todos

documentation technique

1. le projet
2. description fonctionnelle
3. description technique
4. tests unitaires & fonctionnels
5. audit de performance
6. annexes

OpenClassrooms // Parcours développeur d'applications front-end
Projet 8 // Reprenez & améliorer un projet existant // Gaëlle Perchoc

1. le projet

- **CADRE DU PROJET**

Enjeu : Aider les gens à être organisés au quotidien.

Objectif : Création d'un outil simple de gestion de tâches.

Contraintes : Nombreuses applications existantes de to-do-list.

- **CHARTE GRAPHIQUE**

Principe : Création d'une application aux lignes simples, claires et épurées, s'appuyant sur une déclinaison de gris et de rose.

Police de caractère : Helvetica Neue

Palette de couleurs :

#AF2F2F (opacity : 0.15)	*#FFF*	*#F5F5F5*	*#D9D9D9*	*#777*	*#4D4D4D*

- **LISTE DES FONCTIONNALITÉS**

Front-end
Créer une nouvelle todo
Editer le contenu d'une todo existante
Supprimer une todo
Modifier le statut d'une todo : active ou complétée
Modifier le statut de toutes les todos : actives ou complétées
Supprimer toutes les todos complétées
Choisir un filtre d'affichage : toutes / actives / complétées

Back-End
Enregistrer une todo dans le stockage local du navigateur
Assurer la persistance des modifications apportées à une todo
Supprimer une todo du stockage local, quel que soit son statut
Supprimer toutes les todos complétées du stockage local

2. description fonctionnelle

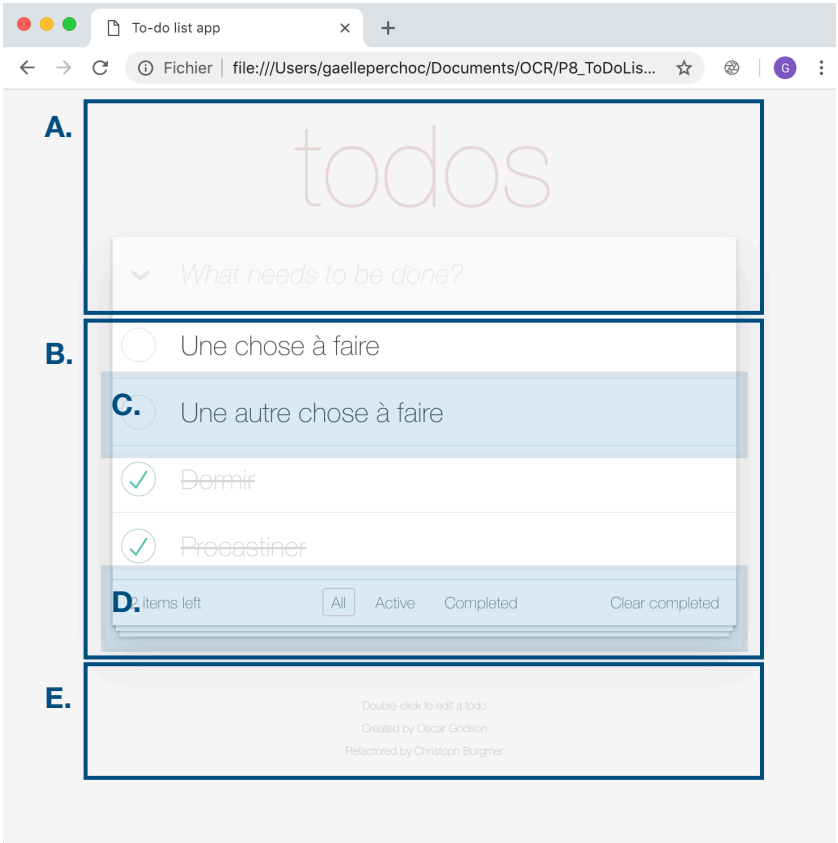
- **PRINCIPE FONCTIONNEL**

L'application comprend une seule page et s'articule autour d'un bloc unique au centre de celle-ci. L'utilisateur est invité à ajouter des tâches à réaliser (des « todos »). Une fois qu'il a saisi une ou plusieurs todos, plusieurs possibilités s'offrent à lui. Quand celles-ci sont complétées, il peut les cocher, individuellement ou collectivement. Il peut également les décocher, individuellement ou collectivement pour qu'elles soient de nouveau actives. Il peut en modifier le contenu s'il le souhaite. Pour plus de lisibilité, l'utilisateur peut choisir de modifier l'affichage de ses todos en fonction de leur statut : fait ou à faire. Enfin, il peut à tout moment supprimer une todo, quel que soit son statut. Il peut aussi en un clic supprimer toutes les todos qui ont été complétées.

- **STRUCTURATION DE LA PAGE**

La page se compose de la manière suivante :

A. Header
B. Content block
C. Todo
D. Barre d'outils
E. Footer de la page



A. Header

- **WIREFRAME**

- **DESCRIPTION FONCTIONNELLE**

Objectif : Le header affiche le nom du site « TODOS » et un formulaire interpelle le visiteur : « *What needs to be done ?* » pour l'inviter à saisir une todo.

Composition :

Le header se compose :

1. du nom du site
2. d'un formulaire pour ajouter une todo.

Interactions du point de vue d'un utilisateur :

A. L'utilisateur peut rédiger une todo dans le champ « *What needs to be done ?* » et valider l'entrée en cliquant sur la touche « Entry » de son clavier.

- **DESCRIPTION TECHNIQUE**

	Bloc	Contenu	Caractéristiques	Commentaires
1	**Nom du site**	Titre	Texte Alphanumérique <h1>	Font-family : Helvetica neue Color : rgba(175, 47, 47, 0.15)
2	**Formulaire d'ajout de todo**	1 champ de saisie	Texte Alphanumérique + caractères de ponctuation et spéciaux Placeholder : « What needs to be done ? »	

B. Content-block

- **WIREFRAME**

- **DESCRIPTION FONCTIONNELLE**

Objectif : Le content block contient la liste des todos saisies par l'utilisateur. Il s'affiche uniquement si des todos existent.

Composition :

Le content block se compose :

1. d'un bouton « Toggle All »,
2. d'une liste de todos,
3. d'une barre d'outils.

Interactions du point de vue d'un utilisateur :

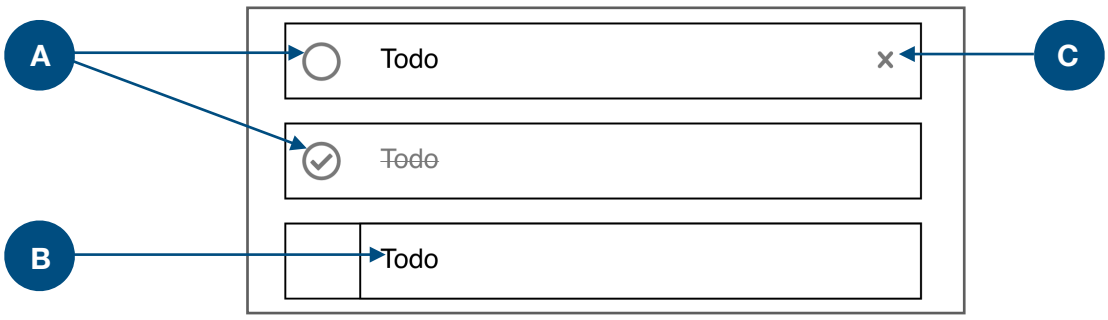
A. En cliquant sur le bouton « Toggle all », l'utilisateur peut choisir de compléter ou dé-compléter l'ensemble des todos de sa liste.

- **DESCRIPTION TECHNIQUE**

	Bloc	Contenu	Caractéristiques	Commentaires
1	**Toggle All**	Bouton	<input> [type = 'checkbox'] Symbole	Position absolute Symbole contenu dans un pseudo-élément :: before
2	**Liste des todos**	Liste de X todos		cf. Todo
3	**Barre d'outils**	Bloc		cf. Barre d'outils

C. Todo

- **WIREFRAME**



- **DESCRIPTION FONCTIONNELLE**

Objectif : Une todo est une tâche à réaliser par l'utilisateur. Son statut peut être actif ou complété. Son contenu peut être mis à jour et elle peut être supprimée par l'utilisateur.

Composition :

Une todo se compose :

1. d'un bouton « toggle »,
2. d'un contenu,
3. d'un formulaire de saisie,
4. d'un bouton « supprimer ».

Interactions du point de vue d'un utilisateur :

A. En cliquant sur le bouton toggle, l'utilisateur change le statut de la todo : actif ou complété. L'apparence de la todo est différente en fonction de son statut.

B. En double cliquant sur le contenu de la todo, l'utilisateur entre dans le mode édition. Il peut modifier le contenu de la todo. Si, suite à la modification, le champ est vide, la todo est supprimée. Pour quitter le mode édition, l'utilisateur peut cliquer sur entrée pour valider la modification ou sur « escape » pour annuler la modification. Quand l'utilisateur est en mode édition, la croix et le bouton toggle ne sont pas affichés.

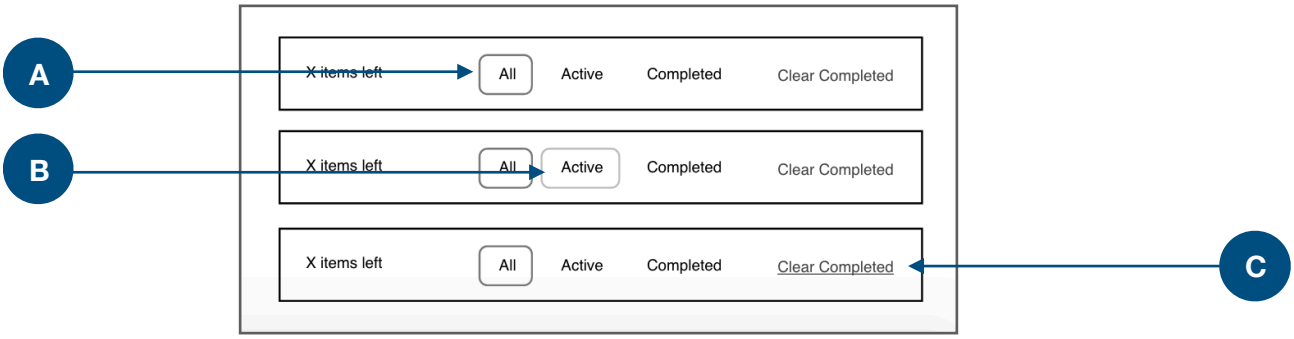
C. Au survol sur la todo, une croix apparaît. Quand l'utilisateur clique sur la croix, la todo est supprimée.

- **DESCRIPTION TECHNIQUE**

	Bloc	Contenu	Caractéristiques	Commentaires
1	**Toggle**	Bouton	<input> [type='checkbox'] svg	Attention au problème d'affichage du svg sur firefox, i.e et edge Svg contenu dans un pseudo-élément ::before Deux svgs différents suivant le statut de la todo
2	**Nom de la todo**	Texte	<label> Texte Alphanumérique + caractères de ponctuation et spéciaux	Deux apparences différentes suivant le statut de la todo
3	**Formulaire de saisie**	Champ de saisie	<input> Texte Alphanumérique + caractères de ponctuation et spéciaux	Généré suite à un double-clic sur le label contenant le nom de la todo
4	**Bouton « supprimer »**	Bouton	<button>	L'affichage du bouton n'est effectif qu'au survol de la todo

D. Barre d'outils

- **WIREFRAME**



- **DESCRIPTION FONCTIONNELLE**

Objectif :

La barre d'outils contient des boutons pour la gestion de la liste des todos.

Composition :

Le footer du content-block se compose :

1. d'un span indiquant le nombre de todos actives,
2. de trois boutons de filtres qui permettent d'afficher au choix : toutes les todos, les todos actives ou les todos complétées,
3. d'un bouton pour supprimer toutes les todos complétées.

Interactions du point de vue d'un utilisateur :

A. En cliquant sur le bouton « All », toutes les todos sont affichées quelque soit leur statut. C'est le bouton actif par défaut. En cliquant sur le bouton « Active », seules les todos actives sont affichées. Si aucune todo n'est active, alors la liste sera vide. En cliquant sur le bouton « Completed », seules les todos complétées sont affichées. Si aucune todo n'est complétée, alors la liste sera vide.

B. Au survol des boutons de filtre, une bordure rosée apparaît.

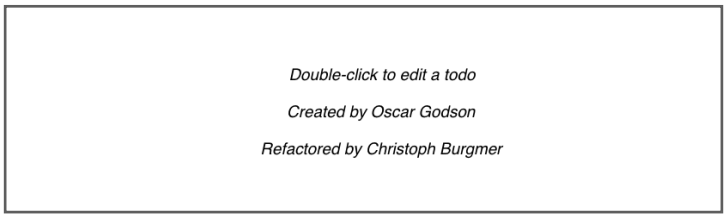
C. Dès qu'une todo est complétée, le bouton « clear completed » s'affiche dans le footer. Il permet de supprimer toutes les todos complétées, quel que soit le réglage du filtre. Au survol de ce bouton, le texte est souligné.

