Documentation technique

Comment l'application est construite ??

– Notre application de liste des tâches est basée et structurée sur une méthode appelée l’architecture "MVC", qui est traduite par modèle, vue et contrôleur.

Qu'est-ce que le modèle « MVC » ??

– Comme on le voit sur l’image ci-dessous, l'utilisateur interagit avec l'interface de l'application, qui est la vue dans laquelle toutes les données et fonctionnalités de l'application sont chargées.

– Si par exemple, l'utilisateur va ajouter une tâche à l’application, la vue communiquera avec le contrôleur et à son tour, le contrôleur communiquera avec le modèle pour ajouter et sauvegarder la nouvelle tâche.

– Comme nous pouvons le voir sur l’image, le contrôleur sert de pont pour relier la vue au modèle et relier chaque action entre les deux, de sorte que dès que l'utilisateur interagira avec une ou plusieurs des fonctionnalités de l'application, la vue et le modèle seront respectivement notifié et mis à jour a l'aide du contrôleur.

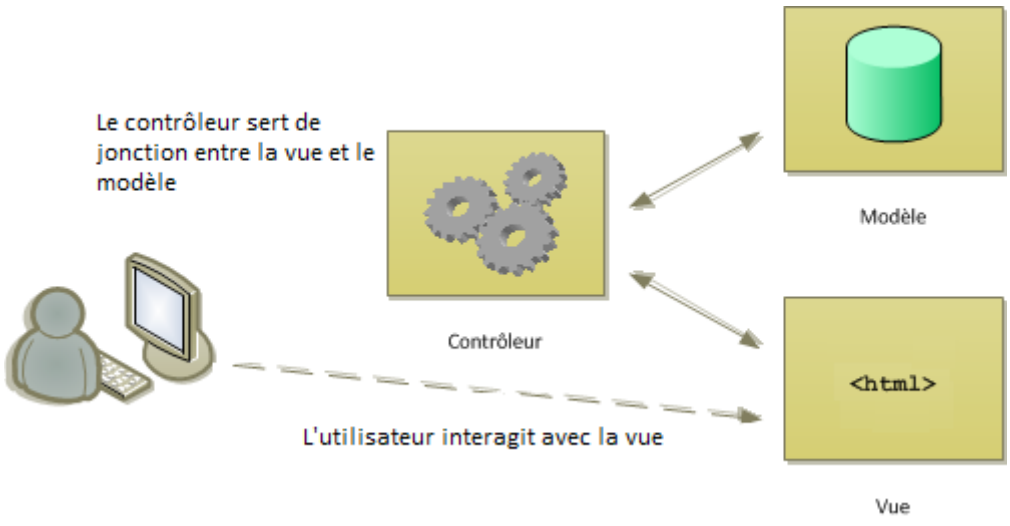


Schéma de l’architecture « MVC »

Quelle version de javascript est utilisée sur l'application ??

– En ce qui concerne la version utilisée sur l'application, nous pouvons clairement voire qu'il s'agit d'ES5 ou comme on l'appelle également : « ECMAScript 2009 » et sa se traduit aussi par la structure utilisés dans le code et la façon dont la programmation orientés objets est présentée.

```
Template.prototype.show = function (data) {
    var view = '';

    for (var i = 0; i < data.length; i++) {
        var template = this.defaultTemplate;
        var completed = '';
        var checked = '';
```

Image d'exemple de la version ES5 de l'application

Sur quelle norme le code est-il écrit ??

– La norme utiliser dans l'application est la « camelCase », elle consiste à écrire des phrases de manière à ce que chaque mot ou abréviation figurant au milieu de la phrase commence par une lettre majuscule, sans espace ni ponctuation.

