Bienvenue dans Reprenez-un-projet-existant

Ce projet est une application de liste de tâches (To-Do List) développée en utilisant les technologies HTML, CSS, et JavaScript. Elle permet aux utilisateurs de créer, marquer comme terminées, et supprimer des tâches.

Pour Commencer

Pour utiliser ce projet, clonez le dépôt depuis GitHub ou téléchargez l'archive du projet sur votre système local.

Introduction

Ce projet est une application de liste de tâches (To-Do List) développée en utilisant les technologies HTML, CSS, et JavaScript. Elle permet aux utilisateurs de créer, marquer comme terminées, et supprimer des tâches. Cet outil vise à simplifier la planification personnelle ou professionnelle, rendant la gestion des tâches à la fois visuelle et interactive.

Pour Commencer

Pour utiliser ce projet, clonez le dépôt depuis GitHub ou téléchargez l'archive du projet sur votre système local.

Prérequis

Pour exécuter et travailler sur ce site web de liste de tâches localement, vous devez disposer des outils et environnements suivants installés sur votre machine :

Navigateur Web Moderne :

Google Chrome : Version 88 ou ultérieure recommandée pour les meilleures performances et la compatibilité avec les fonctionnalités ES6+.

Mozilla Firefox : Version 85 ou ultérieure.

Safari : Version 14 ou ultérieure pour les utilisateurs de macOS.

Microsoft Edge : Version 88 ou ultérieure.

Le site utilise des fonctionnalités JavaScript et CSS modernes qui sont mieux prises en charge par les versions récentes des navigateurs. Cela assure que l'expérience utilisateur et les fonctionnalités du site fonctionnent comme prévu.

Si vous souhaitez utiliser l'application à la manière d'un utilisateur final et si vous ne souhaitez pas exécuter les tests ou travailler le développement de l'application, node.js n'est pas nécessaire.

Pour travailler et développer ce site web de liste de tâches localement, vous devez disposer des outils et environnements suivants installés sur votre machine :

Node.js (Optionnel) :

Version 14.x ou ultérieure. Node.js est nécessaire pour gérer les dépendances du projet, exécuter les scripts de développement et de build, et pour tout processus de développement côté serveur si nécessaire. Téléchargez et installez Node.js depuis nodejs.org. Si vous souhaitez utiliser l'application à la manière d'un utilisateur final et si vous ne souhaitez pas exécuter les tests ou travailler le développement de l'application, node.js n'est pas nécessaire.

npm (Node Package Manager - Optionnel) :

Généralement installé avec Node.js. npm est utilisé pour gérer les librairies et les paquets tiers nécessaires au développement du site. Assurez-vous que npm est à jour en exécutant npm install npm@latest -g dans votre terminal.
Si vous souhaitez utiliser l'application à la manière d'un utilisateur final et si vous ne souhaitez pas exécuter les tests ou travailler le développement de l'application, node.js n'est pas nécessaire.

Éditeur de Code (Optionnel) :

Un éditeur de code tel que Visual Studio Code, Sublime Text, Atom, ou tout autre de votre choix pour éditer et gérer les fichiers du projet.

Outils de Développement (Optionnel) :

Git : Pour le contrôle de version et la gestion du code source. Téléchargez Git depuis git-scm.com. Extensions de Navigateur pour Développeurs : Extensions comme React Developer Tools pour Chrome/Firefox, Vue.js devtools, etc., pour faciliter le débogage et le développement.

Guide d'installation

Pour accéder à la page du site, il suffit d'ouvrir le fichier index.html dans un navigateur web. Cette action affiche l'interface utilisateur de l'application de liste de tâches, permettant immédiatement l'interaction.

Pour configurer l'environnement de développement et installer les dépendances nécessaires, assurez-vous d'avoir Node.js et npm installés. Ouvrez un terminal, naviguez jusqu'au dossier racine du projet, et exécutez npm install. Cette commande installe toutes les dépendances listées dans le fichier package.json, préparant ainsi votre environnement pour le développement.