- **DESCRIPTION FONCTIONNELLE**

	Bloc	Contenu	Caractéristiques	Commentaires
1	**Nombre de todos actives**	Texte	 Texte Alphanumérique + caractères de ponctuation et spéciaux	En fonction du nombre d'items actifs, il faut gérer le pluriel et le singulier du mot « item »
2	**Filtre**	Bouton « All »	Texte Alphanumérique <a>	Le bouton 'All' est actif par défault Quand un bouton est actif, il dispose d'une bordure Quand un bouton est inactif, pas de bordure Au survol, une bordure d'une couleur différente de celle appliquée quand un bouton est actif est également appliquée
		Bouton « Active »		
		Bouton « Completed »		
3	**Bouton « Clear completed »**	Bouton	Texte Alphanumérique <button>	N'est affiché que si des todos sont complétées Au survol du bouton, le texte est souligné

E. Footer

- **WIREFRAME**



- **DESCRIPTION FONCTIONNELLE**

Objectif : Le footer donne des informations sur la manière d'utiliser l'application et des informations relatives aux créateurs de l'application.

Composition :

Le footer comprend du texte.

Interactions du point de vue d'un utilisateur :

Pas d'interactions.

- **DESCRIPTION TECHNIQUE**

Bloc	Contenu	Caractéristiques	Commentaires
Footer	Texte	Texte Alphanumérique <p>	Font-style : italique

3. description technique

- **CHOIX TECHNOLOGIQUES**

L'application fonctionne avec un ensemble de fichiers javascript. Elle ne fait appel à aucun framework ou librairie externe.

1. Structure des fichiers js de l'application

- **PATRON D'ARCHITECTURE MVC**

L'application est organisée selon une architecture MVC (Modèle - Vue - Contrôleur).
L'objectif de ce patron est de séparer la logique du code en trois parties distinctes :

- Le **modèle**. Il contient la logique métier et l'état courant de l'interface pendant le cycle de dialogue avec l'utilisateur.
- La **vue**. Elle porte la logique de présentation. Elle affiche les données du domaine et reçoit les interactions de l'utilisateur dont elle délègue la gestion au contrôleur.
- Le **contrôleur**. Il reçoit les interactions de l'utilisateur de la vue et les traite en modifiant le modèle. Il assure le lien entre le modèle et la vue.

Le patron d'architecture MVC fonctionne en cycle. L'utilisateur interagit avec les composants de la vue qui sont à sa disposition. Cela déclenche la création d'événements qui sont envoyés au contrôleur. Le contrôleur vérifie la conformité des interactions et en déduit les modifications à apporter au modèle qui les gère en fonction de ses règles métiers. Les modifications du modèle sont ensuite signalées à la vue qui se met à jour en conséquence.
Ce patron présente plusieurs avantages : clarté, lisibilité et modularité du code, meilleure maintenabilité de l'application, facilité de mise à jour et une séparation claire des responsabilités des méthodes pour une meilleure cohésion et un couplage plus faible.

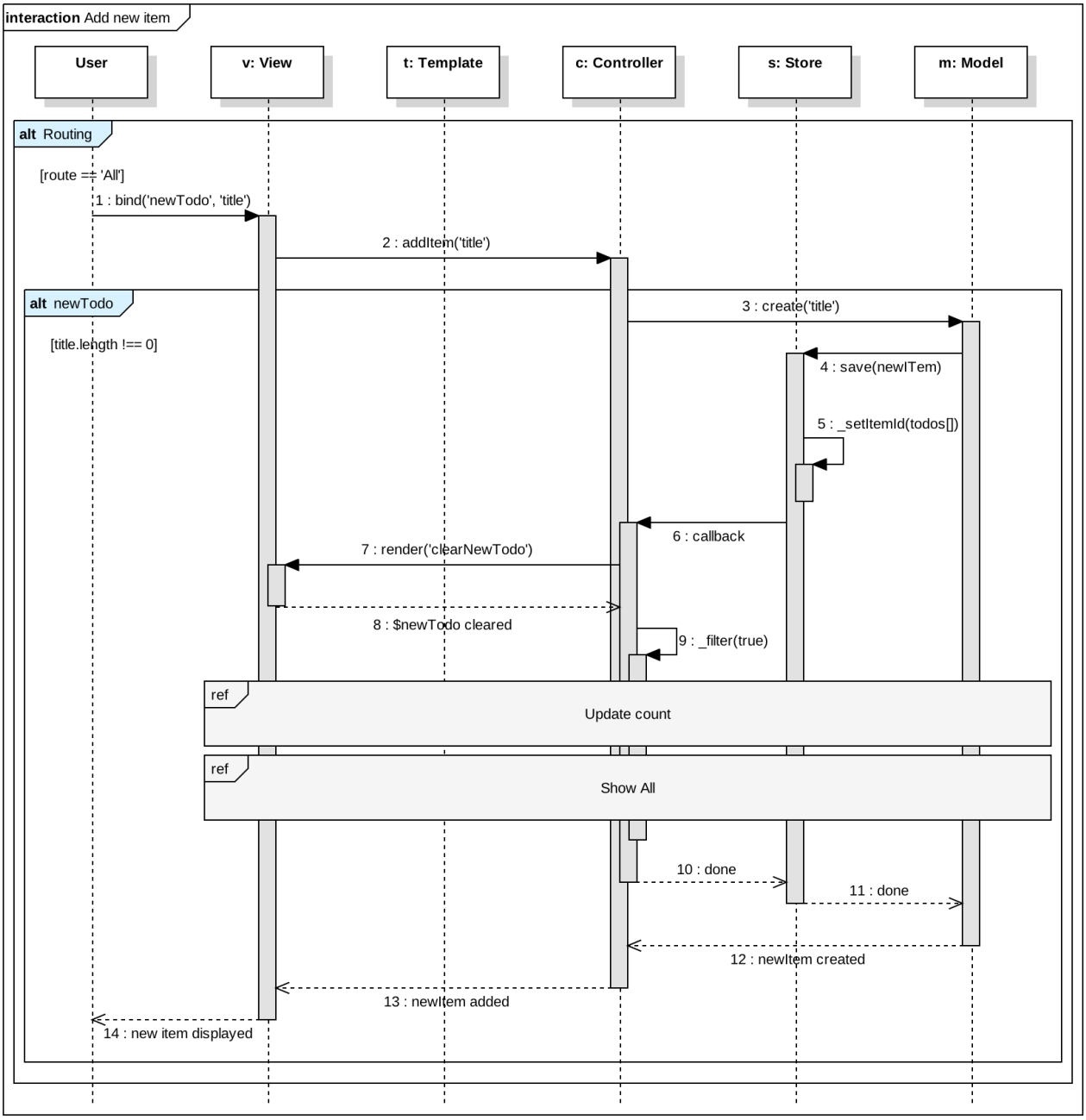
Dans le cadre de notre application, la classe **Model**, contenue dans le fichier Model.js, représente le modèle de l'application. Elle définit les méthodes de création et de mises à jour des todos en mémoire, et ce, grâce à la classe **Store** contenue dans le fichier Store.js qui joue le rôle de base de données. Ici, le stockage ne se fait pas dans une véritable base de données mais dans le stockage local du navigateur.
La classe **View**, contenue dans le fichier View.js, représente la vue de l'application. Elle écoute les différents événements sur la page, transmet l'ajout et les modifications des todos et met à jour l'apparence de la page, grâce à la classe **Template** contenue dans le fichier Template.js.
La classe **Controller**, présente dans le fichier Controller.js, fait le lien entre le modèle et la vue. Elle va interpréter les événements qui se produisent sur la page pour pourvoir donner des instructions au modèle et en retour, demander à la vue de mettre la page à jour.
Lors de sa propre instanciation, la classe **Todo**, contenue dans le fichier app.js, génère une instance de chacune de ces classes.
Enfin, un fichier helpers.js déclare un ensemble de méthodes globales et récurrentes telles que $on pour la gestion des événements, $qsa pour la sélection d'éléments dans le DOM etc…

Les méthodes de chacune des classes de l'application sont détaillées en annexes.

A. Diagrammes de séquence : fonctionnalités

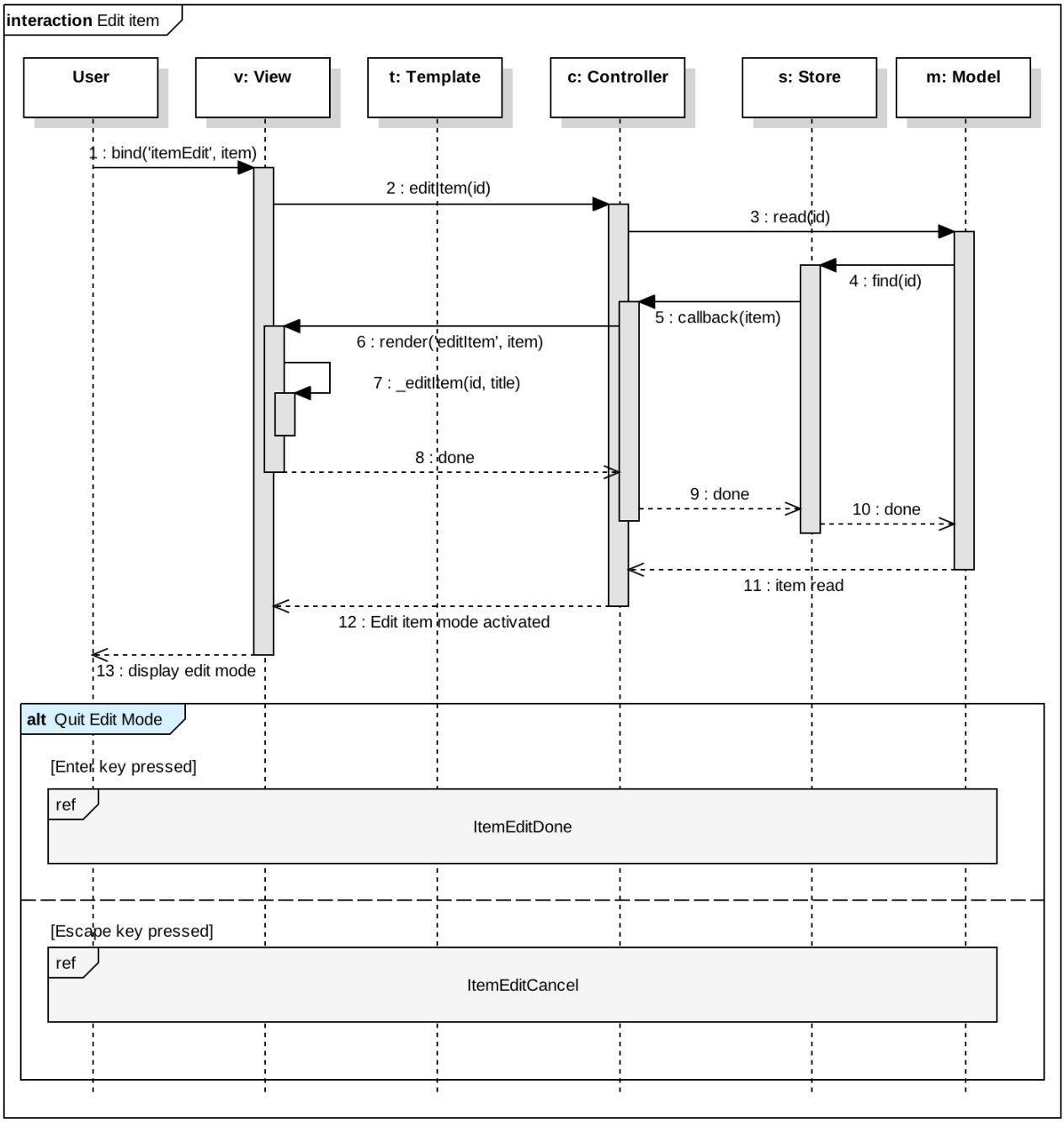
Dans cette partie, un ensemble de diagrammes de séquence illustre pour chacune des fonctionnalités de l'application la manière dont les différents composants du système interagissent entre eux et avec l'utilisateur. Dans un souci de lisibilité, certaines méthodes font l'objet de diagrammes de séquences qui leur sont propres.

