Javascript React Software Crafter

Table of Contents

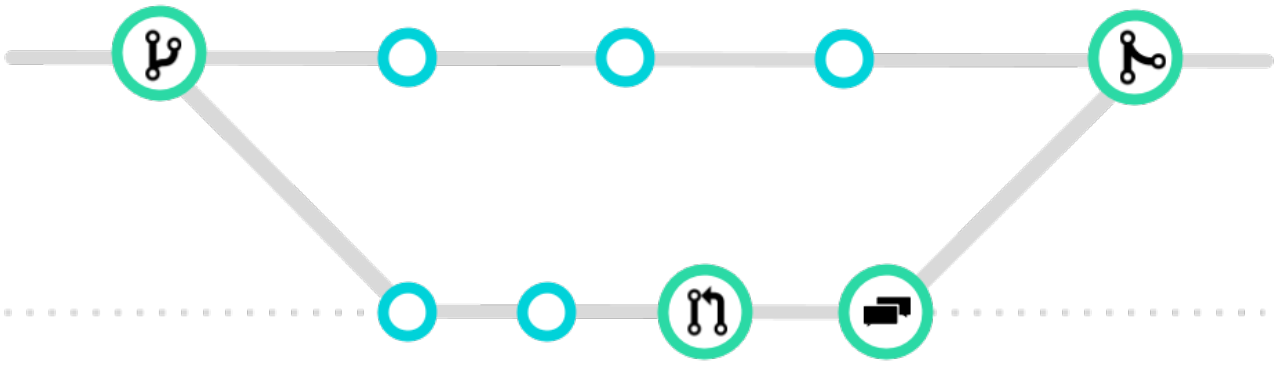
Organisation du code source blablah

GitHub Flow

Scott Chacon l'a présenté dans un post le 31 Août 2011. Il s'agit d'une version allégée du précédent et découle, en quelque sorte, du bon sens.

A noter que je garde ici la référence à GitHub par rapport au superbe post de Scott mais Atlassian (Bitbucket) le présente, par exemple, comme le workflow feature branch.

Le principe Je me suis permis d'illustrer le GitHub flow de la manière suivante :



Vous constaterez qu'il n'y a qu'une seule branche `master` et des `features` à gogo. Il est frappant de constater que le modèle est extrêmement simple et c'est le but. La courbe d'apprentissage est faible, les développeurs prennent rapidement en main ce workflow qui s'avère parfaitement adapté dans certains cas de figures.

Ce workflow se base donc sur 6 règles essentielles :

1. Tout ce qui est dans `master` peut être déployé en production La règle absolue, commune au git-flow d'ailleurs. C'est la seule branche consistante du projet et elle doit rester stable pour pouvoir se baser dessus et être déployée en production.
2. Créer des branches explicites depuis `master` (`features`) Lorsqu'on souhaite travailler sur quelque chose (une fonctionnalité, un hotfix, etc.), on crée une branche depuis `master` avec un nom explicite, si possible : bug-grunt-tests, infrastructure-ssl ou module-game-workers sont des exemples tirés d'un projet sur lequel je travaille actuellement par exemple.
3. Pousser sur `origin` régulièrement Contrairement au git-flow, où le développeur ne va pas forcément pousser sa branche `feature` locale sur le dépôt central, il s'agit ici de pousser les branches régulièrement.
   En effet, tant que nous ne sommes pas sur `master` il importe peu que la branche soit stable. En contrepartie, cela permet de communiquer avec l'équipe.
4. Ouvrir une pull-request à tout moment Si votre projet est hébergé sur GitHub ou Bitbucket, sachez que les deux proposent une fonctionnalité extrêmement intéressante pour la revue de code : les pull-request.
   Il est très facile, depuis une branche, de faire un diff avec `master` et d'ouvrir une pull-request à n'importe quel moment : que l'on pense avoir terminé ou que l'on soit coincé, la pull-request permet de demander

une revue du code. Il est alors possible de commenter le code, d'apporter des modifications et de visualiser ce que l'on s'apprête à fusionner dans `master`.