Comment Utiliser

Pour utiliser l'application de liste de tâches, ouvrez index.html dans un navigateur.

- Ajoutez des tâches via le champ de saisie et validez avec la touche Entrée ou la flèche vers le bas à gauche de champ de saisie.
- Marquez-les tâches comme terminées en cliquant sur la case à cocher.
- Modifiez les tâches par double-clic sur une tâche.
- Filtrez les tâches en cliquant sur les liens en bas :

1. All : voir toutes les tâches
2. Complete : voir les tâches terminées
3. Active : voir les tâches en cours ou à effectuer

- Supprimez en cliquant sur le bouton "x" apparaissant au survol des tâches.

Fonctionnalités

Ajout de tâches : Les utilisateurs peuvent entrer le titre d'une nouvelle tâche dans un champ de saisie et l'ajouter à la liste en appuyant sur la touche Entrée ou en cliquant sur un bouton dédié. La nouvelle tâche apparaît alors en bas de la liste des tâches à faire.

Marquage des tâches comme terminées : À côté de chaque tâche se trouve une case à cocher. Lorsqu'un utilisateur coche cette case, la tâche est visuellement barrée,

indiquant qu'elle a été accomplie. Cette action ne supprime pas la tâche mais change son état à "terminé".

Filtrage des tâches : Des liens ou boutons permettent aux utilisateurs de filtrer les tâches affichées : toutes les tâches, uniquement les tâches actives (non terminées), ou uniquement les tâches terminées. Cette fonctionnalité modifie dynamiquement l'affichage des tâches sans recharger la page.

Suppression de tâches: Chaque tâche dispose d'un bouton de suppression. Lorsqu'un utilisateur clique sur ce bouton, la tâche est immédiatement retirée de la liste. Un bouton "Effacer les tâches terminées" permet également de supprimer en masse toutes les tâches marquées comme terminées.

Structure du Répertoire

Racine du Projet

index.html :

Le fichier HTML principal qui sert de squelette à l'application Todo. Il inclut les liens vers les scripts JavaScript et les feuilles de style CSS nécessaires.

package.json :

Un fichier de configuration pour Node.js qui spécifie les dépendances du projet et peut contenir d'autres métadonnées relatives au projet.

package-lock.json :

Verrouille les versions des dépendances du projet pour assurer la cohérence des installations à travers différents environnements.

Dossier CSS

Contient les feuilles de style qui définissent l'apparence de l'application.

base.css :

Objectif : Ce fichier contient les styles de base qui sont appliqués globalement à toute l'application. Il s'agit notamment de la mise en forme des éléments HTML par défaut, comme les polices de caractères, les couleurs de fond, les marges, et les paddings.

Rôle : Son rôle est d'assurer une cohérence visuelle à travers l'application et de définir un ensemble de règles de style fondamentales sur lesquelles les autres styles peuvent se construire. Il peut inclure des styles pour des éléments couramment utilisés comme les boutons, les en-têtes, et les formulaires.

index.css :

Objectif : index.css contient les styles spécifiques à l'interface utilisateur de l'application Todo. Ces styles sont plus détaillés et ciblent des éléments ou des composants particuliers de l'application, tels que la liste des tâches, les champs de saisie, et les boutons d'action.

Rôle : Il est destiné à personnaliser l'apparence des différentes parties de l'application pour correspondre à l'identité visuelle désirée et améliorer l'expérience utilisateur. Les règles définies dans index.css peuvent surcharger ou compléter les styles de base définis dans base.css.

Dossier JS

Regroupe tous les scripts JavaScript qui mettent en œuvre la logique de l'application.

app.js :

Fonction : Sert de point d'entrée pour l'application. Ce script initialise l'application en instanciant les objets Modèle, Vue et Contrôleur.

Processus : Il lie les composants de l'application pour qu'ils puissent communiquer efficacement, orchestrant ainsi le flux de données et d'actions de l'utilisateur à travers l'application.

controller.js :

Fonction : Agit comme le cerveau de l'application, recevant les entrées de l'utilisateur via la Vue, manipulant les données à travers le Modèle, et mettant à jour la Vue en conséquence.