- **AJOUTER UNE NOUVELLE TODO**[1]



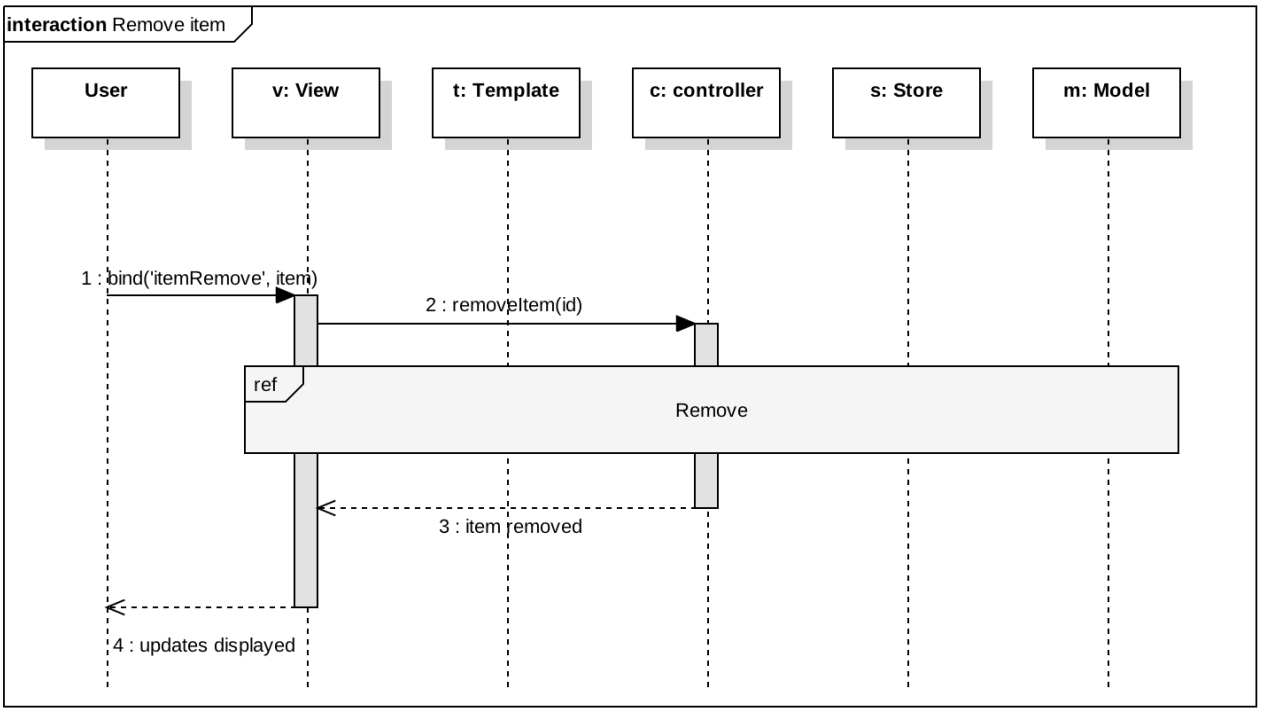
[1] La sous-séquence « Update Count » est détaillée p.24.
La sous-séquence « Show All » est détaillée p.25.

- **EDITER UNE TODO**[2]



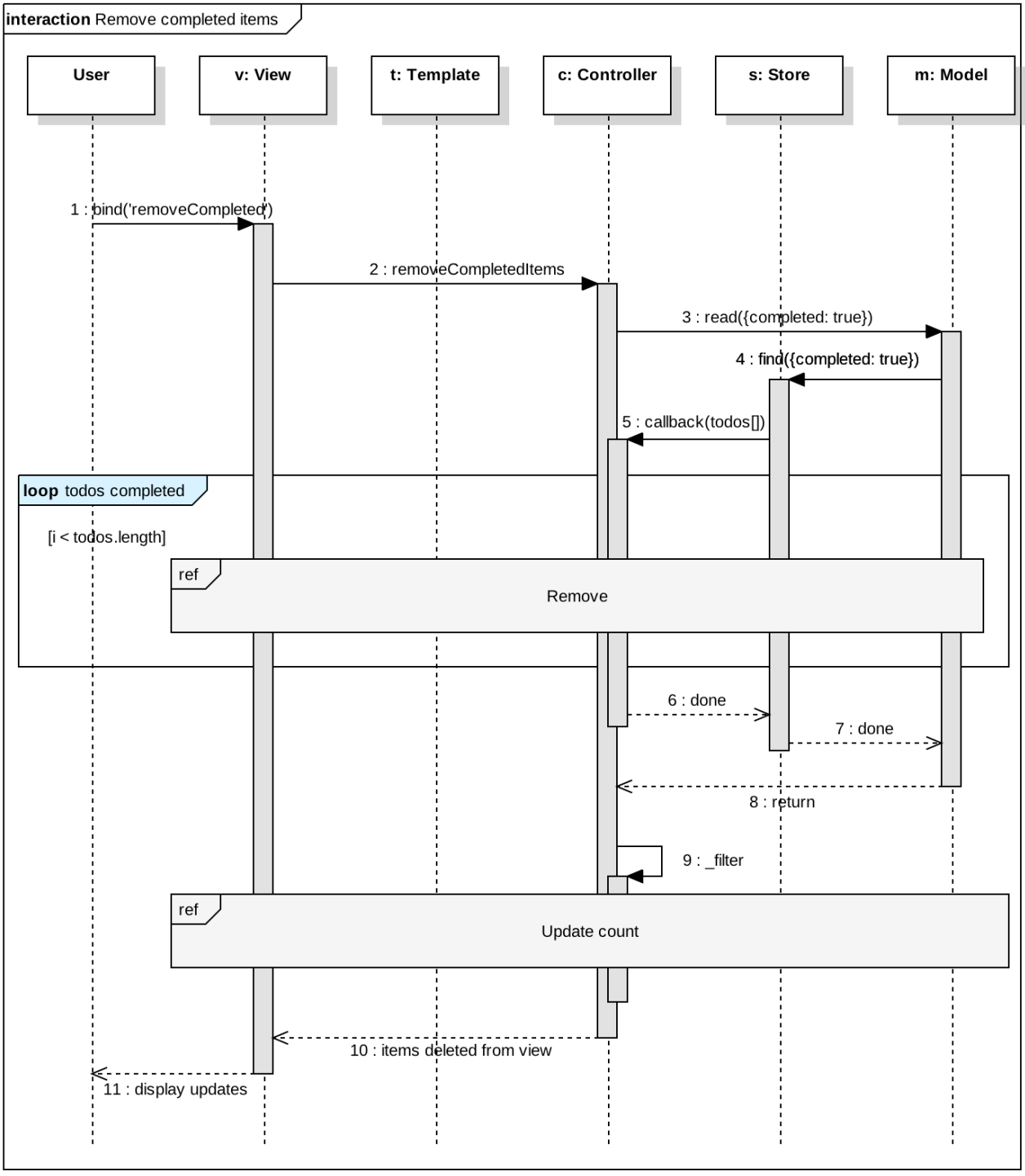
[2] La sous-séquence « ItemEditDone » est détaillée p22.
La sous-séquence « ItemEditCancel » est détaillée p.23.

- **SUPPRIMER UNE TODO[3]**



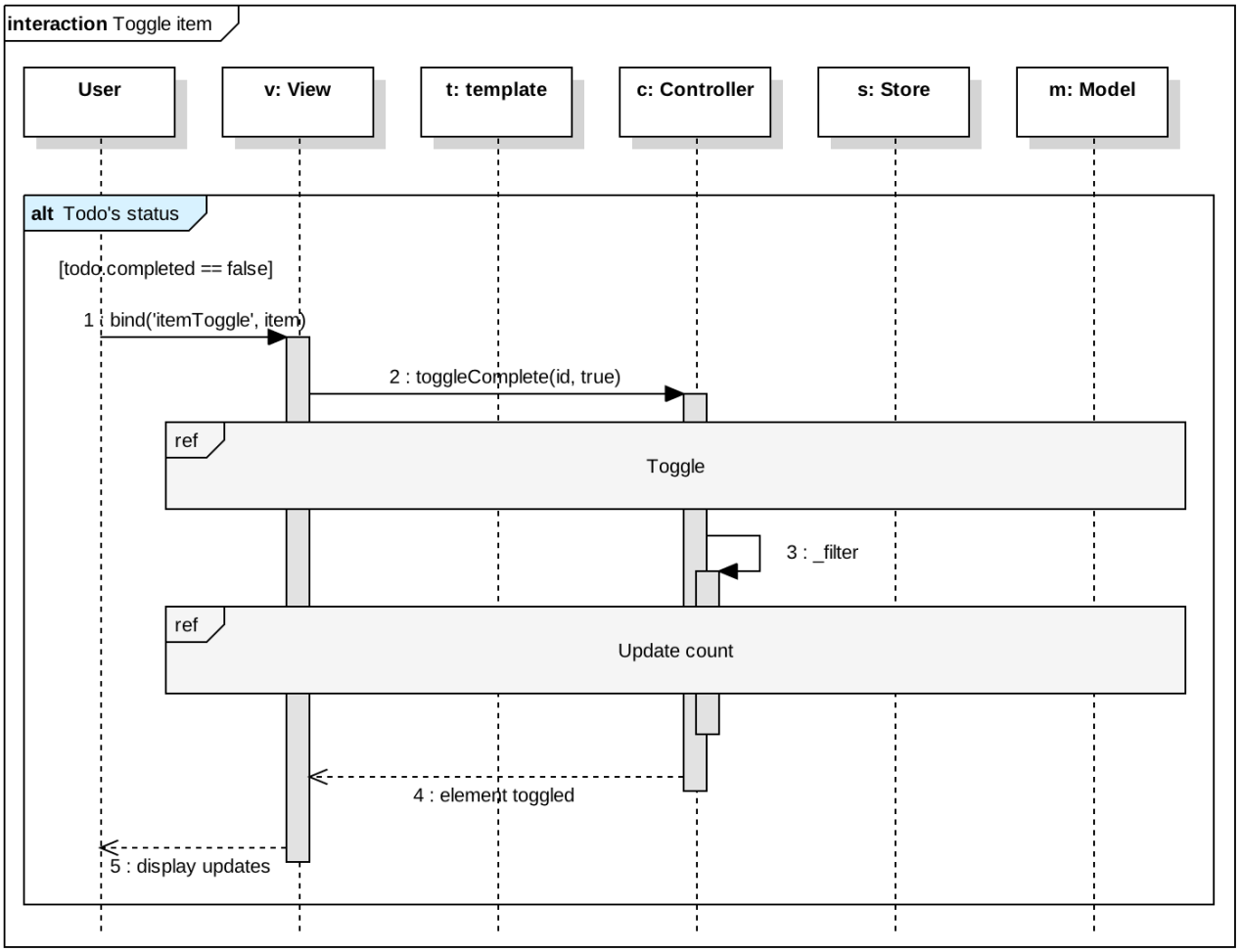
[3] La sous-séquence « remove » est détaillée p.20.

- **SUPPRIMER TOUTES LES TODOS COMPLÉTÉES**[4]



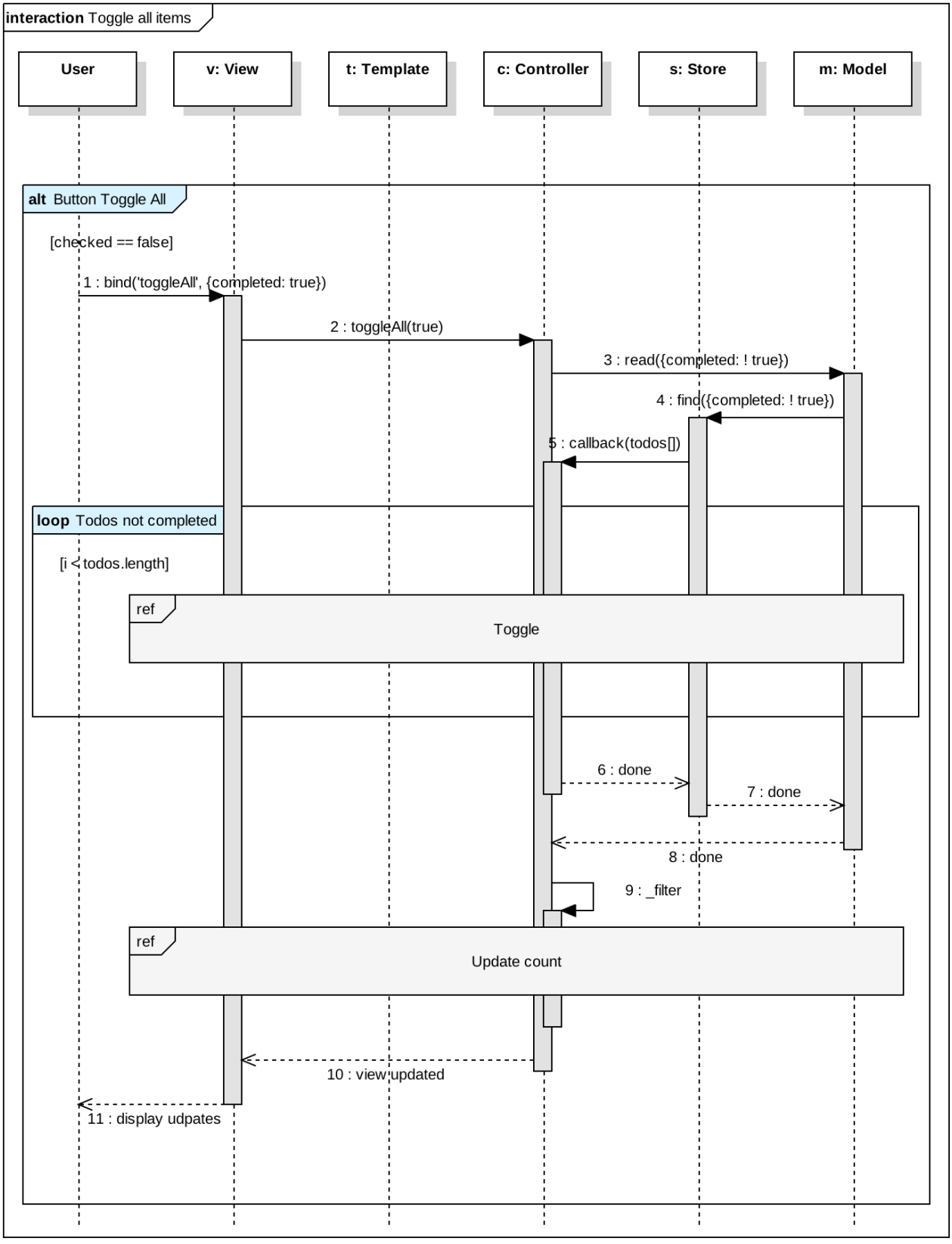
[4] La sous-séquence « Remove » est détaillée p.20.
La sous-séquence « Update count » est détaillée p.24.

