Documentation fonctionnel de l'application

L'application todos permet de créer et de gérer une liste de tâches à l'aide de fonctionnalités simples. Les tâches sont stockées dans le localStorage, il suffit d'utiliser toujours le même navigateur pour les retrouver. Les fonctionnalitées suivante sont :

- Ajouter une tâche
- Modifier une tâche
- Supprimer une tâche
- Terminer une ou plusieurs tâche(s)
- Supprimer toutes les tâches terminées
- Filtrer les tâches

1. Ajouter une tâche

Pour ajouter une tâche, il suffit de cliquer sur le champ What needs to be done?, d'écrire votre tâche et d'appuyer sur la touche Entrée du clavier ou de cliquer en dehors du champ de saisie. La tâche créée s'ajoute à la liste

2. Modifier une tâche

Pour modifier une tâche (ou l'éditer), double-cliquez sur la tâche pour afficher le champ d'édition. Une fois la modification effectuée, il suffit d'appuyer sur la touche Entrée du clavier ou de cliquer en dehors du champ de saisie pour qu'elle soit prise en compte. Si le champ n'est plus renseigné, la tâche sera supprimée. Si vous appuyez sur echappe, votre modification ne sera pas prise en compte.

3. Supprimer une tâche

Pour supprimer une tâche, il suffit de cliquer sur la croix qui apparaît sur la droite de la tâche à supprimer.

4. Terminer une ou plusieurs tâche(s)

Pour terminer une tâche (ou la compléter), il suffit de cliquer dans le cercle de la tâche terminée qui se situe à gauche de chaque tâche. Il est également possible de marquer les tâches comme terminées toutes à la fois en cliquant sur la flèche .toggle-all

Il est également possible de rétablir les tâches comme actives en cliquant de nouveau soit sur le cercle soit sur la flèche vers le bas.

5. Supprimer toutes les tâches terminées

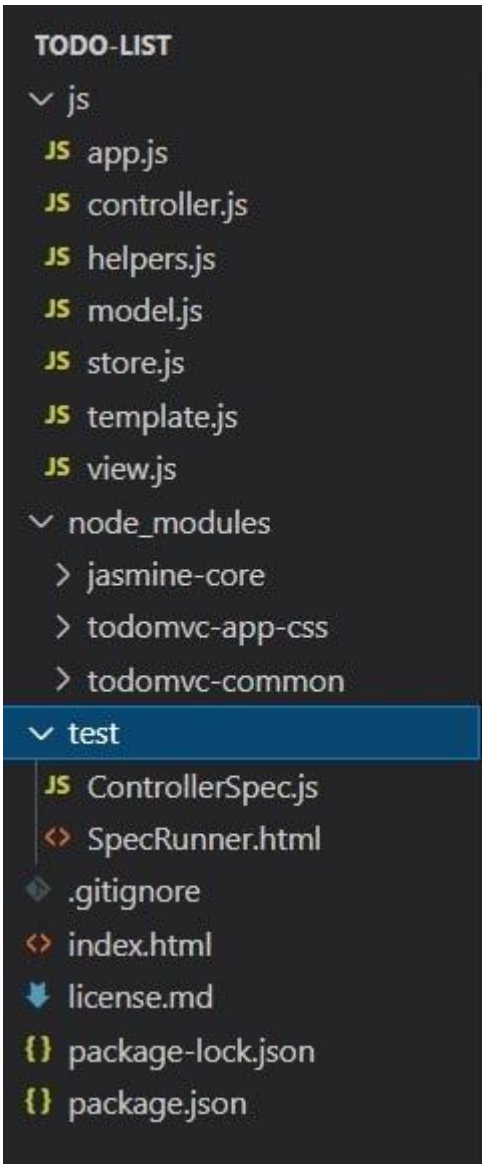
Toutes les tâches ayant le statut terminé (« completed ») peuvent être supprimées d'un seul coup. Il suffit de cliquer sur le bouton Clear completed qui apparaît en bas à droite dès qu'une tâche au moins est terminée.

6. Filtrer les tâches

L'application propose trois boutons sous la liste de tâches. Il s'agit de trois filtres qui permettent d'afficher les tâches selon leur état :

- Le filtre All permet d'afficher toutes les tâches quel que soit leur état de réalisation.
- Le filtre Active permet de n'afficher que les tâches actives, c'est-à-dire non terminées.
- Le filtre Completed permet de n'afficher que les tâches terminées.