– En parcourant le code, nous pouvons constater que la norme « camelCase » est fortement utilisée dans l'application, comme s'est présenter dans les images ci-dessous.

```
View.prototype.render = function (viewCmd, parameter) {
    var self = this;
    var viewCommands = {
        showEntries: function () {
            self.$todoList.innerHTML = self.template.show(parameter);
        },
        removeItem: function () {
            self._removeItem(parameter);
        },
        updateElementCount: function () {
            self.$todoItemCounter.innerHTML = self.template.itemCounter(parameter);
        },
        clearCompletedButton: function () {
            self._clearCompletedButton(parameter.completed, parameter.visible);
        },
        contentBlockVisibility: function () {
            self.$main.style.display = self.$footer.style.display = parameter.visible ? 'block' : 'none';
        },
        toggleAll: function () {
            self.$toggleAll.checked = parameter.checked;
        },
        setFilter: function () {
            self._setFilter(parameter);
        },
        clearNewTodo: function () {
            self.$newTodo.value = '';
        },
```

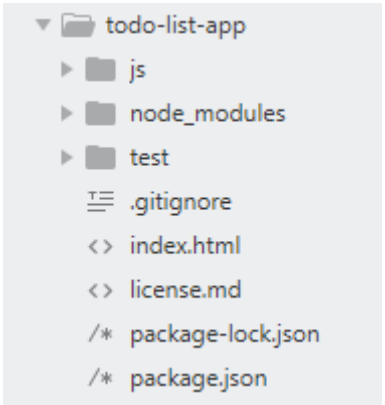
Source : le fichier view.js du dossier js

```
Controller.prototype.showAll = function () {
    var self = this;
    self.model.read(function (data) {
        self.view.render('showEntries', data);
    });
};

/**
 * Renders all active tasks
 */
Controller.prototype.showActive = function () {
    var self = this;
    self.model.read({ completed: false }, function (data) {
        self.view.render('showEntries', data);
    });
};

/**
 * Renders all completed tasks
 */
Controller.prototype.showCompleted = function () {
    var self = this;
    self.model.read({ completed: true }, function (data) {
        self.view.render('showEntries', data);
    });
};
```

Source : le fichier controller.js du dossier js

Comment l'arborescence de l'application est structurée ??

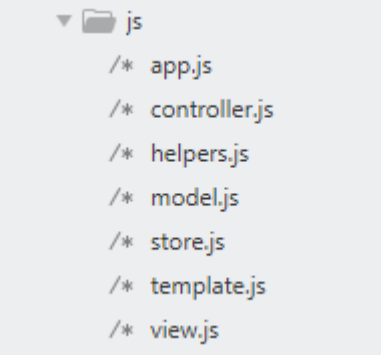
– Comme on peut le voir sur l'image ci-dessus, l'arborescence des fichiers de l'application facilite la compréhension de son fonctionnement et de la manière dont les fichiers sont connectés les uns aux autres.



– En premier lieu, nous avons le fichier html de l'application nommé ici index.html avec d'autres fichiers personnalisés, tels que: package.json, licence.md..ect

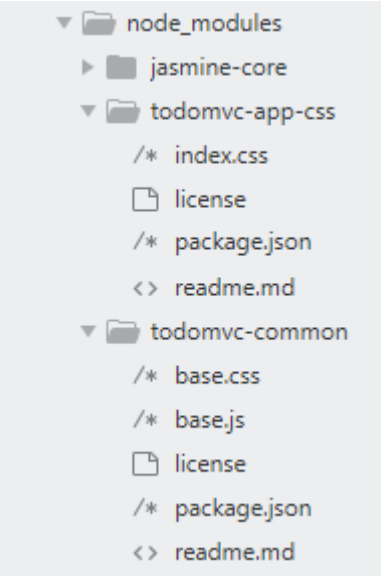
– La manière dont ces fichiers ne se trouvent pas dans des dossiers enfants du dossier parent de l'application permet de les modifier et de corriger certains bugs mineurs facilement pour des mises à jour ultérieures de l'application.

– En second lieu, nous avons le dossier js qui est un dossier enfant du dossier principal de l'application. Il contient tous les fichiers js personnalisés dont l'application a besoin pour fonctionner correctement.

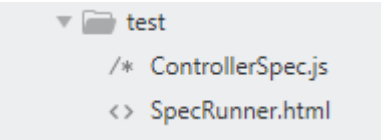


– Ensuite nous avons le dossier node_modules de l'application qui contiennent 3 sous-dossiers:

1. Le dossier jasmine-core qui gérera les tests de l'application (expliqué ci-dessous).
2. Le dossier todomvc-app-css qui inclut le style css de l'application.
3. Le dossier todomvc-common qui inclut deux autres fichiers (base.js et base.css) permettant d'ajouter du style et d'ajouter le code javascript nécessaire au bon fonctionnement de l'application.