- **CHANGER LE STATUT D'UNE TODO[5]**



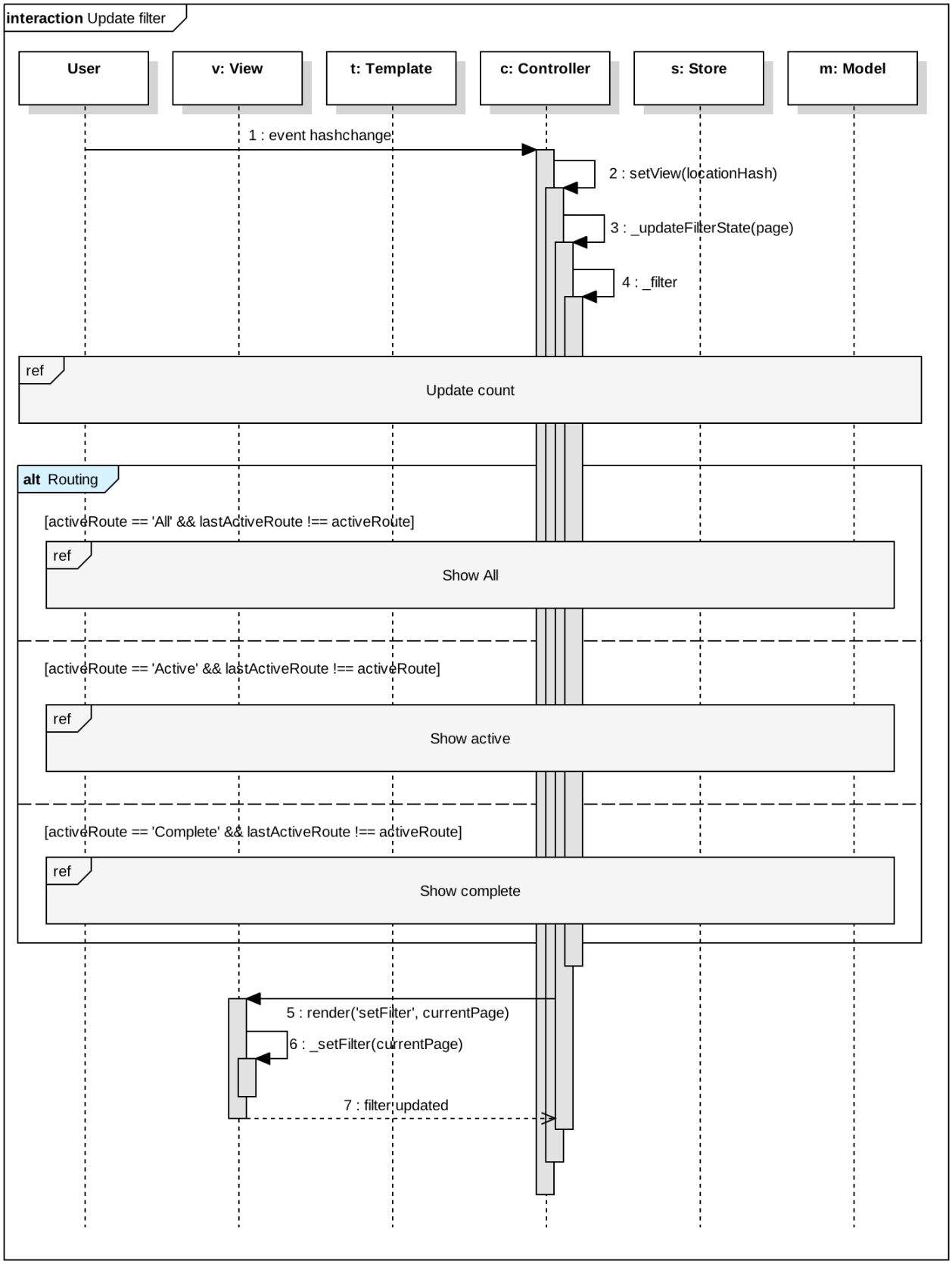
[5] La sous-séquence « Toggle » est détaillée p.21.
La sous-séquence « Update count » est détaillée p.24.

- **CHANGER LE STATUT DE TOUTES LES TODOS**[6]



[6] La sous-séquence « Toggle » est détaillée p.21.
La sous-séquence « Update count » est détaillée p.24.

- **CHOISIR LE FILTRE D'AFFICHAGE DES TODOS**[7]

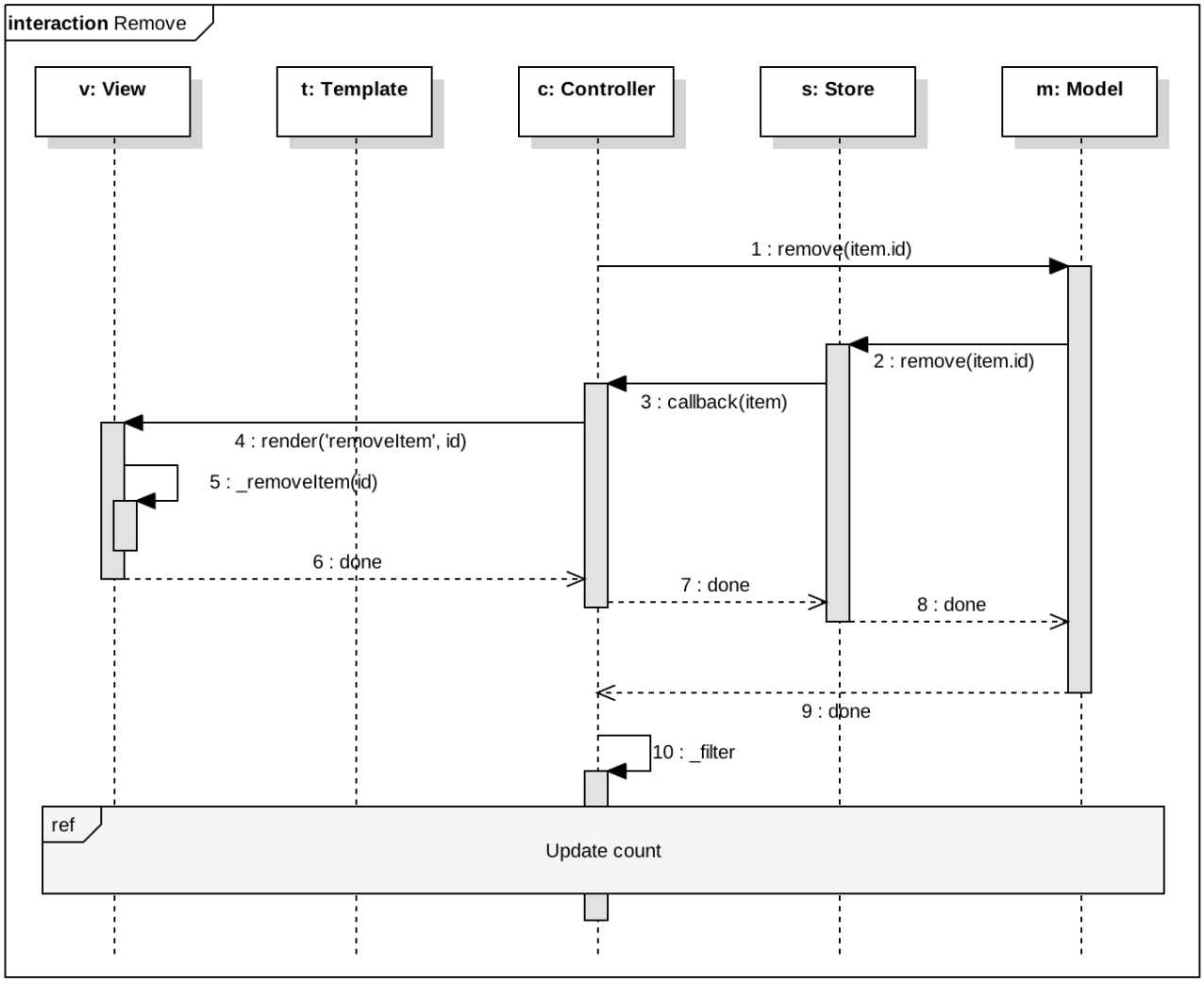


[7] La sous-séquence « Update count » est détaillée p.24.
La sous-séquence « Show all » est détaillée p.25.
La sous-séquence « Show active » est détaillée p.26.
La sous-séquence « Show completed » est détaillée p.27.

B. Fragments de séquence

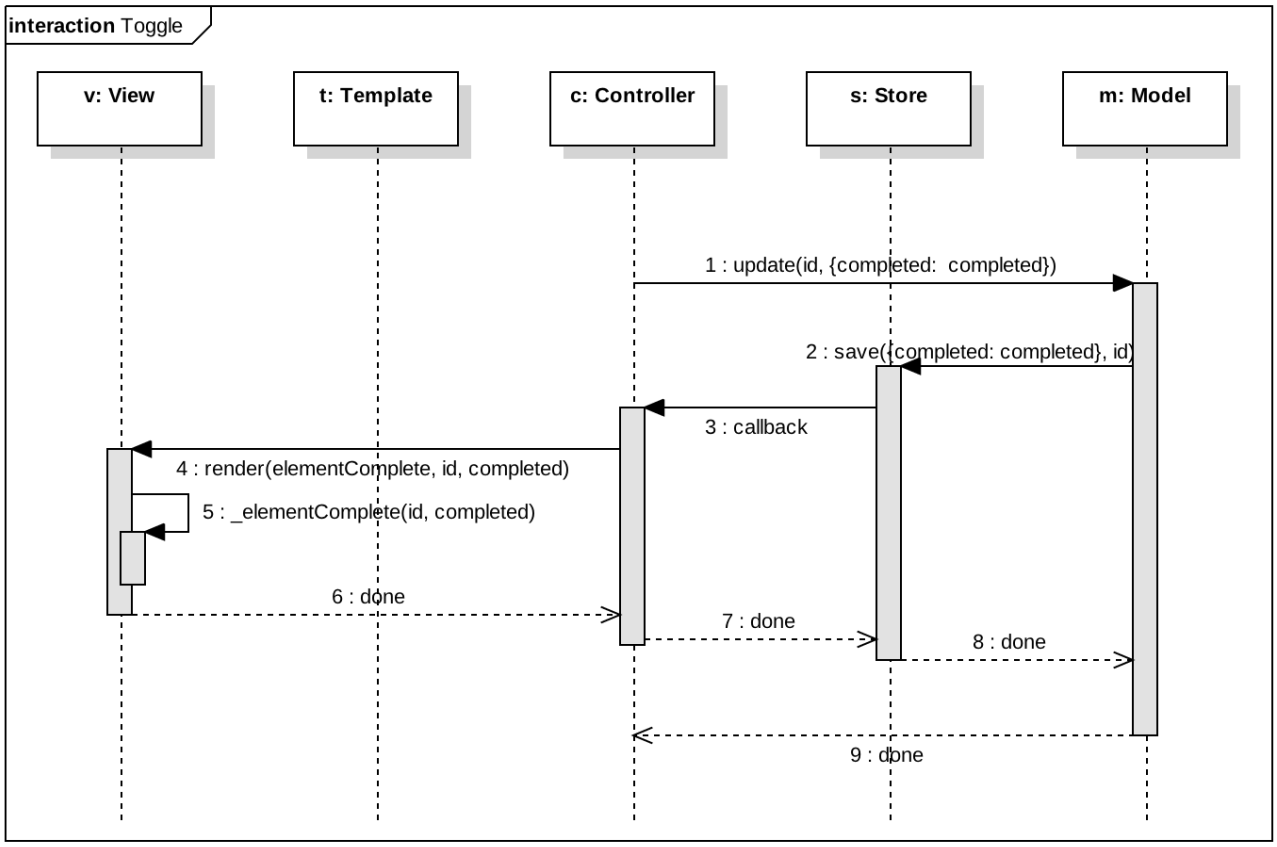
Ici suivent les diagrammes de séquence auxquels il est fait référence dans les diagrammes de séquence précédents, illustrant les fonctionnalités de l'application.