Processus : Gère les commandes de l'utilisateur, telles que l'ajout, la suppression, et la mise à jour des tâches Todo, ainsi que le filtrage des tâches affichées selon leur état (toutes, actives, complétées).

model.js :

Fonction : Représente le modèle de données de l'application, où la logique métier et la gestion des données sont implémentées.

Processus : Inclut des méthodes pour créer, lire, mettre à jour, et supprimer des tâches, ainsi que pour persister ces tâches entre les sessions à l'aide du localStorage.

store.js :

Fonction : Fournit une couche d'abstraction pour accéder et modifier les données stockées dans le localStorage.

Processus : Encapsule la logique de persistance des données, permettant au modèle d'enregistrer et de récupérer les tâches sans se soucier des détails de mise en œuvre du stockage.

template.js :

Fonction : Gère les templates HTML pour l'affichage dynamique des éléments de l'interface utilisateur, comme les tâches Todo.

Processus: Utilise des chaînes de caractères HTML avec des placeholders pour les données, qui sont remplacés par les valeurs actuelles lors de la génération du contenu dynamique.

view.js :

Fonction : Responsable de l'affichage de l'application, mettant à jour l'interface utilisateur en réponse aux changements dans le modèle ou aux interactions de l'utilisateur.

Processus : Écoute les événements de l'utilisateur, informe le Contrôleur des actions de l'utilisateur, et met à jour l'affichage lorsque le modèle change.

helpers.js :

Fonction : Contient des fonctions utilitaires pour effectuer des tâches communes à travers l'application, comme la manipulation du DOM ou le formatage des données.

Processus: Fournit des méthodes réutilisables qui simplifient certaines opérations et réduisent la duplication du code dans l'application.

base.js :

Fonction: Inclut des fonctions et des configurations de base utilisées à travers l'application, telles que les paramètres de configuration du template et les helpers de manipulation du DOM.

Processus : Peut inclure la configuration de bibliothèques tierces, l'initialisation de composants communs ou des polyfills pour assurer la compatibilité entre navigateurs.

Dossier de Tests

Contient les fichiers relatifs aux tests unitaires ou d'intégration de l'application.

ControllerSpec.js :

Spécifie les tests pour le contrôleur de l'application Todo.

SpecRunner.html :

Un fichier HTML pour exécuter les tests définis dans ControllerSpec.js dans un navigateur.

Spécification technique de l'application

Structure HTML

La structure HTML de l'application de liste de tâches est définie dans le fichier index.html. Elle est conçue pour être simple et intuitive, facilitant l'interaction de l'utilisateur avec l'application.

DOCTYPE Declaration : La déclaration <!doctype html> spécifie le type de document et la version d'HTML utilisés.

Élément racine HTML : L'élément < html > est l'élément racine du document.

Métadonnées du document : L'élément < head > contient les métadonnées du document, telles que le jeu de caractères, le titre de la page et les liens vers les feuilles de style.

Contenu visible : L'élément < body > contient le contenu visible du document.

Section principale : L'élément < section class="todoapp" > est le conteneur principal de l'application de liste de tâches.

En-tête : L'élément < header class="header" > contient le titre de la page et le champ d'entrée pour ajouter de nouvelles tâches.

Champ d'entrée pour les nouvelles tâches : L'élément < input class="new-todo" placeholder="What needs to be done?" id="new-todo" name="new-todo" autofocus > est le champ d'entrée pour ajouter de nouvelles tâches.

Section principale : L'élément < section class="main" > contient la liste des tâches.

Liste des tâches : L'élément < ul class="todo-list"> est l'endroit où la liste des tâches sera affichée.

Pied de page : L'élément < footer class="footer" > contient les filtres et le bouton "effacer terminé".

Nombre de tâches restantes : L'élément < span class="todo-count"> affiche le nombre de tâches restantes.

Filtres : L'élément < ul class="filters" > contient les filtres pour afficher les tâches actives et terminées.

