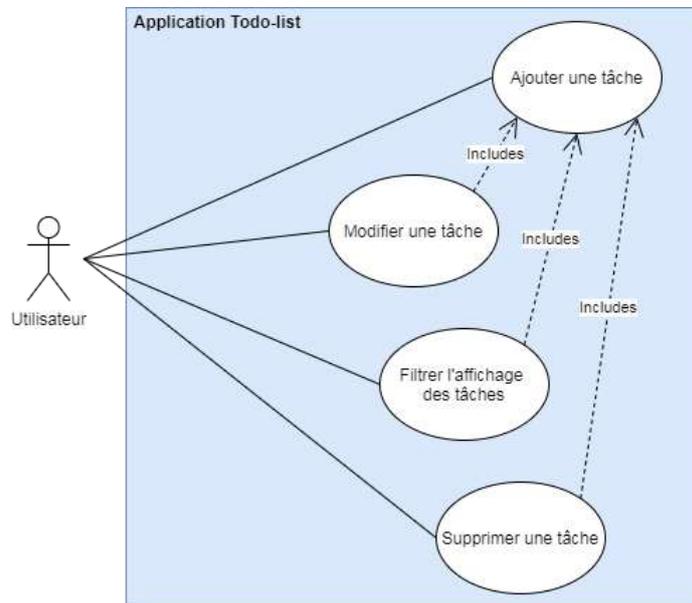


Documentation technique

1. Présentation :



L'application To-do list permet à un utilisateur de créer une liste de tâches. Il peut ajouter, modifier ou supprimer des tâches, mais aussi choisir de n'afficher que celles qui lui restent à effectuer. Les tâches restent stockées dans son navigateur, il pourra donc les retrouver même après avoir quitté la page.

2. Architecture du projet :

Le projet utilise des technologies spécifiques, détaillées dans les parties suivantes.

```
▼ TODO-LIST-APP
  > docs
  > documentation
  > js
  > node_modules
  > src
  > test
  ◆ .gitignore
  <> index.html
  {} jsdoc.json
  ⚡ license.md
  {} package.json
  {} package-lock.json
```

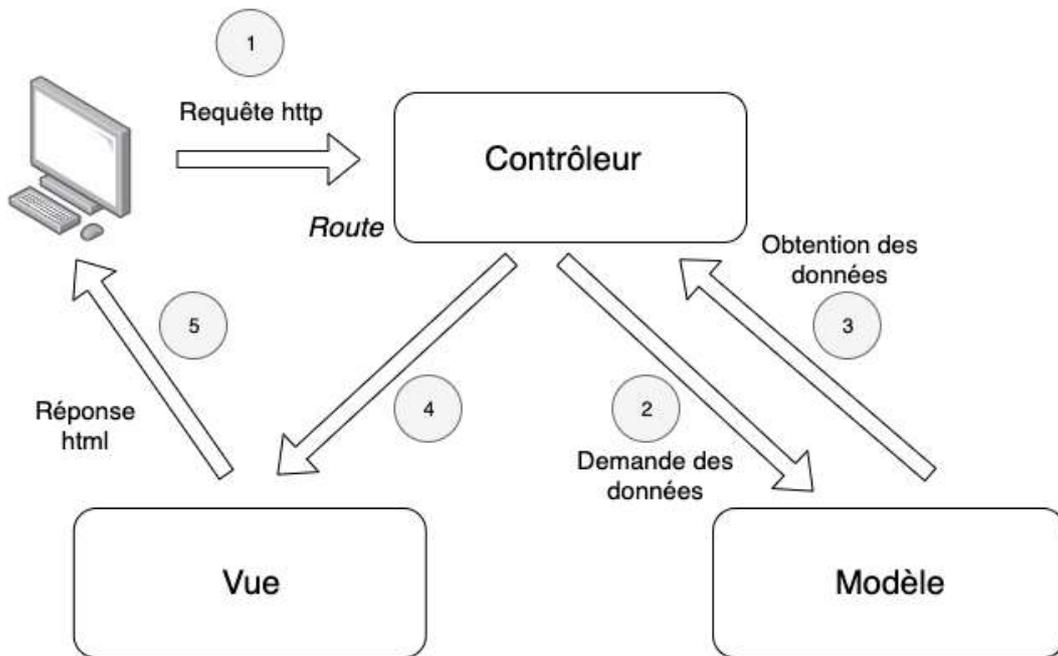
Le cœur du code se situe dans les fichiers JavaScript

```
▼ TODO-LIST-APP
  > docs
  > documentation
  ▼ js
    JS app.js
    JS controller.js
    JS helpers.js
    JS model.js
    JS store.js
    JS template.js
    JS view.js
  > node_modules
  > src
  > test
  ◆ .gitignore
  <> index.html
  {} jsdoc.json
  ⚡ license.md
  {} package.json
  {} package-lock.json
```

