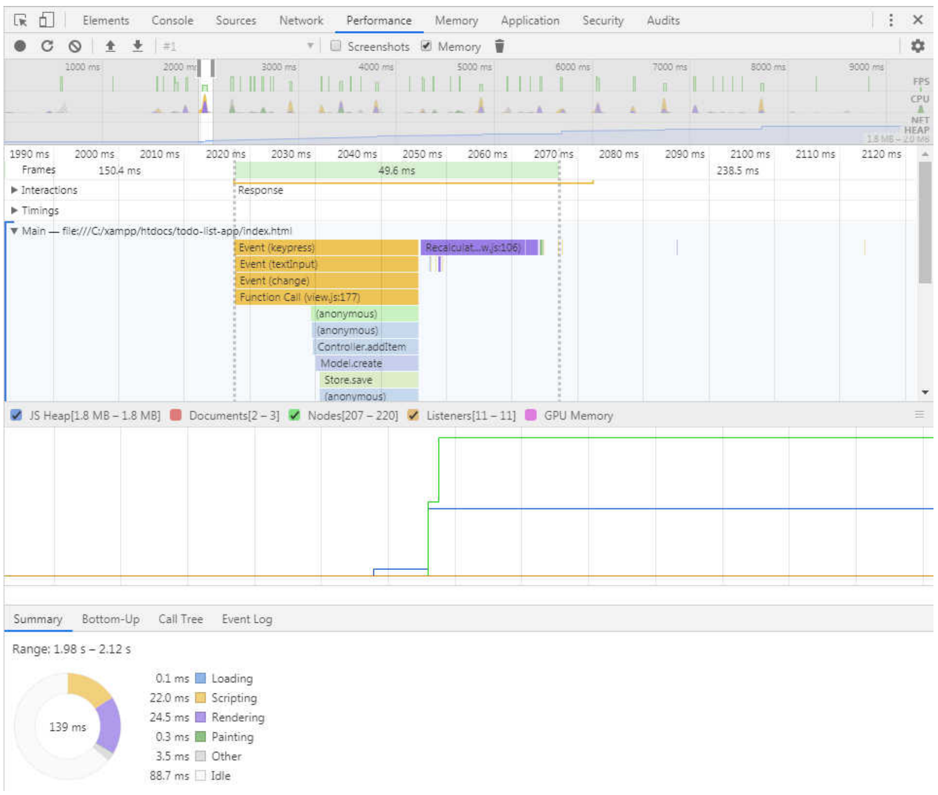
Benchmarks will be provided using Chrome Developer tools and graded on the following:

- Latency (time from first paint until idle).
- Memory usage
- Number of DOM nodes
- Javascript (ms) for tasks (addItem)
- Memory leakage (if any) — test by adding many items and then removing them.

From a UX perspective the most important of the above statistics is the Latency time followed by javascript performance during tasks. Amazon produced statistics in 2009 which showed that they lost a projected 1% of revenue for every 100ms of latency[1]. Simply put, customers want highly responsive and speedy websites.

[1] *“Amazon found every 100ms of latency cost them 1% in sales”*, Yoav Einav, 1.20.2019 www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/

| | **Todo MVC** | **ToDoListMe (with ads/without ads)** |
|---|---|---|
| **Latency (time from first paint until idle).** | 76.4ms | 902.4ms |
| **Memory usage (steady state after initial load)** | 2MB | 13.3MB /7.5MB |
| **Number of DOM nodes (idle, no list items)** | 273 nodes | 1649 nodes/ 1400 nodes |
| **Number of Listeners(at idle)** | 11 | 202/ 140 |
| **Scripting (ms) idle per second** | 0ms | 202.8ms/ 21ms |
| **Javascript add item function call time (ms)** | 22ms | 10ms-20ms/75ms |
| **Rendering& Patinting (ms) per second** | 24.5ms | 24.2ms |
| **Memory leakage (if any)** | No memory leaks detected | No memory leaks detected |



Range: 1.98 s – 2.12 s

139 ms

| | | |
|---|---|---|
| 0.1 ms | 🟦 | Loading |
| 22.0 ms | 🟨 | Scripting |
| 24.5 ms | 🟪 | Rendering |
| 0.3 ms | 🟩 | Painting |
| 3.5 ms | ⬜ | Other |
| 88.7 ms | ⬜ | Idle |

## PERFOMANCE CONCLUSION

There is some difficulty in comparing *ToDoListMe* with *TodoMVC* in that *ToDoListMe* has an ads plug-in installed in their site. This plug-in uses approximately half of the available memory and a sizeable number of nodes. This plug-in will also significantly impair the latency of the site.

Notwithstanding the above, three conclusion are clear from the data.

(Note that the figures and times obtained are only a rough guide as they come from Chrome Dev tools. For more accurate statistics an in-app “get time” function, such as performance.now() and requestAnimationFrame(function), should be used to mark the time of function calls and first load until application ready state. Times are also relative to platform. These tests were carried out on Pentium i7 8GB memory, SSD 512MB, CPU Passmark ~10000.)

1) *TodoMVC* has a lower latency for both time until “content Loaded” and “application idle”. The latency is 76.4ms. The lower latency is due to the small file size and lack of pictures, fonts, and other loaded elements. I also tested a similar simple text-based web application for a fair comparison. It had a latency of just 0.7ms. So, when compared like-for-like, *TodoMVC* is actually very slow and the comparison with *ToDoListMe* belies the poor performance of the actual code in *TodoMVC*.

2)*TodoMVC* has a comparatively high time for its javascript functions when calling *“additem”.* Whilst a time of 22ms may not seem long, this must be compared against *ToDoListMe* App which had a time of just 10ms (interestingly it took longer with ads than without them implying that the ad blocker in developer tools caused interference).

Taking the best results of each application, the difference in time significant. When these result are translated from the high speed computer used in these tests to a mobile device, it could result very poor performance in *TodoMVC*.

3)*TodoListMe* has far more functionality in terms of its UI with an ability to drag and drop list items, add multiple lists, and also has animations etc.

The above conclusions are not unexpected and are in-line with the findings of the spaghetti-like code base in *TodoMVC*. Given also that *TodoListMe* uses *JQuery* for some of its functionality, a framework known for sluggish performance, it can only be concluded that *TodoMVC* is a very poor performer in these tests. This is due, in part, to the numerous *“callback”* functions.

For comparison, a test on *“add item to list”* on the W3C school website gave a time of just 4ms, 3.92ms of which was the modification to the DOM with the InnerHTML function call. Since the modification of the DOM is the slow bit (or should be) the comparison in actual javascript time is 18ms for *TodoMVC* vs 0.08ms (80µs) for performant vanillaJS — thus *TodoMVC* is hundreds of times slower than it should be. I conducted similar tests for both *TodoMVC* and a text based application using a custom “*performance.now()”* function. It gave similar result of 50-100 times slower code for *TodoMVC*.