Explicatif technique de l'architecture de l'application



L'arborescence ci-dessous présente les principaux dossiers et fichiers du projet :

Le site est développé en HTML5 pour le fichier html, en CSS3 pour les fichiers de style et en JavaScript (ECMAScript 5) pour les fichiers fonctionnels.

● Le fichier index.html dispose des élément de base de l'application ainsi que de tous les appelles js, css

● Le fichier package.json liste les dépendances nécessaires à installer avec npm.

● Le dossier js contient tous les fichiers JavaScript nécessaires pour le fonctionnement de l'application.

● Le dossier node_modules contient tous les packages node / dépendances nécessaires au projet : les feuilles de style CSS de l'application et le Framework Jasmine pour la réalisation des tests unitaires.

● Le dossier test contient les tests unitaires qui utilisent Jasmine : un fichier JavaScript et une page html pour visualiser les résultats de tests

L'application est construite sur le design pattern (masque de conception) MVC. Le pattern MVC permet de séparer la logique du code en trois partie que l'on retrouve dans des fichiers distincts

- **Modèle** : Cette partie gère les données réelles de l'application. Son rôle est d'aller récupérer les informations brute, en l'occurrence via l'api localstorage du navigateur et de les envoyer au contrôleur. La classe qui représente le modèle est Model dépendante de Store.

- **Vue** : Cette partie se concentre sur l'affichage. La vue de l'application est construite autour de deux classes, Template et View.

  La classe Template, en récupérant les données du modèle, permet de créer les éléments html des tâches via un modèle html et les retourne en chaine de caractère à View.

  View est une classe beaucoup plus complète, elle permet "d'ajouter / modifier / supprimer" des tâches du DOM (document object model) et également d'écouter des événements de la page ex : ajout d'un élément en appuyant sur entrée / édition d'une tâche avec dblclick etc. Ces événements sont ensuite reliés au contrôleur afin qu'il puisse appliquer les traitements nécessaires à la demande de l'utilisateur.

- **Contrôleur** : Cette partie gère la logique du code. C'est l'intermédiaire entre le modèle et la vue. Le contrôleur va demander au modèle les données, les analyser, et renvoyer ces données à la vue pour les afficher. Le contrôleur est représenté par la classe Controller.
  Le contrôleur implémente également un routeur permettant d'afficher une vue correspondant au suffix d'url choisi par l'utilisateur via l'élément .filter.

Tester l'application avec Jasmine

Jasmine est une bibliothèque js permettant de "tester" le comportement des applications via des suite "describe", des spécifications "it" et des expectations "expect". Voir doc jasmine : https://jasmine.github.io/tutorials/your_first_suite

Pour cette application, nous testons seulement la classe Controller. Pour ce faire, nous créons des classes fictives du modèle et de la vue avec les mêmes méthodes que les classes originales.

Via la bibliothèque Jasmine nous observons chaque appelle au methode de ces classes afin de vérifier que le contrôleur les appellent avec les bon paramètres envoyés.
selon l'action à effectuer

Descriptif des Class de l'application

helpers.js

Ce fichier définit des méthodes à l'objet global window. Ces méthodes sont des raccourcis ou des nouvelles fonctionnalités du DOM (Document Object Model).

window.qs(selector, [scop]) : Element
Raccourcis de la méthode querySelector

---

window.qsa(selector, [scop]) : NodeList
Raccourcis de la méthode querySelectorAll

---

window.$on(target, type, callback, useCapture)
Créer un écouteur d'événement à l'aide de la méthode addEventListener sur l'élément choisi.

target : La cible de l'événement
type : Le type d'événement
callback : Fonction appelée quand l'événement spécifié est envoyé à la cible
useCapture (facultatif) : Un boolean pour choisir entre la phase de bouillonnement ou de capture d'événements.

---

window.$delegate(target, selector, type, handler)
Créer des écouteurs d'événement sur des éléments existants ou futurs depuis un élément racine.

target : l'élément racine cible
selector : Un sélecteur css en chaîne de caractère représentant les futurs éléments enfant à pointer comme cible également.
type : Le type d'événement
handler : Fonction appelée quand l'événement spécifié est envoyé à la cible

---

window.$parent(element, tagName)
Recherche un noeud parent (par son nom de balise) d'un élément

element : L'élément enfant
tagName : Le nom de la balise de l'élément parent recherché

NodeList::forEach
Ajoute la méthode forEach de la class Array au prototype de NodeList

store.js

Le fichier store.js contient le code de la class Store
La class Store gére l'API Web Storage et permet de : créer, d'ajouter, rechercher, et supprimer des données stockées dans un objet localStorage.