3. L'architecture MCV

Ou "**Modèle-Vue-Contrôleur**" : c'est un motif d'architecture logicielle destiné aux interfaces graphiques, composé de trois types de modules ayant trois responsabilités différentes :

- Un modèle (Model), qui contient les données à afficher.
- Une vue (View), qui contient la présentation de l'interface graphique.
- Un contrôleur (Controller), qui contient la logique concernant les actions effectuées par l'utilisateur.



Nous retrouvons ce fonctionnement dans les classes JavaScript du projet :

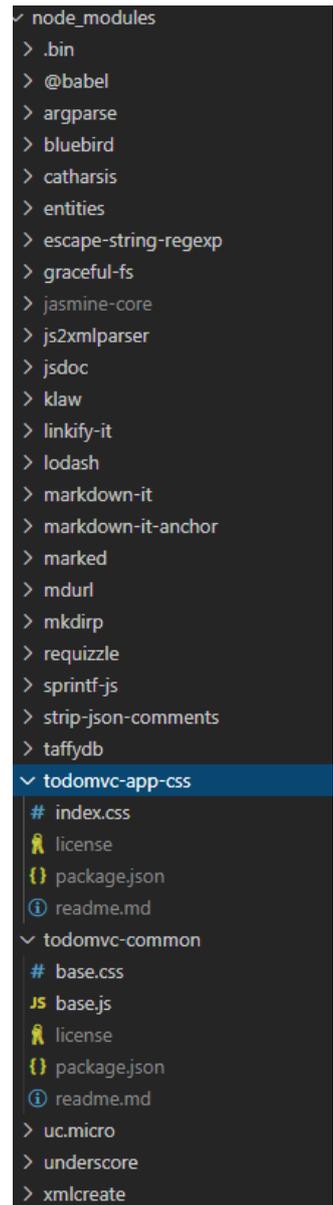
```
@class Model @class View { @class Controller
```

4. Gestionnaire de paquets npm

Le projet utilise npm, le gestionnaire de paquets officiel de Node.js. Il permet de télécharger et d'installer des paquets (**node_modules, dans l'arborescence ci-contre**) pour les utiliser pour un projet ou pour les partager avec d'autres utilisateurs.

Les différents paquets installés pour un projet sont détaillés dans un fichier **package.json**.

Chacun de ces paquets est régulièrement mis à jour (ce qui peut changer l'apparence de notre projet).



5. JSHint

JSHint est utilisé dans le projet, pour analyser le code de façon statique (c'est-à-dire sans l'exécuter). JSHint vérifie notamment si le code correspond aux bonnes pratiques.



Des options peuvent être utilisées, telles que `/* jshint ignore */`, qui ignore une partie du code :

```
1  /* global _ */
2  (function () {
3      'use strict';
4
5      /* jshint ignore:start */
6      // Underscore's Template Module
7      // Courtesy of underscorejs.org
8  > var _ = (function (_) { ...
114  })({});
115
116 > if (location.hostname === 'todomvc.com') { ...
123  }
124  /* jshint ignore:end */
```

les lignes 5 à 124 sont ignorées

Ou encore `/*global*/`, qui spécifie une liste de variables globales qui ne sont pas définies formellement dans le code source :

```
1  /*global app, $on */
```

6. Le mode strict

```
(function (window) {
    'use strict';
```

Le projet est codé entièrement en mode strict, qui apporte quelques changements à la sémantique de JavaScript. Cette écriture va permettre de mieux détecter des erreurs que le moteur JavaScript aurait laissé passer en mode "normal".