- **REMOVE**[8]

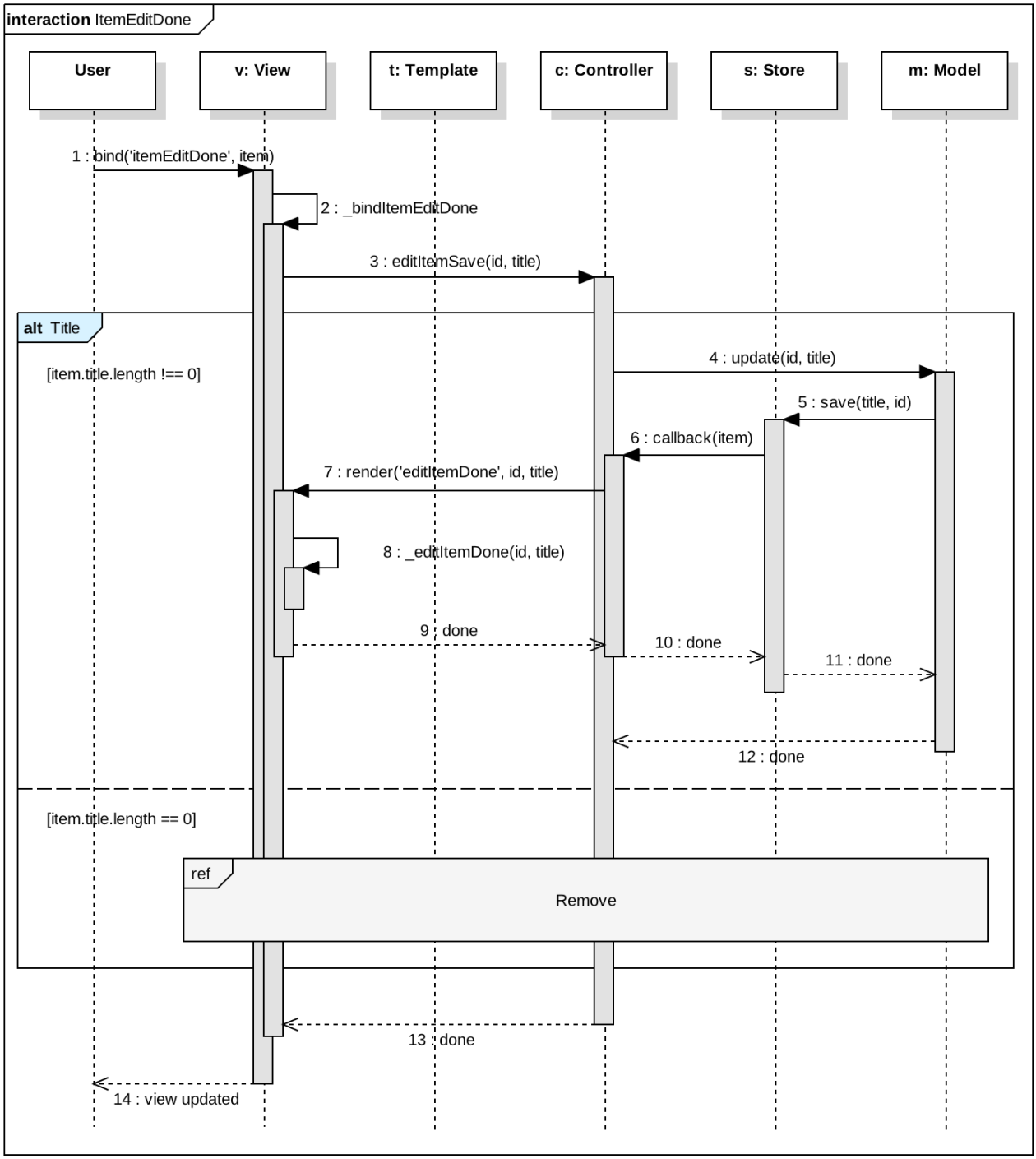


[8] La sous-séquence « Update count » est détaillée en page 24.

- **TOGGLE**

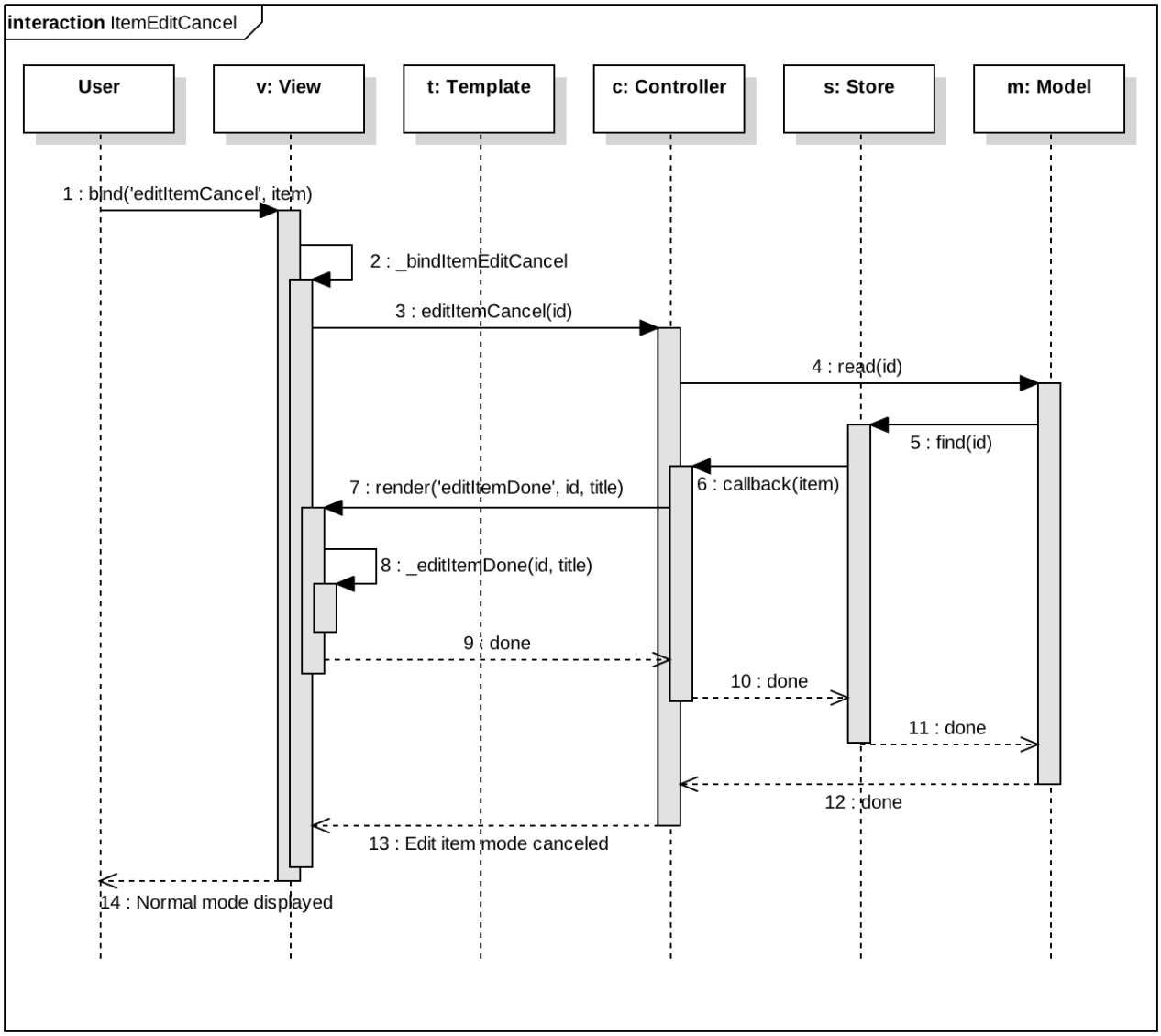

interaction Toggle
v: View
t: Template
c: Controller
s: Store
m: Model
1 : update(id, {completed: completed})
2 : save({completed: completed}, id)
3 : callback
4 : render(elementComplete, id, completed)
5 : _elementComplete(id, completed)
6 : done
7 : done
8 : done
9 : done

- **ITEM EDIT DONE**[9]

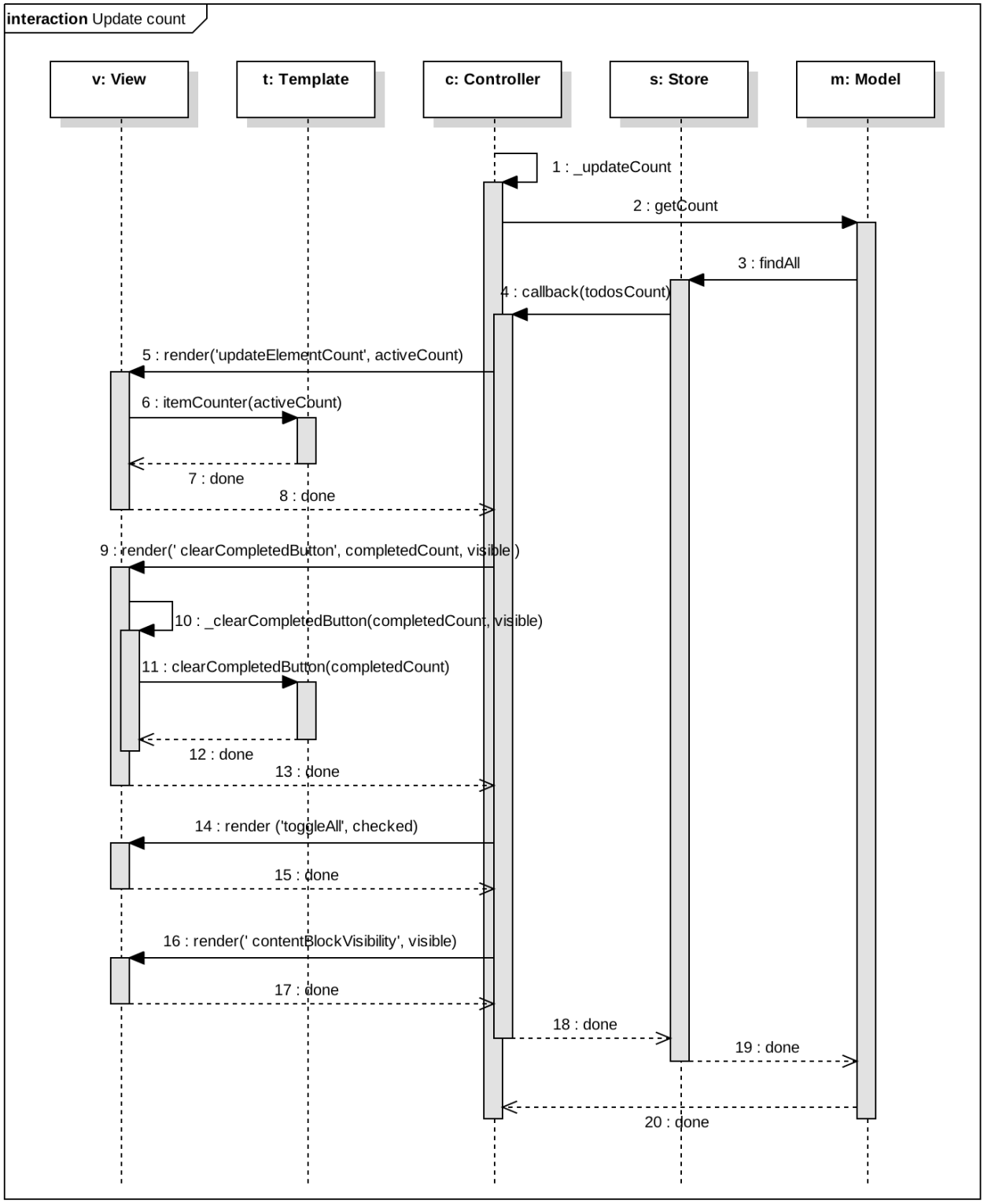


[9] La sous-séquence « remove » est détaillée en page 20.