Bouton "Effacer terminé" : L'élément < button class="clear-completed" > Clear completed < /button > est utilisé pour effacer les tâches terminées.

Informations sur les créateurs : L'élément < footer class="info" > contient des informations sur les créateurs de l'application.

Éléments de script : Les éléments < script > sont utilisés pour inclure les fichiers JavaScript nécessaires à l'application.

Fonctionnalités JavaScript principales

Modèle (js/model.js) :

Le fichier model.js définit la logique métier de l'application de liste de tâches. Il crée une instance de la classe Model qui interagit avec la couche de stockage pour créer, lire, mettre à jour et supprimer des tâches. Il fournit également une méthode pour compter le nombre total de tâches actives et complétées.

Model(storage) : constructeur qui crée une nouvelle instance de la classe Model et prend en paramètre une référence à la classe de stockage.

create(title, callback) : crée une nouvelle tâche avec le titre fourni et appelle la fonction de rappel une fois la tâche enregistrée dans le stockage.

read(query, callback) : lit les tâches du stockage en fonction de la requête fournie. La requête peut être une chaîne de caractères, un nombre ou un objet. Si la requête est une chaîne de caractères ou un nombre, elle est considérée comme l'ID de la tâche. Si la requête est un objet, elle est considérée comme un filtre pour lire les tâches correspondantes.

update(id, data, callback) : met à jour les propriétés d'une tâche existante en fonction de l'ID fourni et des nouvelles données. La fonction de rappel est appelée une fois la mise à jour effectuée.

remove(id, callback) : supprime une tâche existante en fonction de son ID et appelle la fonction de rappel une fois la suppression effectuée.

removeAll(callback) : supprime toutes les tâches du stockage et appelle la fonction de rappel une fois la suppression effectuée.

getCount(callback) : renvoie le nombre total de tâches actives et complétées en appelant la fonction de rappel une fois le décompte effectué.

Chaque méthode de la classe Model prend en paramètre une fonction de rappel (callback) qui est appelée une fois l'opération effectuée. Cette fonction de rappel prend en paramètre les données récupérées ou mises à jour, ou un message d'erreur en cas d'échec de l'opération.

Vue (js/view.js) :

Génère la représentation HTML des tâches à partir des données du modèle et gère les éléments de l'interface utilisateur comme les boutons et les champs de saisie. Écoute et réagit aux actions de l'utilisateur (clics, entrée de texte, etc.) en émettant des événements appropriés. Met à jour l'affichage lorsque le modèle change, reflétant immédiatement les ajouts, suppressions et autres modifications des tâches. La classe View prend en paramètre un objet template qui contient des fonctions pour générer du HTML à partir de modèles de données. Elle définit également plusieurs propriétés pour les éléments DOM couramment utilisés dans l'application.

La classe View possède deux méthodes importantes : **bind** et **render**. La méthode **bind** lie des événements aux éléments DOM correspondants et enregistre les gestionnaires de ces événements. La méthode **render** met à jour l'interface utilisateur en fonction des données reçues.

View(template) : constructeur de la classe View. Il prend en paramètre un objet template qui contient des fonctions pour générer du HTML à partir de modèles de données. Il définit également plusieurs propriétés pour les éléments DOM couramment utilisés dans l'application.

_removeItem(id) : supprime l'élément DOM correspondant à la tâche avec l'ID donné.

_clearCompletedButton(completedCount, visible) : met à jour le bouton "Clear completed" en fonction du nombre de tâches complétées et de sa visibilité.

_setFilter(currentPage) : définit le filtre actif en fonction de la page courante.

_elementComplete(id, completed): met à jour l'état "completed" d'une tâche en fonction de l'ID donné et de la valeur completed.

_editItem(id, title) : passe une tâche en mode édition et crée un champ de saisie pour le titre de l'élément.

_editItemDone(id, title) : met à jour le titre d'un élément et le passe hors mode édition.

render(viewCmd, parameter) : met à jour l'interface utilisateur en fonction des données reçues. Cette méthode prend en paramètre une chaîne de caractères viewCmd qui représente la commande à exécuter, et un objet parameter qui contient les données nécessaires à cette commande.