7. Le localStorage

```
60  * Will retrieve all data from the collection
61  *
62  * @param {function} callback The callback to fire upon retrieving data
63  */
64  Store.prototype.findAll = function (callback) {
65      callback = callback || function () {};
66      callback.call(this, JSON.parse(localStorage[this._dbName]).todos);
67  };
```

Le localStorage permet de stocker des données dans le navigateur. Les données sont persistantes d'une session à l'autre (à la différence du sessionStorage). Cela évite d'avoir recours à une base de données, ce qui rend l'utilisation particulièrement intéressante dans le développement frontend.

```

6      * Creates a new client side storage object and will create an empty
7      * collection if no collection already exists.
8      *
9      * @param {string} name The name of our DB we want to use
10     * @param {function} callback Our fake DB uses callbacks because in
11     * real life you probably would be making AJAX calls
12     */
13     function Store(name, callback) {
14         callback = callback || function () {};
15
16         this._dbName = name;
17
18         if (!localStorage[name]) {
19             var data = {
20                 todos: []
21             };
22
23             localStorage[name] = JSON.stringify(data);
24         }
25
26         callback.call(this, JSON.parse(localStorage[name]));
27     }

```

Ces données sont au format JSON est doivent être converties en String pour être stockées.

8. JSDoc

```

118
119     * @param {number} id The ID of the item you want to remove
120     * @param {function} callback The callback to fire after saving
121     */

```

JSDoc est un langage de balisage utilisé pour documenter le code du projet, à l'aide de certains tags comme le **@param**

Class: Model

Model(storage)

new Model(storage)

Creates a new Model instance and hooks up the storage.

Parameters:

Name	Type	Description
storage	object	A reference to the client side storage class

Source: [model.js, line 4](#)

Methods

create(title_{opt}, callback_{opt})

Creates a new todo model

Parameters:

Name	Type	Attributes	Description
title	string	<optional>	The title of the task
callback	function	<optional>	The callback to fire after the model is created

Source: [model.js, line 19](#)

```

v TODO-LIST-APP
v docs
> fonts
> scripts
> styles
< app.js.html
< Controller.html
< controller.js.html
< global.html
< helpers.js.html
< index.html
< Model.html
< model.js.html
< Store.html
< store.js.html
< Template.html
< template.js.html
< Todo.html
< View.html
< view.js.html
> documentation
> js
> node_modules
> src
> test
< .gitignore
< index.html
< jsdoc.json
< license.md
< package.json
< package-lock.json

```

9. Jasmine

Jasmine est un framework qui a permis d'effectuer des test unitaires du code du projet et ainsi lui assurer une meilleure stabilité.



```
Jasmine 2.99.0 options
-----
32 specs, 0 failures Finished in 0.009s

controller
  should show entries on start-up

routing
  should show all entries without a route
  should show all entries without "all" route
  should show active entries
  should show completed entries

  should show the content block when todos exists
  should hide the content block when no todos exists
  should check the toggle all button, if all todos are completed
  should set the "clear completed" button
  should highlight "All" filter by default
  should highlight "Active" filter when switching to active view
  should highlight "Completed" filter when switching to completed view

toggle all
  should toggle all todos to active
  should toggle all todos to completed
  should update the view

new todo
  should add a new todo to the model
  should add a new todo to the view
  should clear the input field when a new todo is added

element removal
  should remove an entry from the model
  should remove an entry from the view
  should update the element count

remove completed
  should remove a completed entry from the model
  should remove a completed entry from the view

element complete toggle
  should update the model
  should update the view

edit item
  should switch to edit mode
  should leave edit mode on done
  should persist the changes on done
  should remove the element from the model when persisting an empty title
  should remove the element from the view when persisting an empty title
  should leave edit mode on cancel
  should not persist the changes on cancel
```

```
▼ TODO-LIST-APP
  > docs
  > documentation
  > js
  > node_modules
  > src
  ▼ test
    JS ControllerSpec.js
    <> SpecRunner.html
    .gitignore
    <> index.html
    {} jsdoc.json
    license.md
    {} package.json
    {} package-lock.json
```