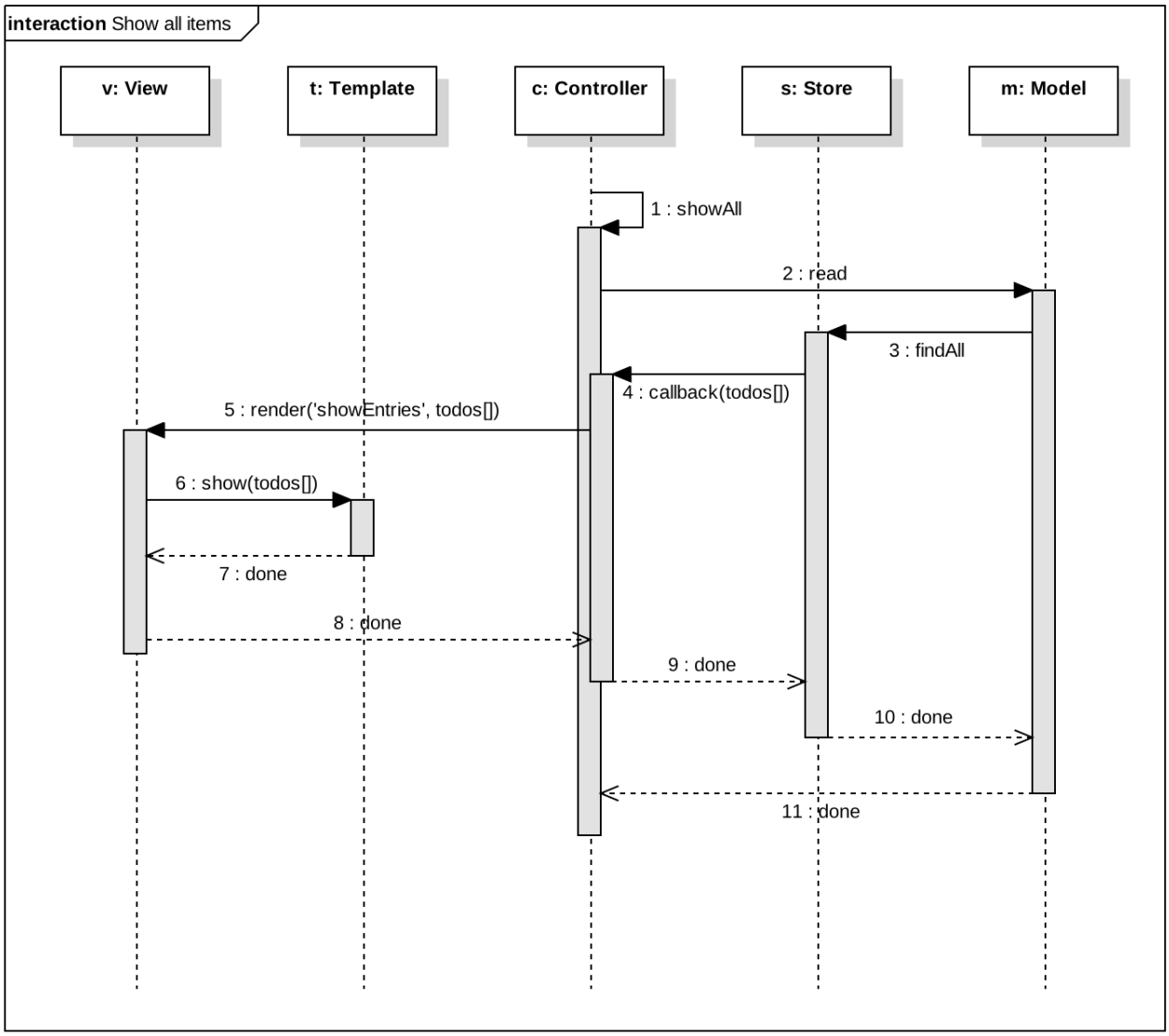
- **EDIT ITEM CANCEL**



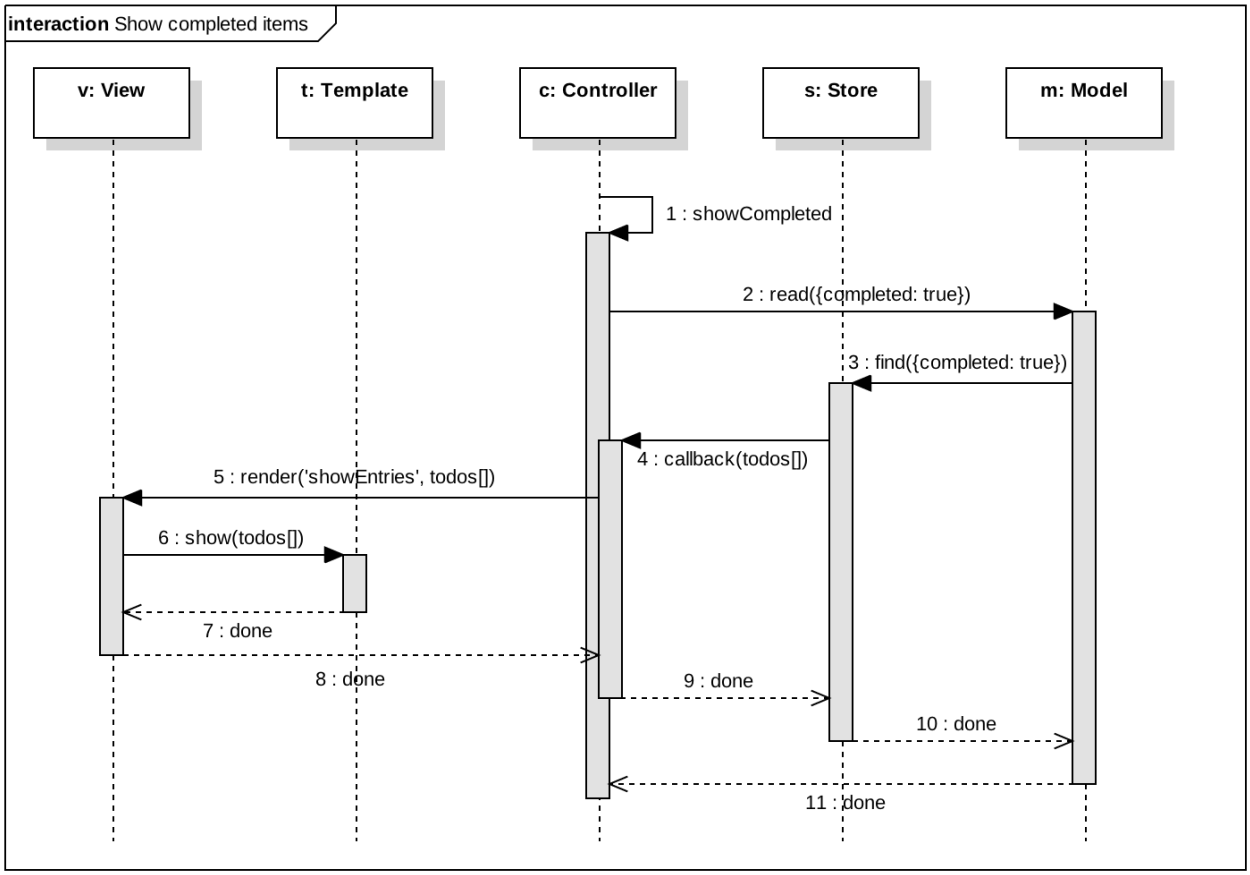
- **UPDATE COUNT**



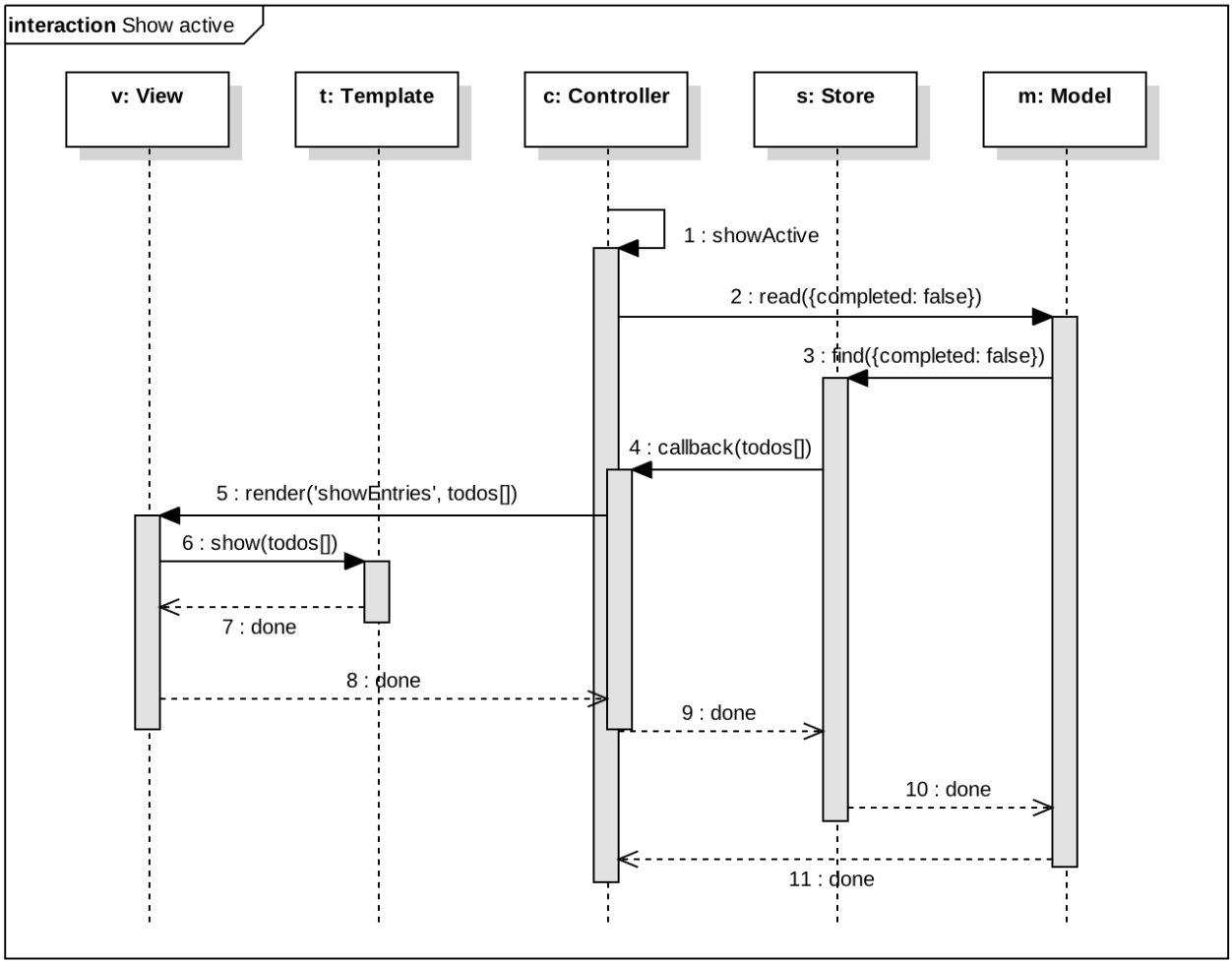
- **SHOW ALL ITEMS**



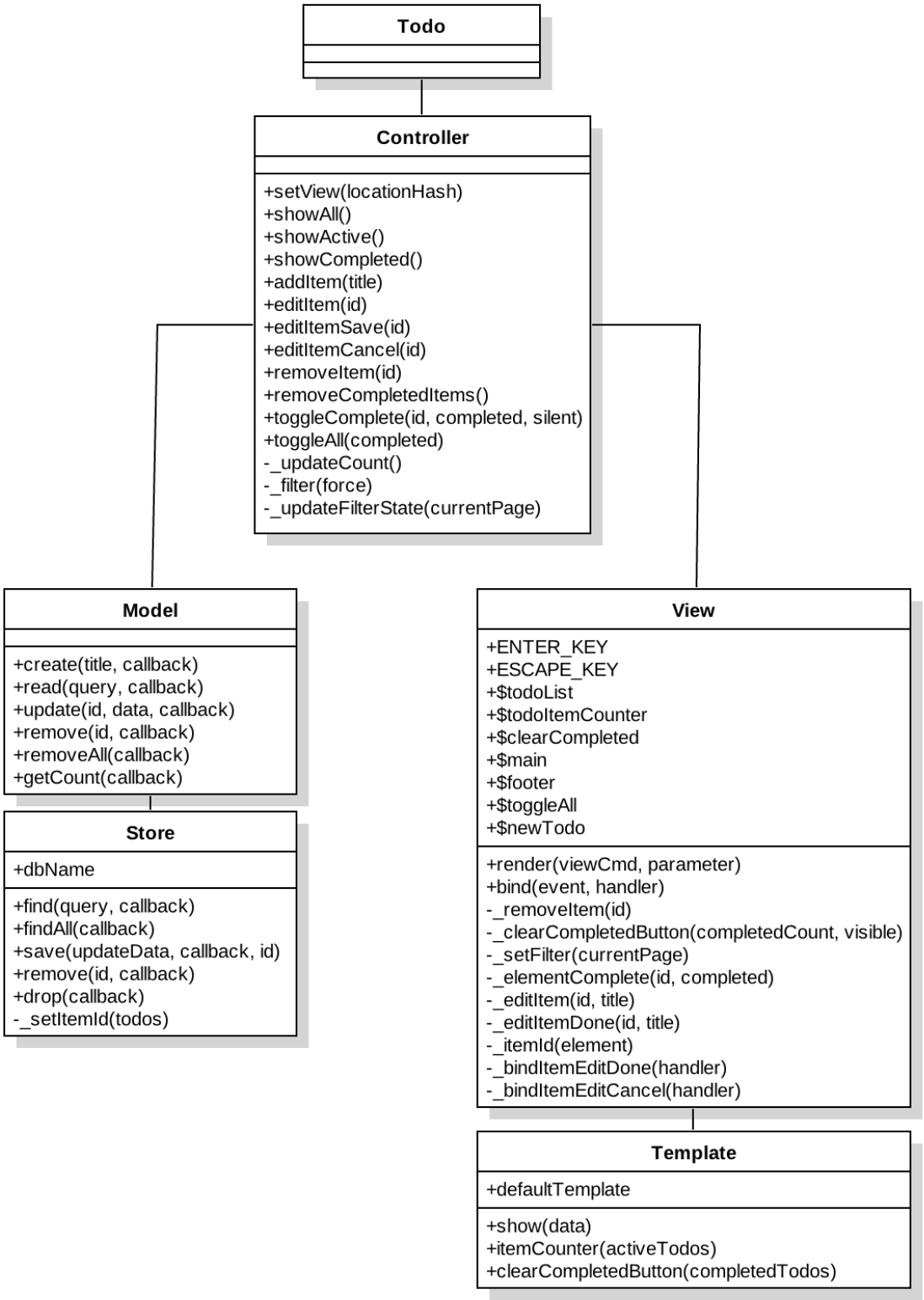
- **SHOW ACTIVE ITEMS**



- **SHOW COMPLETED ITEMS**



C. Diagramme de classes



4. tests unitaires & fonctionnels

Quelle que soit l'étape du projet, les tests représentent une étape importante dans le développement. S'ils ont été structurés et écrits en amont, ils aident par la suite à l'écriture du code de l'application. Puis, ils permettent de s'assurer du bon fonctionnement des fonctionnalités. Ils protègent également le code des modifications qui peuvent involontairement le casser.

- **Les tests**

Une série de tests a été mise en place à l'aide de Jasmine, un « behavior-driven-development » framework. Ces tests vont permettre de vérifier un ensemble de situations pendant le fonctionnement de l'application :

- Ma todo est-elle bien ajoutée au stockage local une fois entrée via le formulaire ?
- La vue se met-elle à jour en changeant de filtre ?
- etc.

- **Exemple de tests**

Dans cet exemple, la fonctionnalité d'édition d'une todo est testée et plus spécifiquement, l'activation du mode édition. Pour ce faire, on simule une situation - l'existence d'une todo « my todo », une vue positionnée sur le filtre 'All' - et l'appel de l'événement 'itemEdit' pour « my todo ». Pour que le test soit un succès, la méthode render de la vue est espionnée afin de savoir si elle a été appelée et ce, avec les bons paramètres, ici 'editItem' et la todo en question.

```
describe('edit item', function () {
    it('should switch to edit mode', function () {
        var todo = {id: 21, title: 'my todo', completed: false};
        setUpModel([todo]);
        subject.setView('');
        view.trigger('itemEdit', {id: 21});
        expect(view.render).toHaveBeenCalledWith('editItem', {id: 21, title: 'my todo'});
    });
});
```

5. audit de performance

Ici, nous livrerons l'analyse des performances menée sur une application concurrente : todolistme.net. Toutefois, il convient de souligner que cette application est difficilement comparable à la notre en l'état. En effet, l'étendue des fonctionnalités diffère d'une application à l'autre. *Todos* est une application basique qui consiste en une liste unique de tâches à réaliser, quand *todolistme* offre un système de listes multiples, de catégories et de planification des tâches. D'un point de vue technique également, *todolistme* fait appel à des librairies externes et comprend des third parties, ce qui n'est pas le cas de notre application. De plus, *todolistme* ne semble pas disposer de mises à jour depuis 2013, ce qui laisse supposer le recours à des technologies, des API et des pratiques obsolètes. Ainsi, plutôt que d'envisager de scaler notre application sur *todolistme*, l'audit portant sur ses performances permettra davantage de pointer ce qui ne fonctionne pas et ainsi de formuler des préconisations applicables à notre application pour ses développements futurs.

- **Méthodologie et outils**

Pour cadrer la réalisation de cet audit de performance, nous nous sommes appuyés sur les préconisations du RAIL Model performance proposées par les équipes de Google[10] et une méthodologie proposée par Addy Osmani[11], qui place l'utilisateur au centre de l'analyse des performances[12]. Les tests ont été menés avec les outils de développement du navigateur Chrome (onglets, Network, Performance et Audit), WebPageTest et RequestMap. Ils ont été simulés sur différents devices, différents CPU et différentes connexions.

- **Etat des lieux**

Que ce soit sur desktop ou sur mobile, quelle que soit la qualité du device ou de la connexion, les performances du site au premier comme au second lancement sont problématiques. Ce constat est valable pour le temps de chargement et pour la qualité du critical rendering path. Le chargement complet dépasse largement la fenêtre des 5 secondes préconisée par le RAIL Model performance. La page n'est pas interactive avant plusieurs secondes et l'ordre d'affichage des éléments n'est pas particulièrement pertinent. De plus, le CPU et la bande passante sont activement sollicités. Une fois la page chargée, certains événements ne rencontrent pas non plus de réponses dans le laps des 50 ms préconisés par le RAIL Model performance.

---

[10] Measure Performance with the RAIL Model : https://developers.google.com/web/fundamentals/performance/rail

[11] Addy Osmani. The cost of Javascript in 2018 : https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4

[12] Cf. Annexe.

- **Préconisations**

Un certain nombre d'actions peuvent être aisément mises en place pour améliorer les performances de la page :

1. L'organisation du document HTML est à revoir. Des balises <script> sont disséminées à travers le document, ce qui bloque le chargement de la page. Il serait judicieux de les placer à la fin du document.
2. Le nombre de requêtes est particulièrement conséquent (près de 70). Il serait possible d'en réduire le nombre global. En effet, plusieurs requêtes sont consacrées au chargement d'images. Or, la plupart ces images sont utilisées comme des icônes mais le recours à des icônes en format svg serait bien plus pertinent. De même, l'arrière-plan du site est une image. L'usage d'un aplat de couleur via des propriétés CSS serait plus judicieux.
3. L'application fait appel à jQuery. Mais, elle exploite assez peu ses potentialités et son utilisation n'est pas nécessairement justifiée. Elle n'en reste pas moins coûteuse en ressource et provoque des ralentissements pendant et après le chargement.