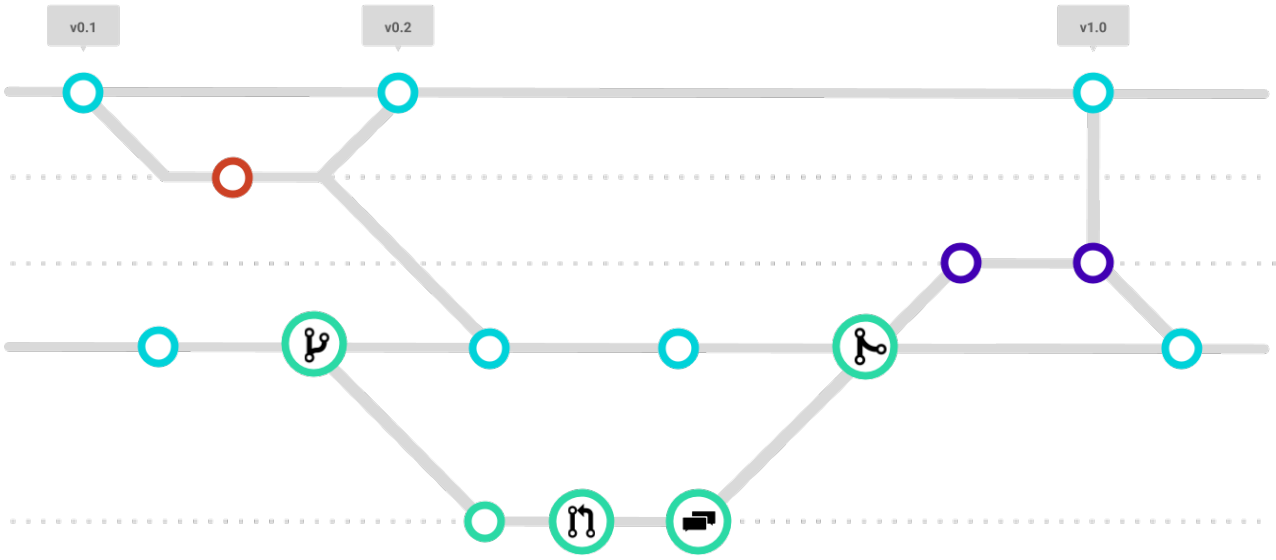
5. Fusionner seulement après une pull-request review Il s'agit plus d'un conseil que d'une règle absolue. Mais c'est la bonne pratique à mettre en place pour s'assurer que la première règle est respectée : un développeur ne doit pas fusionner sa branche dans `master` lorsqu'il pense que c'est bon, un autre doit venir faire une revue du code et confirmer la stabilité de la branche.
   Ainsi, si le merge vient mettre la pagaille sur `master`, on sait sur qui taper… 0:-)
   Vous pouvez alors :
   fusionner la branche dans `master`, directement depuis la pull-request supprimer la branche, devenue inutile, sur le serveur
   Si cela fait un moment que vous travaillez sur votre branche et que vous n'êtes plus synchro avec `master`, vous ne pourrez pas fusionner directement depuis la pull-request car il y aura des conflits. Il vous suffit de faire un merge de `master` dans votre branche et règler les potentiels conflits pour la mettre à jour.
6. Déployer immédiatement après merge dans `master` Une fois que la pull-request est validée, la branche est mergée dans `master` ce qui signifie que le tout est déployé en production (ou va être déployé sous peu).
   C'est également une façon d'accentuer la pression sur la nécessaire stabilité de `master` : le développeur ne souhaite généralement pas que le déploiement de ses modifications ne casse tout, il fera donc d'autant plus attention à la stabilité de son code avant de fusionner.
   Quelques remarques
   Ce genre de workflow s'adapte parfaitement à un projet pour lequel il n'y a pas vraiment de releases, ni de versions.
   On intègre en continue dans `master` les fonctionnalités et on déploie, parfois plusieurs fois par jour, le projet stable. On revient assez rarement (jamais) en arrière sur `master`.
   Ainsi, il est difficile d'ajouter une série de gros bugs. Si des problèmes apparaissent, ils sont rapidement résolus dans la foulée. Il n'y a pas de différence entre une grosse feature et un petit hotfix avec ce process. Peu importe la nature du changement à effectuer, le workflow est toujours le même (et donc très simple à prendre en main).