– Enfin, nous avons le dossier de test qui contient nos tests Jasmine dans le fichier ControllerSpec.js et le fichier SpecRunner.html qui affichent le résultat des tests sur le navigateur.



Quels sont les fichiers les plus importants de l'application ??

– Comme pour tous sites web, chaque fichier est important et il en va de même pour notre application. Dans notre dossier js expliqué ci-dessus, nous avons nos fichiers js personnalisés. Chacun de ces fichiers a une tâche importante pour l'application, en voici quelques-uns:

1) Le fichier store.js qui contient la classe « Store »

```
/**
 * Will remove an item from the Store based on its ID
 *
 * @param {number} id The ID of the item you want to remove
 * @param {function} callback The callback to fire after saving
 */
Store.prototype.remove = function (id, callback) {
    var data = JSON.parse(localStorage[this._dbName]);
    var todos = data.todos;

    for (var i = 0; i < todos.length; i++) {
        if (todos[i].id == id) {
            todos.splice(i, 1);
        }
    }

    localStorage[this._dbName] = JSON.stringify(data);
    callback.call(this, todos);
};
```

Sa methode « remove » supprime une todo de la base de données fictive qui contient tous les todos crées par l'utilisateur

2) le fichier template.js qui contient la classe « Template »

```
/**
 * Displays a counter of how many to dos are left to complete
 *
 * @param {number} activeTodos The number of active todos.
 * @returns {string} String containing the count
 */
Template.prototype.itemCounter = function (activeTodos) {
    var plural = activeTodos === 1 ? '' : 's';

    return '<strong>' + activeTodos + '</strong> item' + plural + ' left';
};
```

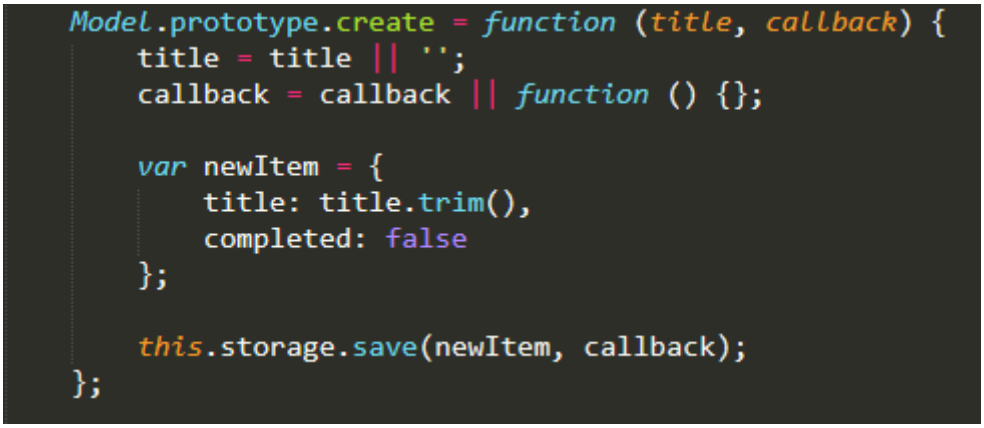
Sa methode « itemCounter » compte et affiche le nombre de
tâches à compléter comme indiqué sur cette image

– Mais vu que notre application est construite sur une base « MVC », les fichiers les plus important sont model.js, view.js et controler.js

1) Pour ce qui concerne model.js, son rôle dans notre application est de créer, éditer, mettre à jour et aussi de supprimer une tâche de notre liste de todo. (voir les images ci-dessous)

```
Model.prototype.update = function (id, data, callback) {
    this.storage.save(data, callback, id);
};

/**
 * Removes a model from storage
 *
 * @param {number} id The ID of the model to remove
 * @param {function} callback The callback to fire when the removal is complete.
 */
Model.prototype.remove = function (id, callback) {
    this.storage.remove(id, callback);
};

/**
 * WARNING: Will remove ALL data from storage.
 *
 * @param {function} callback The callback to fire when the storage is wiped.
 */
Model.prototype.removeAll = function (callback) {
    this.storage.drop(callback);
};
```