4. Il serait également possible de réduire le nombre de requêtes au chargement. En effet, une grand nombre d'entre elles concernent les publicités qui apparaissent sur le site. Celles-ci affectent le rendering path en monopolisant le thread principal avant même que les fonctionnalités principales du site aient été chargées et soient utilisables. L'emploi de l'attribut «defer » dans la balise <script> permettrait de reporter l'exécution de ces requêtes à la fin du chargement de la page.
5. La mise en cache de certaines ressources permettrait également d'améliorer le chargement de la page lors des visites ultérieures de l'utilisateur.
6. L'application a recours à des iframes qui ont un impact sur les performances mais aussi sur la sécurité. Dans le cas des boutons de partage sur les réseaux sociaux, un lien cliquable et un script asynchrone sont aujourd'hui préconisés dans les documentations.

Ces pistes de préconisations permettrait d'améliorer la performance mais ne sauraient toutefois être exhaustives. La mise en place d'outils de Real User Monitoring, le code-splitting, l'analyse du critical rendering path, le recours à un CDN sont autant de pistes qui pourraient être creusées pour améliorer la performance de ce site et être pensées dans le développement de notre application.

7. annexes

A. Détails des méthodes de la classe Model.js

create(title, callback) : Création d'une nouvelle todo.

Paramètres	Type	Description
title	string	Titre de la todo
callback	function	Fonction à exécuter une fois que la todo a été créée

read(query, callback) : Recherche et retourne la todo répondant à la requête passée en paramètre.

Paramètres	Type	Description
query	string | number | object	Requête
callback	function	Fonction à exécuter une fois que la todo a été

update(id, data, callback) : Mise à jour des données de la todo dont l'id est passé en paramètre.

Paramètres	Type	Description
id	number	Id de la todo à mettre à jour
data	object	Propriétés et nouvelles valeurs de la todo
callback	function	Fonction à exécuter une fois que la modification a été réalisée

remove(id, callback) : Retire de la base de données la todo dont l'id est passé en paramètre.

Paramètres	Type	Description
id	number	Id de la todo à supprimer
callback	function	Fonction à exécuter une fois que la todo a été supprimée de la base de données

removeAll(callback) : Retire toutes les todos de la collection.

Paramètres	Type	Description
callback	function	Fonction à exécuter une fois que le stockage a été vidé.

getCount(callback) : Retourne le nombre total de todos contenues dans la collection.

Paramètres	Type	Description
callback	function	Fonction à exécuter une fois que le calcul a été réalisé.

B. Détails des méthodes de la classe Store.js

find(query, callback) : Trouve la todo répondant à la requête passée en paramètre.

Paramètres	Type	Description
query	object	Requête
callback	function	Fonction à exécuter une fois que la requête a été traitée

findAll(callback) : Trouve toutes les todos appartenant à la collection

Paramètres	Type	Description
callback	function	Fonction à exécuter une fois que la récupération des données est achevée

save(updateData, callback, id) : Enregistre la todo dans la base. Si la todo n'existait pas, crée une nouvelle todo, sinon, met à jour les propriétés de la todo existante dont l'id est passé en paramètre.

Paramètres	Type	Description
updateData	object	La donnée à enregistrer dans la base de données
callback	function	Fonction à exécuter une fois que la récupération des données est achevée
id	number	Id de la todo à mettre à jour

remove(id, callback) : Supprime de la base de donnée une todo dont l'id est passé en argument.

Paramètres	Type	Description
id	number	Id de la todo à supprimer
callback	function	Fonction à exécuter une fois que la todo a été supprimée.

drop(callback) : Supprime toutes les todos contenues dans la base de données.

Paramètres	Type	Description
callback	function	Fonction à exécuter une fois que la base de données a été supprimée

C. Détails des méthodes de la classe View.js

_removeItem(id) : Retire de la vue la todo dont l'id est passé en paramètre

Paramètres	Type	Description
id	number	Id de la todo à supprimer

_clearCompletedButton(completedCount) : Affiche le bouton « clear completed » si l'argument « visible » est vraie.

Paramètres	Type	Description
completedCount	number	Total de todos complétées
visible	boolean	Détermine la visibilité de l'élément

_elementComplete(id, completed) : Affiche la todo dont l'id est passé en paramètre comme complété si l'argument completed est vraie, sinon elle est affichée comme active.

Paramètres	Type	Description
id	number	Id de la todo
completed	boolean	Détermine le statut de la todo

_setFilter(currentPage) : Met en valeur le bouton de filtre correspondant au nom de la page passé en paramètre.

Paramètres	Type	Description
currentPage	string	Nom de la page

_editItem(id, title) : Affiche le mode édition de la todo dont l'id est passé en paramètre.

Paramètres	Type	Description
id	number	Id de la todo
title	string	Titre de la todo

_editItemDone(id, title) : Quitte le mode édition de la todo dont l'id est passé en paramètre et affiche le titre passé en paramètre.

Paramètres	Type	Description
id	number	Id de la todo
title	string	Titre de la todo

render(viewCmd, parameter) : Exécute la commande passée en paramètre et appelle la méthode privée correspondante. Met à jour le DOM.

Paramètres	Type	Description
viewCmd	string	Nom de la méthode privée à exécuter
parameter	object | string	Paramètres de la méthode passée en paramètre

_itemId(element) : Retourne l'id d'une todo.

Paramètres	Type	Description
element	HTML Element	Element du DOM affichant les informations d'une todo

_bindItemEditDone(handler) : Quitte le mode édition en appuyant sur la touche « enter » ou en cliquant hors du champ de saisie.