Store::Store(name, [callback]) : Store
Création de la base de données de l'application. Un nouvel objet sera stocké dans le LocalStorage du navigateur.

name : Nom de la base de donnée
callback (facultatif) : Fonction de rappel appelée après la création de l'objet dans le contexte de l'instance de Store et reçoit en paramètre l'objet js de la bdd.

Store::find(query, [callback])
Récupère une donnée dans localStorage à partir d'une requête sous forme d'objet js
Utilisation : db.find({id: deux}, function(item) { /*Retourne l'item avec l'id = deux*/ });

query : L'objet js à comparer avec l'objet js enregistrer dans localStorage
callback (facultatif) : Fonction de rappel appelée après la comparaison recevant en paramètre l'objet js correspondant à la requête dans le contexte de Store.

Store::findAll([callback])
Récupère toutes les données de la base de données

callback (facultatif) : Fonction de rappel recevant en paramètre l'objet contenant toutes les données de la bdd dans le contexte de l'instance de Store.

Store::save(updateData, [callback, id])
Enregistre ou modifie une entrée dans la bdd.

updateData : L'objet js à ajouter ou modifier dans la bdd
callback (facultatif) : Fonction de rappel recevant en paramètre l'objet ajouter ou toutes les données de la bdd s' il s'agit d'une modification. La fonction est exécutée dans le contexte de Store.
id (facultatif) : L'id de l'entrée à modifier

Store::remove(id, callback)
Supprimer un item dans l'objet js représentant la bdd en fonction de l'id passé paramètre

id : L'id unique identifiant l'item
callback : Fonction de rappel recevant en paramètre l'objet js représentant toute la bdd. La fonction est appelée juste après la suppression de l'item dans le contexte de l'instance de Store.

Store::drop(callback)
Vide l'objet de localStorage

callback : Fonction de rappel recevant en paramètre l'objet js représentant la bdd. La fonction est appelée juste après la suppression dans le contexte de l'instance de Store.

model.js

Le fichier model.js contient le code de la class Model
La class Model gère les “tâches” en bdd. Elle permet d’ajouter des tâches, de les modifier ainsi que de les supprimer. La class Model dépend donc de la class Store qui gére l’API Web Storage

Model::Model(storage) : Model
Constructeur de la class Model, il définit une propriété contenant une instance de Store

storage : Une instance de Store

Model::create([title, callback])
Créer une nouvelle tâche et l'ajoute à l’objet localStorage.

title (facultatif) : Le titre de la tâche à créer
callback (facultatif) : Fonction de rappel recevant en paramètre l’objet créé. La fonction est exécutée dans le contexte de Store.

Model::read(query, [callback])
Recherche des tâches dans l’objet localStorage via une requête.

query : Si fonction de rappel voir Store::findAll, si c’est un chiffre, il s’agit de la *sélection par id unique*, si c’est un objet de comparaison voir Store::find
callback (facultatif) : Fonction de rappel recevant en paramètre les resultat de la recherche

---

Model::update(id, data, callback)
Modifie les données d’une tâche dans l’objet localStorage

id : L’id unique de la tâche à modifier
data : Les propriétés à mettre à jour et leur nouvelle valeur
callback : Fonction de rappel recevant en paramètre toutes les données de la bdd. La fonction est exécutée dans le contexte de Store.

Model::remove(id, callback)
Supprime une tâche de l’objet localStorage

id : L’id unique de la tâche à supprimer
callback : Fonction de rappel recevant en paramètre l’objet js représentant toute la bdd. La fonction est appelée juste après la suppression de l’item dans le contexte de l’instance de Store.

Model::removeAll(callback)
Vide l’objet de localStorage

callback : Fonction de rappel recevant en paramètre l’objet js représentant toute la bdd. La fonction est appelée juste après la suppression dans le contexte de l’instance de Store.