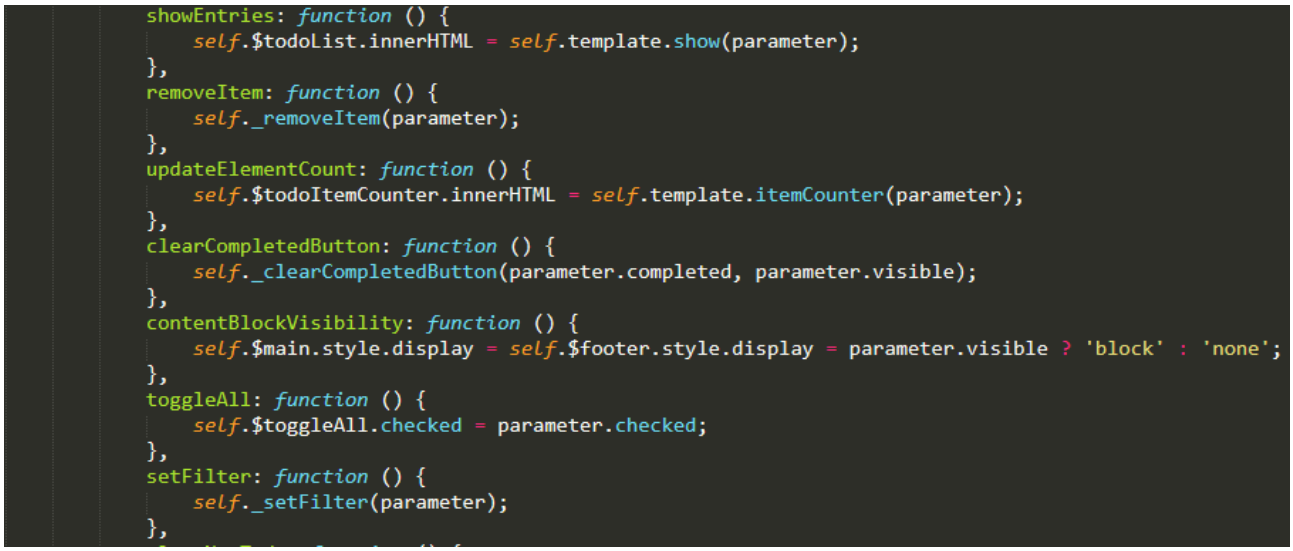
Image (1) : quelque méthodes de la classe « Model »



```
Model.prototype.create = function (title, callback) {
    title = title || '';
    callback = callback || function () {};

    var newItem = {
        title: title.trim(),
        completed: false
    };

    this.storage.save(newItem, callback);
};
```

Image (2) : quelque méthodes de la classe « Model »

2) Ensuite nous avons le fichier view.js qui va mettre a jour le rendu pour l'utilisateur selon ses actions, exemple : ajouter une tâche va faire apparaître celle-ci dans la liste des tâches a faire, afficher un bouton pour supprimer toutes les tâches terminer..ect (voir les images ci-dessous)



```
showEntries: function () {
    self.$todoList.innerHTML = self.template.show(parameter);
},
removeItem: function () {
    self._removeItem(parameter);
},
updateElementCount: function () {
    self.$todoItemCounter.innerHTML = self.template.itemCounter(parameter);
},
clearCompletedButton: function () {
    self._clearCompletedButton(parameter.completed, parameter.visible);
},
contentBlockVisibility: function () {
    self.$main.style.display = self.$footer.style.display = parameter.visible ? 'block' : 'none';
},
toggleAll: function () {
    self.$toggleAll.checked = parameter.checked;
},
setFilter: function () {
    self._setFilter(parameter);
},
```