_itemId(element) : renvoie l'identifiant de l'élément DOM parent le plus proche correspondant à une tâche.

_bindItemEditDone(handler) : lie un événement "blur" au champ de saisie d'un élément en mode édition pour mettre à jour le titre de l'élément.

_bindItemEditCancel(handler) : lie un événement "keyup" au champ de saisie d'un élément en mode édition pour annuler les modifications apportées au titre de l'élément.

bind(event, handler): lie des événements aux éléments DOM correspondants et enregistre les gestionnaires de ces événements. Cette méthode prend en paramètre une chaîne de caractères event qui représente l'événement à traiter, et une fonction handler qui sera appelée lorsqu'un événement se produira.

Enfin, la classe View est exportée vers l'objet global window.app.

Contrôleur (js/controller.js) :

Initialise l'application et relie le modèle et la vue en écoutant les événements émis par la vue. Réagit aux interactions de l'utilisateur en mettant à jour le modèle (par exemple, ajoutant une nouvelle tâche) ou en modifiant la vue (par exemple, filtrant les tâches affichées). Gère la logique de filtrage des tâches et le basculement entre les tâches actives et terminées.

La fonction Controller est définie avec deux paramètres : model et view. Elle crée une instance de la classe Controller qui encapsule les deux instances model et view, et définit plusieurs gestionnaires d'événements pour interagir avec la vue.

Controller(model, view) : constructeur de la classe Controller. Il prend en paramètre deux instances, model et view, et les stocke en tant que propriétés de l'objet Controller.

setView(locationHash) : méthode appelée lors du chargement de la page ou lorsqu'un utilisateur change de filtre. Elle met à jour l'état du filtre en fonction de l'URL actuelle et déclenche un filtrage des tâches en conséquence.

showAll() : méthode appelée pour afficher toutes les tâches. Elle lit toutes les tâches à partir du modèle et les affiche dans la vue.

showActive() : méthode appelée pour afficher uniquement les tâches actives. Elle lit toutes les tâches à partir du modèle, filtre les tâches actives et les affiche dans la vue.

showCompleted() : méthode appelée pour afficher uniquement les tâches complétées. Elle lit toutes les tâches à partir du modèle, filtre les tâches complétées et les affiche dans la vue.

addItem(title) : méthode appelée lorsqu'un utilisateur ajoute une nouvelle tâche. Elle vérifie si le titre de la tâche est vide, crée une nouvelle tâche à partir du modèle et affiche la tâche dans la vue.

editItem(id) : méthode appelée lorsqu'un utilisateur clique sur une tâche pour la modifier. Elle lit la tâche à partir du modèle, affiche le formulaire de modification dans la vue et met à jour la tâche en fonction des modifications apportées par l'utilisateur.

editItemSave(id, title) : méthode appelée lorsqu'un utilisateur enregistre une tâche modifiée. Elle met à jour la tâche dans le modèle et affiche la tâche mise à jour dans la vue.

editItemCancel(id) : méthode appelée lorsqu'un utilisateur annule une modification de tâche. Elle lit la tâche à partir du modèle et affiche la tâche dans la vue.

removeItem(id) : méthode appelée pour supprimer une tâche. Elle supprime la tâche du modèle et de la vue.

removeCompletedItems() : méthode appelée pour supprimer toutes les tâches complétées. Elle lit toutes les tâches à partir du modèle, supprime les tâches complétées et met à jour l'affichage de la vue.

toggleComplete(id, completed, silent) : méthode appelée lorsqu'un utilisateur coche ou décoche une tâche. Elle met à jour la tâche dans le modèle et affiche la tâche mise à jour dans la vue.

toggleAll(completed) : méthode appelée pour cocher ou décocher toutes les tâches. Elle lit toutes les tâches à partir du modèle, met à jour les tâches en fonction de l'état de la case à cocher "Tout cocher" et affiche les tâches mises à jour dans la vue.

_updateCount() : méthode appelée pour mettre à jour le nombre de tâches actives et complétées affichées dans la vue.