Paramètres	Type	Description
handler	function	Méthode attachée à l'exécution de l'événement

_bindIdemEditCancel(handler) : Quitte le mode édition en appuyant sur la touche « escape ».

Paramètres	Type	Description
handler	function	Méthode attachée à l'exécution de l'événement

bind(event, handler) : Associe une liste d'événements à une méthode passée en paramètre.

Paramètres	Type	Description
handler	function	Méthode attachée à l'exécution de l'événement

D. Détails des méthodes de la classe Template.js

show(data) : Retourne une chaîne de caractères comprenant des éléments HTML et dont les balises ont été complétées à l'aide de l'objet passé en paramètre.

Paramètres	Type	Description
data	object	L'objet todo et ses propriétés

itemCounter(activeTodos) : Retourne le nombre de todos actives dans une chaîne de caractères.

Paramètres	Type	Description
activeTodos	number	Le nombre de todos actives

clearCompletedButton(completedTodos) : Met à jour le texte dans le bouton « clear completed ».

Paramètres	Type	Description
completedTodos	number	Le nombre de todos complétées

E. Détails des méthodes de la classe Controller.js

setView(locationHash) : Charge et initialise la vue.

Paramètres	Type	Description
locationHash	string	Peux prendre trois valeurs : '#/' | '#/active' | '#/completed' Correspond à la partie de l'url après # et comprenant ce symbole.

showAll() : Récupère toutes les todos via le modèle et les affiche grâce à la vue.

showActive() : Récupère toutes les todos actives via le modèle et les affiche grâce à la vue.

showCompleted() : Récupère toutes les todos complétées via le modèle et les affiche grâce à la vue.

removeItem(id). Demande au modèle de supprimer la todo dont l'id est passé en paramètre.

Paramètres	Type	Description
id	number	Id de la todo

removeCompletedItems() : Demande au modèle de supprimer toutes les todos complétées.

addItem(title) : Demande au modèle l'ajout d'une nouvelle todo.

Paramètres	Type	Description
title	string	Valeur du titre de la todo

editItem(id) : Fait passer la vue en mode édition sur la todo correspondante récupérée par le modèle.

Paramètres	Type	Description
id	number	Id de la todo

editItemSave(id, title) : Sort du mode édition après avoir modifié une todo dont l'id et la nouvelle valeur du titre sont passés en paramètre.

Paramètres	Type	Description
id	number	Id de la todo
title	string	Valeur du titre de la todo

editItemCancel(id) : Annule le mode édition de la todo dont l'id est passé en paramètre.

Paramètres	Type	Description
id	number	Id de la todo

toggleComplete(id, completed, silent) : Demande au modèle de modifier le statut d'une todo dont l'id est passé en paramètre

Paramètres	Type	Description
id	number	Id de la todo
completed	boolean	Détermine si la todo est complétée ou non
silent	boolean | undefined	Empêche de refiltrer la liste des todos

toggleAll(completed) : Demande au modèle de modifier le statut de toutes les todos.

Paramètres	Type	Description
completed	boolean	Détermine si les todos doivent être complété

_udpateCount() : Demande au modèle de calculer le nombre total des todos, le nombre de todos actives et le nombre de todos complétées

_filter(force) : Filtre les todos selon la sélection : all / actives / complétées.

Paramètres	Type	Description
force	boolean | undefined	Détermine si il faut adapter la vue

_updateFilterState(currentPage) : Met à jour le filtre.

Paramètres	Type	Description
currentPage	string	

F. Guide pour l'audit de performance

1. **S'appuyer sur le RAIL Model Performance**
    1. Response : réponse à une interaction en moins de 50ms
    2. Animation : produire une image pour 10ms
    3. Idle : maximiser les temps de repos
    4. Load : Chargement en moins de 5 secondes

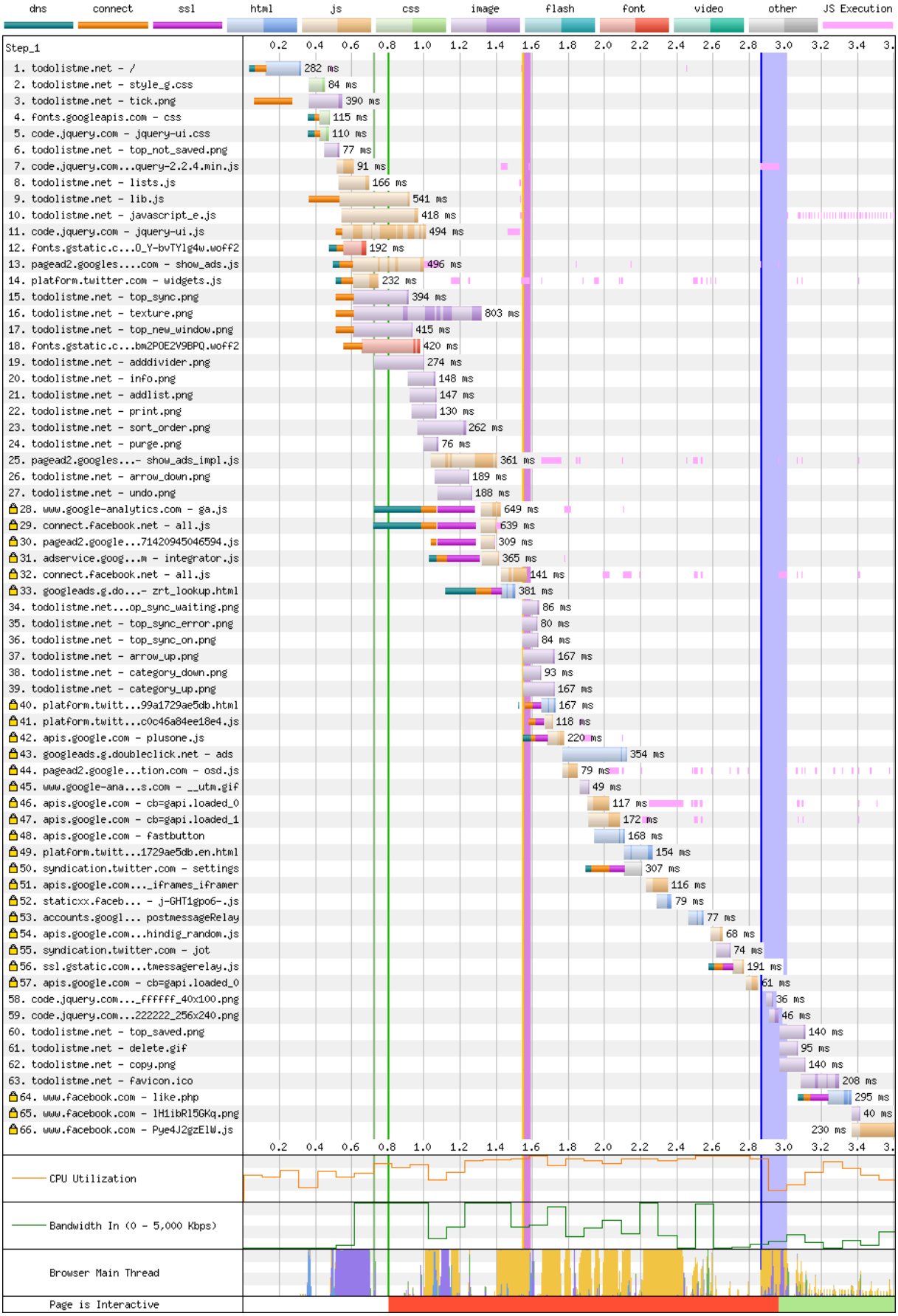
2. **Outils**
    1. Page Speed Insight
    2. Lighthouse
    3. WebPageTest
    4. DevTools
        1. Network : identifier les requêtes
        2. Performance :
    5. YellowLab
    6. RequestsMap

3. **Environnement des tests de performance**
    1. Lancer deux runs : le premier simulant une première visite, le second simulant le retour d'un visiteur
    2. Tester sur des supports différents : desktop, mobile
    3. Tester sur des supports de qualité différente
    4. Tester avec des connections différentes
    5. Tester avec des navigateurs différents
    6. Données utilisateurs vs données de lab

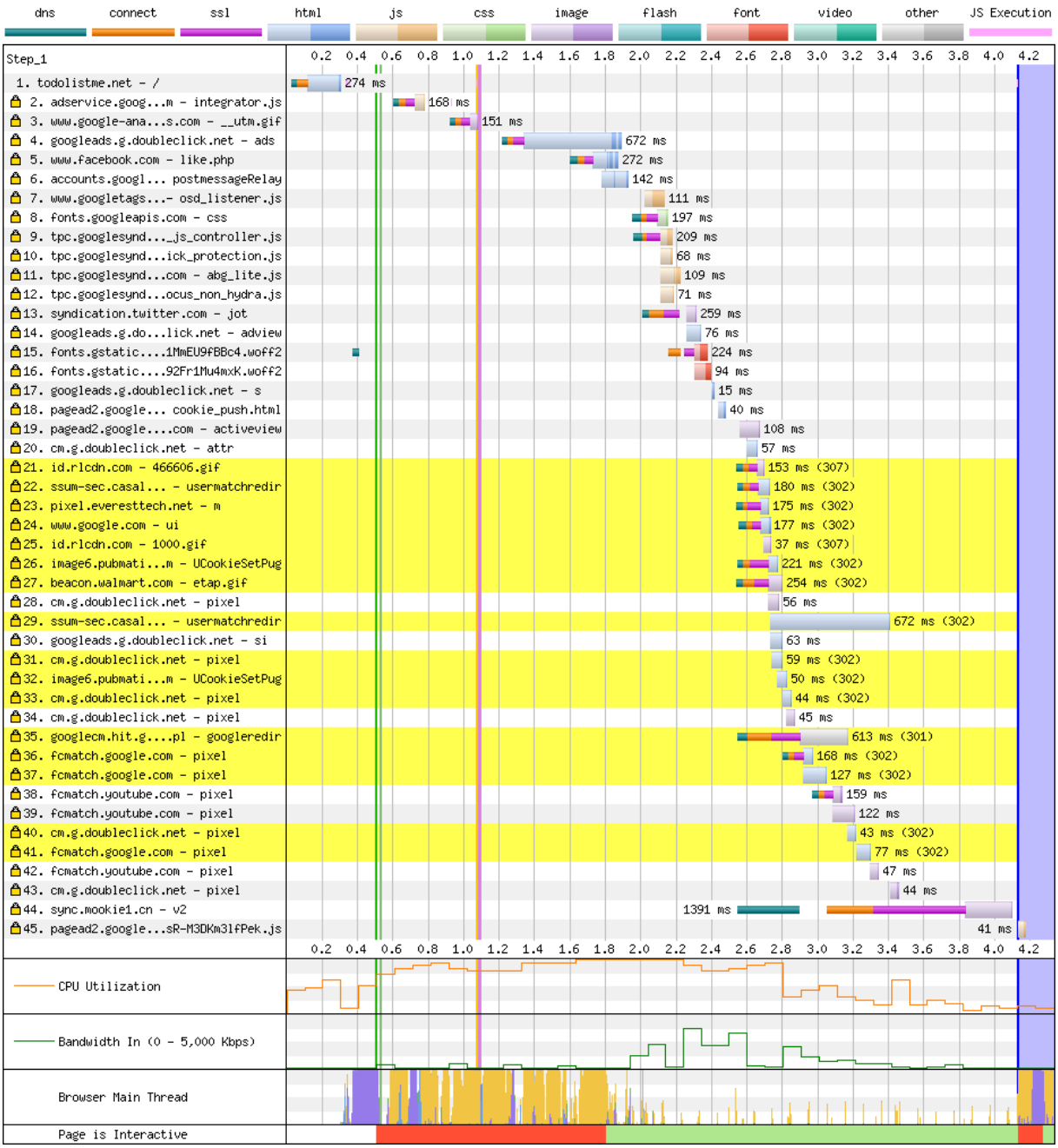
4. **Au préalable**
    1. Identifier les différentes fonctionnalités de l'application
    2. Identifier les « hero elements »
    3. Identifier les « third parties »

5. **Tests**
    1. Au chargement
        1. Qu'est-ce qu'il se passe ?
            1. First Paint
            2. First Contentful Paint
        2. Est-ce utile ?
            1. First Meaningful Paint
            2. Hero Element
        3. Est-ce utilisable ?
            1. Time To Interactive
        4. Analyser les ressources :
            1. Qu'est-ce qui est envoyé à l'utilisateur ?
            2. Comment est-ce envoyé ?
            3. Quelle est la quantité envoyé ?
        5. Analyser le critical rendering path
    2. Après le chargement
        1. L'expérience est-elle agréable pour l'utilisateur ?
            1. Long Tasks

G. WebPageTest : Waterfall 1st run / Desktop



H. WebPageTest : Waterfall 2nd run / Desktop



I. Request Map