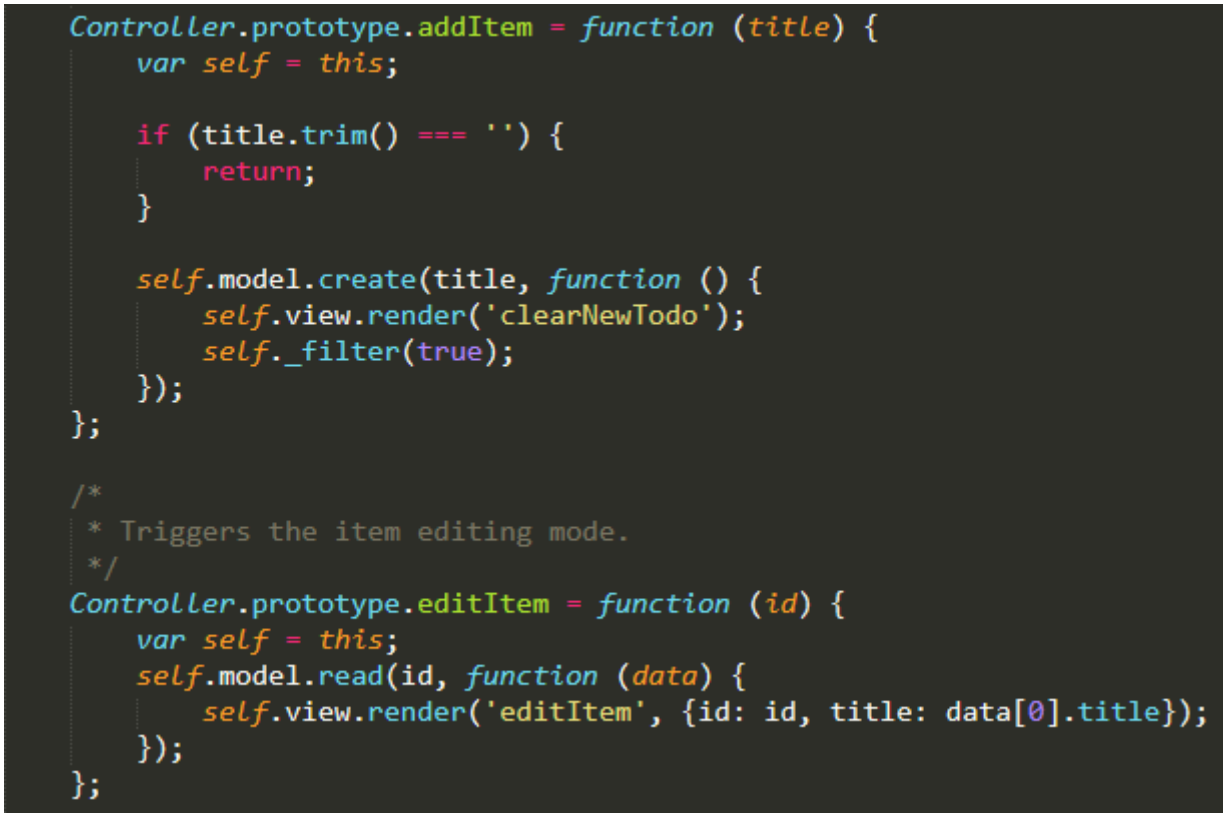
Image (1) : quelque méthodes de la classe « View »

```
View.prototype._bindItemEditDone = function (handler) {
    var self = this;
    $delegate(self.$todoList, 'li .edit', 'blur', function () {
        if (!this.dataset.iscanceled) {
            handler({
                id: self._itemId(this),
                title: this.value
            });
        }
    });

    $delegate(self.$todoList, 'li .edit', 'keypress', function (event) {
        if (event.keyCode === self.ENTER_KEY) {
            // Remove the cursor from the input when you hit enter just like if it
            // were a real form
            this.blur();
        }
    });
};
```

Image (2) : quelque méthodes de la classe « View »