Model::getCount(callback)
Compte le nombre de tâches présente dans l’objet localStorage

callback : Fonction de rappel recevant en paramètre un objet js contenant le nombre de tâche par catégorie.

template.js

Le fichier template.js contient le code de la class Template ainsi qu'une fonction d'échappement de caractère : ascape(string)
La class Template permet de créer le modèle html des tâches tel un moteur de template simplifié.

Template::Template() : Template
Constructeur de la class Template, il définit la propriété privée defaultTemplate qui contient le modèle html d'une tâche

---

Template::show(data) : String
Construit la liste des tâches suivant le modèle html (defaultTemplate) et la retourne dans une chaine de caractère.

data : Objet js contenant toutes la liste des tâches à construire en html

---

Template::itemCounter(activeTodos) : String
Retourne une chaîne de caractère "html" indiquant combien de tâche sont "actives"

activeTodos : Le nombre de tâche "active"

Template::clearCompleteButton(completedTodos) : String
Retourne la chaîne de caractère "Clear completed" si au moins une tâche est complétée sinon retourne une chaîne vide.

completedTodos : Le nombre de tâche "complétée"

view.js

Le fichier view.js contient le code de la class View
La class View gère la vue de l'application, c-a-d toute la partie html. Elle permet d'ajouter supprimer les tâches du DOM (documentObjectModel)

View::View(template) : View
Constructeur de la class View, initialise la vue de l'application en sélectionnant tous les éléments html de base de l'application.

template : Une instance de la class Template

View::_removeItem(id)
Supprimer une tâche du DOM en la sélectionnant par son identifiant unique.

id : Un chiffre qui représente l'identifiant unique d'une tâche.

View::_clearCompleteButton(completedCount, visible)
Rend l'élément .clear-completed visible ou non sur la page.

completedCount : Le nombre de tâche "complétée"
visible : Un booléen indiquant si le bouton doit être visible ou pas

---

View::_setFilter(currentPage)
Ajoute la class .selected à l'un des liens de la liste à puce .filters qui correspond au suffix de l'url actuelle.

currentPage : Le suffix de l'url actuelle

View::_elementComplete(id, completed)
Ajouter la class .completed ou pas à une tâche sélectionnée par son id unique

id : Un chiffre qui représente l'identifiant unique d'une tâche pour la sélectionner
completed : Un booléen indiquant si la tâche est complétée ou pas.

View::_editItem(id, title)
Rend la tâche éditable en lui ajoutant un champ de texte avec la class .edit qui à pour valeur le titre actuel de la tâche et ajoute la class .editing à la tâche.

id : Un chiffre qui représente l'identifiant unique d'une tâche pour la sélectionner
title : Le titre de la tâche

View::_editItemDone(id, title)
Supprime le champ de texte ajouté pour l'édition de la tâche. Rend la tâche à son état initial en supprimant la class .editing de l'élément

id : Un chiffre qui représente l'identifiant unique d'une tâche pour la sélectionner
title : Le titre de la tâche

Suite view.js

View::render(viewCmd, parameter)
Méthode regroupant 11 actions possibles sur la vue par le biais des méthodes internes commençant par “_” ainsi que les méthodes de la class Template.

showEntries : Construit toutes les tâches et les ajoute au DOM
removeItem : Voire View::_removeItem(id)
updateElementCount : Met à jour le compteur de nombre de tâche active
clearCompletedButton : Rend l'élément .clear-completed visible ou non.
contentBlockVisibility : Rend l'élément .main et .footer visible ou non
toggleAll : Coche ou non la case .toogle-all
setFilter : Voire View::_setFilter(currentPage)
clearNewTodo : Vide le champ de texte .new-todo
elementComplete : Voire View::__elementComplete(id, completed)
editItem : Voire View::_editItem(id, title)
edtiItemDone :Voire View::_editItemDone(id, title)

viewCmd : Une chaine de caractère correspondant à une action à effectuer sur la vue
parameter : Un objet js contenant les données venant du modèle à fournir à la vue selon l’action
---
View::_itemId(element) Number
Retourne l’id d’une tâche présent en “attribut personnalisé” (data-set) sur l'élément racine de la tâche

element : Un Element DOM enfant de la balise li racine d’une tâche.