_filter(force) : méthode appelée pour filtrer les tâches en fonction de l'état actuel du filtre. Elle met à jour l'affichage de la vue en fonction des tâches filtrées.

_updateFilterState(currentPage) : méthode appelée pour mettre à jour l'état du filtre en fonction de l'URL actuelle. Elle met à jour l'affichage de la vue pour refléter l'état actuel du filtre.

Enfin, la classe Controller est exportée vers l'objet global window.app.

Autres fonctionnalités

App.js

Le fichier app.js est le point d'entrée principal de l'application de liste de tâches. Il crée une instance de la classe Todo en utilisant le nom de la liste de tâches, puis configure les gestionnaires d'événements pour mettre à jour la vue en fonction de l'URL actuelle.

Todo(name) : constructeur de la classe Todo. Il prend en paramètre le nom de la liste de tâches et crée une nouvelle instance des classes Model, View, Controller, et Template, en utilisant le nom de la liste de tâches pour initialiser le stockage.

setView() : fonction qui est appelée lorsque la page est chargée ou lorsque l'URL est modifiée. Elle met à jour la vue en appelant la méthode setView() du contrôleur en utilisant l'URL actuelle.

todo: instance de la classe Todo qui est créée en utilisant le nom de la liste de tâches.

$on(window, 'load', setView) : fonction qui est utilisée pour attacher un gestionnaire d'événements à la fenêtre lorsque la page est chargée. Elle appelle la fonction setView() pour mettre à jour la vue en fonction de l'URL actuelle.

$on(window, 'hashchange', setView) : fonction qui est utilisée pour attacher un gestionnaire d'événements à la fenêtre lorsque l'URL est modifiée. Elle appelle la fonction setView() pour mettre à jour la vue en fonction de l'URL actuelle.

Template.js

Le fichier template.js contient la définition de la classe Template, qui est responsable de générer du HTML à partir de données fournies en entrée. Cette classe est utilisée dans l'application pour créer des éléments HTML pour les tâches à afficher dans la liste des tâches.

escapeHtmlChar(chr) : Cette fonction prend un caractère en entrée et renvoie le caractère correspondant échappé en HTML. Par exemple, si l'entrée est <, la fonction renverra <. Cette fonction est utilisée pour échapper les caractères spéciaux dans une chaîne de caractères afin d'éviter les problèmes de sécurité et de rendu HTML.

reUnescapedHtml : Cette expression régulière correspond aux caractères qui doivent être échappés en HTML. Elle correspond aux caractères &, <, >, ", ', et .

reHasUnescapedHtml : Cette expression régulière correspond aux chaînes qui contiennent des caractères non échappés. Elle est utilisée pour vérifier si une chaîne de caractères contient des caractères spéciaux qui doivent être échappés avant d'être insérés dans du HTML.

escape(string) : Cette fonction prend une chaîne de caractères en entrée et renvoie la chaîne échappée en utilisant la fonction escapeHtmlChar. Cette fonction est utilisée pour échapper les caractères spéciaux dans une chaîne de caractères avant de l'insérer dans du HTML.

Template() : Cette fonction construit un nouvel objet Template. Cette classe est utilisée pour générer du HTML à partir de données.

Template.prototype.defaultTemplate : Cette propriété contient le modèle par défaut pour un élément de liste de tâches. Le modèle est une chaîne de caractères qui contient des placeholders pour les données, qui seront remplacées par les valeurs réelles lors de la génération du HTML.

Template.prototype.show(data) : Cette méthode prend un tableau de données en entrée et génère une chaîne de caractères HTML pour chaque élément du tableau en utilisant le modèle par défaut. Cette méthode utilise la fonction escape pour échapper les caractères spéciaux dans les données avant de les insérer dans le HTML.

Template.prototype.itemCounter(activeTodos) : Cette méthode prend un nombre en entrée et renvoie une chaîne de caractères indiquant le nombre de tâches actives. Cette méthode est utilisée pour mettre à jour le compteur de tâches actives dans l'interface utilisateur.