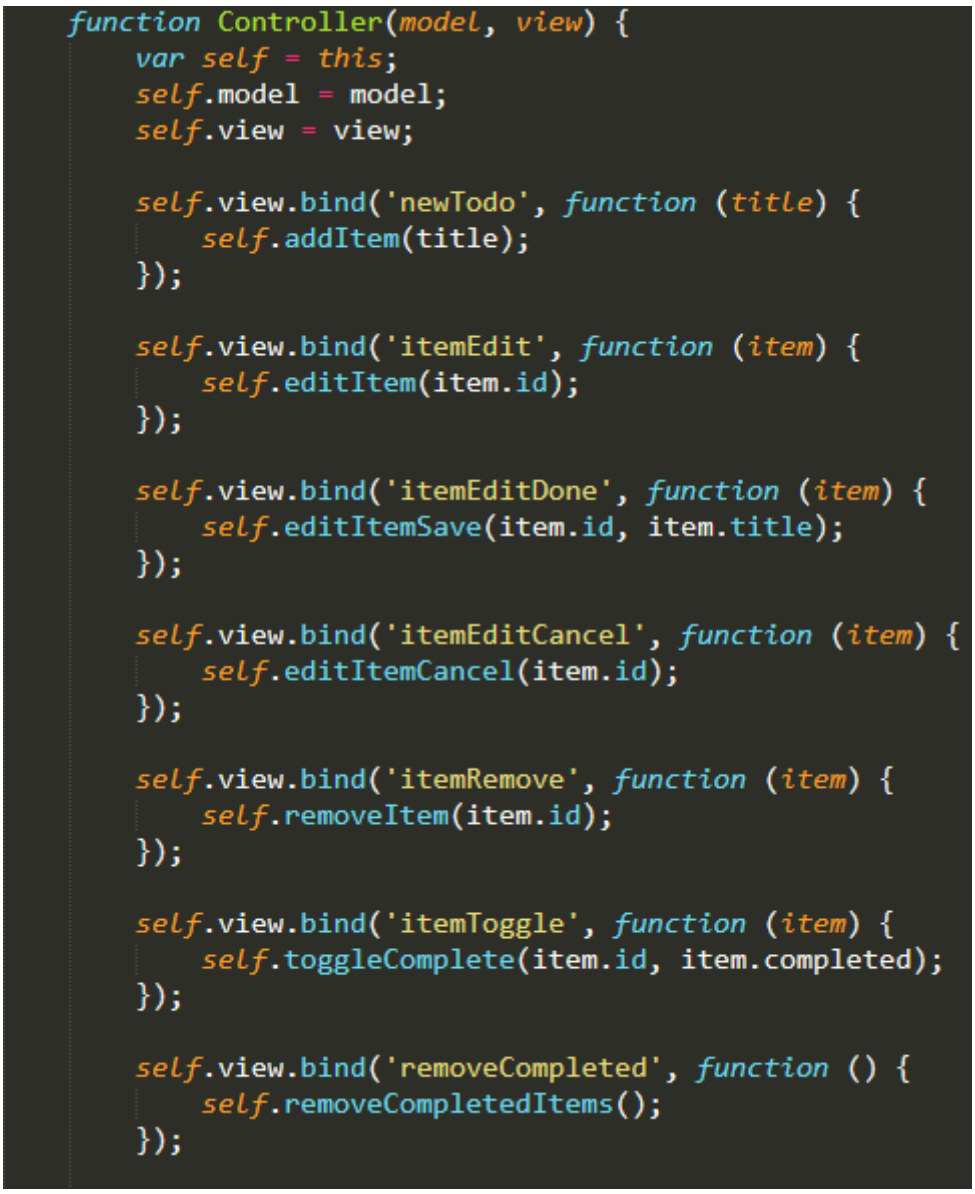
3) Enfin nous avons le fichier controller.js qui connecte la vue au modèle.

– Il sert de jonction entre la vue et le modèle lors de la création, l'édition et la suppression d'une tâche, et assure la bonne coordination lors du traitement des données saisie ou événements déclencher par l'utilisateur. (voir les images ci-dessous)



```
Controller.prototype.addItem = function (title) {
    var self = this;

    if (title.trim() === '') {
        return;
    }

    self.model.create(title, function () {
        self.view.render('clearNewTodo');
        self._filter(true);
    });
};

/*
 * Triggers the item editing mode.
 */
Controller.prototype.editItem = function (id) {
    var self = this;
    self.model.read(id, function (data) {
        self.view.render('editItem', {id: id, title: data[0].title});
    });
};
```

Image (1) : quelque méthodes de la classe « Controller »



```
function Controller(model, view) {
    var self = this;
    self.model = model;
    self.view = view;

    self.view.bind('newTodo', function (title) {
        self.addItem(title);
    });

    self.view.bind('itemEdit', function (item) {
        self.editItem(item.id);
    });

    self.view.bind('itemEditDone', function (item) {
        self.editItemSave(item.id, item.title);
    });

    self.view.bind('itemEditCancel', function (item) {
        self.editItemCancel(item.id);
    });

    self.view.bind('itemRemove', function (item) {
        self.removeItem(item.id);
    });

    self.view.bind('itemToggle', function (item) {
        self.toggleComplete(item.id, item.completed);
    });

    self.view.bind('removeCompleted', function () {
        self.removeCompletedItems();
    });
```

Image (2) : quelque méthodes de la classe « Controller »