View::_bindItemEditDone(handler)
Execute handler sur les événements keypress (appuyer sur entrée) et blur (suppression du focus) sur la cible du champ de texte présent lors de l'édition d’une tâche.

handler : Fonction de rappel qui reçoit en paramètre un objet js contenant id de la tâche modifiée ainsi que title correspondant à la valeur du champ de text
---
View::_bindItemEditCancel(handler)
Execute handler sur l'événement keyup (relâcher la touche échappe) pour annuler l'édition d’une tâche sur la cible du champ de texte présent lors de l'édition d’une tâche.

handler : Fonction de rappel qui reçoit en paramètre l’id de la tâche qui été en cours d'édition
---

Suite view.js

View::bind(event, handler)
Méthode gérant un groupe de huit gestionnaires d'événements prêt définit possible d’utiliser.

newTodo : Execute handler sur change pour le champ de text .new-todo
removeCompleted : Execute handler sur click pour le bouton .clear-completed
toggleAll : Execute handler sur click pour le bouton .toogle-all
itemEdit : Execute handler sur dbclick pour les éléments label de chaque tâches
itemRemove : Execute handler sur click pour les boutons .destroy de chaque tâches
itemToggle : Execute handler sur click pour les boutons .toggle de chaque tâches
itemEditDone : Voire View::_bindItemEditDone(handler)
itemEditCancel :Voire View::_bindItemEditCancel(handler)

event : Une chaine de caractère correspondant à l'événement que l’on souhaite utiliser
handler : La fonction de rappel appelé quand l'événement sera déclenché

controller.js

Le fichier controller contient le code de la class Controller
La class Controller gère la logique du code qui prend des *décisions*. C'est l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, prendre des décisions et demander de modifier le texte à afficher à la vue.

Controller::Controller(model, view) : Controller

model : Instance de la class Model
view : Instance de la class View

Controller::setView(locationHash)
Exécute toute les méthodes privée commençant par _ de la class Controller

locationHash : Une chaine de caractère qui représente l'url de la page actuel

---

Controller::showAll()
Récupère tous les objets dans le Model et les envoient à la vue qui va afficher toutes les tâches

---

Controller::showActive()
Récupère tous les objets qui ne sont pas "completed" dans le Model et les envoient à la vue qui va les afficher

---

Controller::showCompleted()
L'inverse de Controller::showActive()

Controller::addItem(title)
Ajoute une tâche au Model et demande à la vue de l'afficher

title : Le titre de la nouvelle tâche

Controller::editItem(id)
Récupère l'objet à modifier en fonction de son id dans le Model et demande à la vue de rendre la tâche editable

id : L'id de la tâche à modifier

Controller::editItemSave(id, title)
Modifie l'item dans le Model et demande à la vue de ne plus le rendre éditable et afficher le nouveau titre. Si le titre envoyé en paramètre est vide, cela supprimer l'élément du Model

id : L'id de la tâche à modifier
title : Le titre de la tâche

Controller::editItemCancel(id)
Récupérer l'item en cours d'édition par son id et demande à la vue de ne plus le rendre éditable et d' afficher l'ancien titre.

id : L'id de la tâche en cours d'édition

Suite controller.js

Controller::removeItem(id)
Supprime l'item du Model et demande à la vue de supprimer l'item de la page

id : L'id de l'item à supprimer

Controller::removeCompletedItems()
Récupère tous les items qui sont "completed" pour les supprimer par le biais
de la méthode Controller::removeItem(id)

Controller::toggleComplete(id, completed, silent)
Modifie dans le Model l'état complétez ou pas d'un item et demande à la vue de modifier l'élément en fonction de son état.

id : L'id de l'élément sélectionné
completed : Un booléen représentant l'état complétez ou pas.
silent :

Controller::toggleAll(completed)
Va basculer l'état complétez ou pas de tous les items

completed : Un booléen représentant l'état complétez ou pas.

Méthode privée

Controller::_updateCount()
Met à jour les éléments de la page qui changent en fonction du nombre de tâches

---

Controller::_filter(force)
Cette méthode est un "router' elle est chargé d'exécuter les méthodes commençant par show du controller en fonction de l'url de la page

force : Un booléen permettant de forcer l'appelle à l'une des méthode show du controller

---

Controller::_updateFilterState(currentPage)
Execute Controller::_filter() en lui envoyant la route active

currentPage : L'url de la page actuel

app.js
app est le fichier amorce "bootstrap" de l'application, il initialise l'application

Todo::Todo(name) : Todo
Initialise toutes les class de l'application

name : Une chaîne de caractère contenant le nom de l'objet qui va représenter la base de donnée

---