Template.prototype.clearCompletedButton(completedTodos) : Cette méthode prend un nombre en entrée et renvoie une chaîne de caractères pour le bouton "Clear completed" si le nombre de tâches complétées est supérieur à zéro, ou une chaîne vide sinon. Cette méthode est utilisée pour afficher ou masquer le bouton "Clear completed" en fonction du nombre de tâches complétées.

Store.js

Le fichier store.js est responsable de la gestion de la persistance des données de l'application Todo. Il utilise le stockage local du navigateur pour stocker et récupérer les données des tâches. Il définit une classe nommée Store qui offre des méthodes pour trouver, enregistrer, mettre à jour et supprimer les données des tâches.

Le fichier store.js définit une classe nommée Store qui est utilisée pour gérer la persistance des données de l'application Todo.

La classe Store est initialisée avec un nom qui est utilisé pour stocker les données des tâches dans le stockage local du navigateur. Si aucune collection n'existe déjà pour ce nom, une nouvelle collection est créée avec un tableau vide de tâches.

La classe Store offre les méthodes suivantes :

find(query, callback) : Cette méthode prend en entrée un objet de requête et un rappel de fonction. Elle recherche les éléments correspondant à la requête dans le

tableau de tâches et renvoie un tableau filtré de tâches qui correspondent à la requête.

findAll(callback) : Cette méthode prend en entrée un rappel de fonction et renvoie le tableau complet de tâches. save(updateData, callback, id) : Cette méthode prend en entrée les données de mise à jour, un rappel de fonction et une ID facultative. Si une ID est fournie, elle met à jour les propriétés de l'élément de tâche existant correspondant à cette ID. Sinon, elle ajoute un nouvel élément de tâche au tableau de tâches avec un ID unique généré à partir de l'heure actuelle.

remove(id, callback) : Cette méthode prend en entrée une ID et un rappel de fonction. Elle supprime l'élément de tâche correspondant à cette ID du tableau de tâches.

drop(callback) : Cette méthode supprime toutes les données du stockage local et crée une nouvelle collection avec un tableau vide de tâches.

Helpers.js

Le fichier helpers.js contient plusieurs fonctions utilitaires pour faciliter la manipulation du DOM et la gestion des événements.

window.qs(selector, scope) : Cette fonction renvoie le premier élément correspondant au sélecteur CSS spécifié, en recherchant dans le DOM à partir de l'objet scope (par défaut, document).

window.qsa(selector, scope) : Cette fonction renvoie une liste d'éléments correspondant au sélecteur CSS spécifié, en recherchant dans le DOM à partir de l'objet scope (par défaut, document).

window.$on(target, type, callback, useCapture) : Cette fonction ajoute un gestionnaire d'événements au niveau de l'élément cible spécifié, pour le type d'événement spécifié. Le gestionnaire d'événements est défini par la fonction callback. L'argument useCapture est facultatif et indique si le gestionnaire doit être déclenché en mode de capture ou en mode de bubbling.

window.$delegate(target, selector, type, handler) : Cette fonction ajoute un gestionnaire d'événements à un élément cible, qui sera déclenché pour tous les éléments correspondant au sélecteur spécifié, maintenant ou dans le futur. La fonction handler sera appelée avec l'événement en paramètre.

window.$parent(element, tagName) : Cette fonction renvoie l'élément parent de l'élément spécifié qui correspond au nom de balise spécifié. Si aucun parent correspondant n'est trouvé, la fonction renvoie undefined.

NodeList.prototype.forEach: Cette fonction étend le prototype NodeList pour ajouter une méthode forEach qui permet de parcourir la liste d'éléments en utilisant une fonction de rappel.

Les tests

ControllerSpec.js

Ce fichier est un ensemble de tests unitaires pour le contrôleur de l'application Todo. Il utilise le framework de test Jasmine pour vérifier le comportement attendu du contrôleur en réponse à différentes actions de l'utilisateur.

Le fichier définit plusieurs fonctions de test, chacune vérifiant un aspect particulier du comportement du contrôleur, pour vérifier que les méthodes du contrôleur sont appelées avec les bons arguments et que les résultats attendus sont renvoyés. Les fonctions de test utilisent des mocks pour simuler le comportement du modèle et de la vue, et vérifient que le contrôleur interagit correctement avec eux.

should show entries on start-up: Cette fonction de test vérifie que lorsque le contrôleur est initialisé, il récupère la liste des tâches à partir du modèle et la rend à l'aide de la vue.

should show all entries without a route: Cette fonction de test vérifie que lorsque l'utilisateur accède à la page d'accueil de l'application, le contrôleur récupère toutes les tâches à partir du modèle et les rend à l'aide de la vue.

should show active entries: Cette fonction de test vérifie que lorsque l'utilisateur clique sur le filtre "active", le contrôleur récupère toutes les tâches actives à partir du modèle et les rend à l'aide de la vue.

should show completed entries: Cette fonction de test vérifie que lorsque l'utilisateur clique sur le filtre "completed", le contrôleur récupère toutes les tâches complétées à partir du modèle et les rend à l'aide de la vue.

should show the content block when todos exists: Cette fonction de test vérifie que lorsque des tâches existent, le contrôleur rend le bloc de contenu à l'aide de la vue.

should hide the content block when no todos exists: Cette fonction de test vérifie que lorsque aucune tâche n'existe, le contrôleur masque le bloc de contenu à l'aide de la vue.

should toggle all todos to completed: Cette fonction de test vérifie que lorsque l'utilisateur clique sur le bouton "toggle all", le contrôleur met à jour toutes les tâches dans le modèle pour les marquer comme complétées.

should add a new todo to the model: Cette fonction de test vérifie que lorsque l'utilisateur ajoute une nouvelle tâche, le contrôleur l'ajoute au modèle.

should add a new todo to the view: Cette fonction de test vérifie que lorsque l'utilisateur ajoute une nouvelle tâche, le contrôleur la rend à l'aide de la vue.

should remove an entry from the model: Cette fonction de test vérifie que lorsque l'utilisateur supprime une tâche, le contrôleur la supprime du modèle.

should remove an entry from the view: Cette fonction de test vérifie que lorsque l'utilisateur supprime une tâche, le contrôleur la supprime de la vue.

should persist the changes on done: Cette fonction de test vérifie que lorsque l'utilisateur modifie une tâche et clique sur le bouton "done", le contrôleur enregistre les modifications dans le modèle.

should leave edit mode on cancel: Cette fonction de test vérifie que lorsque l'utilisateur modifie une tâche et clique sur le bouton "cancel", le contrôleur annule les modifications et laisse la tâche dans l'état éditable.

SpecRunner.html

Ce fichier HTML est utilisé pour exécuter les tests unitaires en utilisant la bibliothèque Jasmine.

Il charge le fichier de spécification "ControllerSpec.js".

La section script crée un objet app dans la fenêtre globale. Cet objet est utilisé pour stocker les données de l'application et est accessible depuis les tests.

Le fichier charge ensuite le contrôleur de l'application qui est le code JavaScript que les tests vont exercer : "controller.js"

Les résultats des tests sont affichés dans la page web.

CSS/Styles

Les styles CSS sont organisés en deux fichiers principaux :

base.css :

Définit les styles de base de l'application, tels que les polices, les couleurs de fond, et les styles généraux qui s'appliquent à toute l'application.

index.css :

Contient les styles spécifiques à l'application de liste de tâches, y compris la mise en forme des listes de tâches, des boutons, et des champs de saisie. La séparation des

fichiers CSS assure une organisation claire et maintenable des styles, permettant une modification et une extension faciles des styles de l'application.

Référence API

Ce projet n'utilise pas d'API externe. Les données sont stockées localement dans le localStorage du navigateur.

Construction et Déploiement

Le projet peut être déployé sur n'importe quel serveur web statique. Il n'y a pas de processus de construction nécessaire. Pour le déploiement, copiez simplement les fichiers sur votre serveur.