GitFlow

Ce workflow a été présenté le 5 Janvier 2010 par Vincent Driessen comme un workflow de branchements git efficace. Il couvre l'ensemble des besoins standards d'un projet de développement classique.

Le principe

Le git-flow présente le système de branches suivant :



On distingue les branches principales, fixes et immuables :

`master` est la branche où tout est stable. Chaque commit correspond à une version stable du projet (release) qui peut être déployée en production et taguée en conséquence (vX.Y.Z).

`develop` est la branche sur laquelle s'effectue le développement proprement dit. On y prépare les changements en vue de la prochaine release dans `master`.

Puis les branches secondaires qui se font et se défont avec le temps :

`feature` part de `develop` et se merge dans `develop`.

On crée une branche `feature/xxx` lorsque l'on travaille sur une fonctionnalité en particulier. Lorsqu'elle est terminée, on la merge dans `develop` pour ajouter la `feature` stable dans le scope de la prochaine release.

`release` part de `develop` et se merge dans `master` et `develop`.

On crée une branche `release/xxx` à partir de `develop` lorsque celle-ci reflète l'état désiré de la release (l'ensemble des fonctionnalités du scope ont été mergées). Ainsi, on peut préparer la prochaine release tranquillement, corriger d'éventuels bugs et poursuivre le développement en parallèle.

Une fois que la release est prête (stable) on merge alors la branche dans `master`, mais aussi dans `develop` pour mettre à jour les modifications apportées.

`hotfix` part de `master` et se merge dans `master` et `develop` / `release`.

On crée une branche `hotfix/xxx` lorsque l'on veut résoudre un bug critique en production rapidement. C'est un peu comme une release non planifiée.

Lorsque le correctif est développé, on le merge dans `master` avec le numéro de version qui convient, ainsi que dans `develop` (ou la branche `release` en cours, le cas échéant) pour mettre à jour les modifications apportées.

Tester en Javascript

Kata

Un kata de code est un exercice de programmation qui permet aux programmeuses et aux programmeurs de perfectionner leurs compétences à travers la pratique et la répétition

TDD

Le Test-Driven Development (TDD), ou développements pilotés par les tests en français, est une méthode de développement de logiciel qui consiste à écrire chaque test, notamment des tests unitaires, avant d'écrire le code source d'un logiciel, de façon itérative.

Le processus standard du TDD est :

Red Green Refactor

Red

Ecrire un test qui repond au attente du developpeur mais qui test un cas non pris en compte par le code d'implementation.

Cela peut aussi etre un test qui appelle du code qui n'existe pas encore (et qui fail a cause de ca)

Green

Ecrire le code minimum (naïf) qui repond à l'ensemble des tests.

Il est important de tester ce nouveau code face a l'ensemble des tests et pas seulement au test qui vient d'etre ecrit pour se premunir de non regression sur d'autre fonctionnalité.

Refactor

Se poser la question : "est ce que le code est satisfaisant - test et implementation"

C'est a cette etape que l'implementation naive est remplacé par du code plus robuste et plus modulaire.

Il n'est pas rare de rajouter des tests suite a une decision prise dans la phase de refactoring.

Tester en React

Enzyme

Enzyme est une librairie creer par AirBnB pour tester des composants React.

Vous pouvez trouver des exemples de l'utilisation de Enzyme dans le projet `blink` de ce Repository

Mock et Spy avec Jest

Les mocks et les spy sont indispensable mais peuvent prendre du temps si ils sont mal utilisé.

Il est important de bien comprendre que l'on doit donner des mocks a utiliser a nos fonctions et pas mocker les comportements.

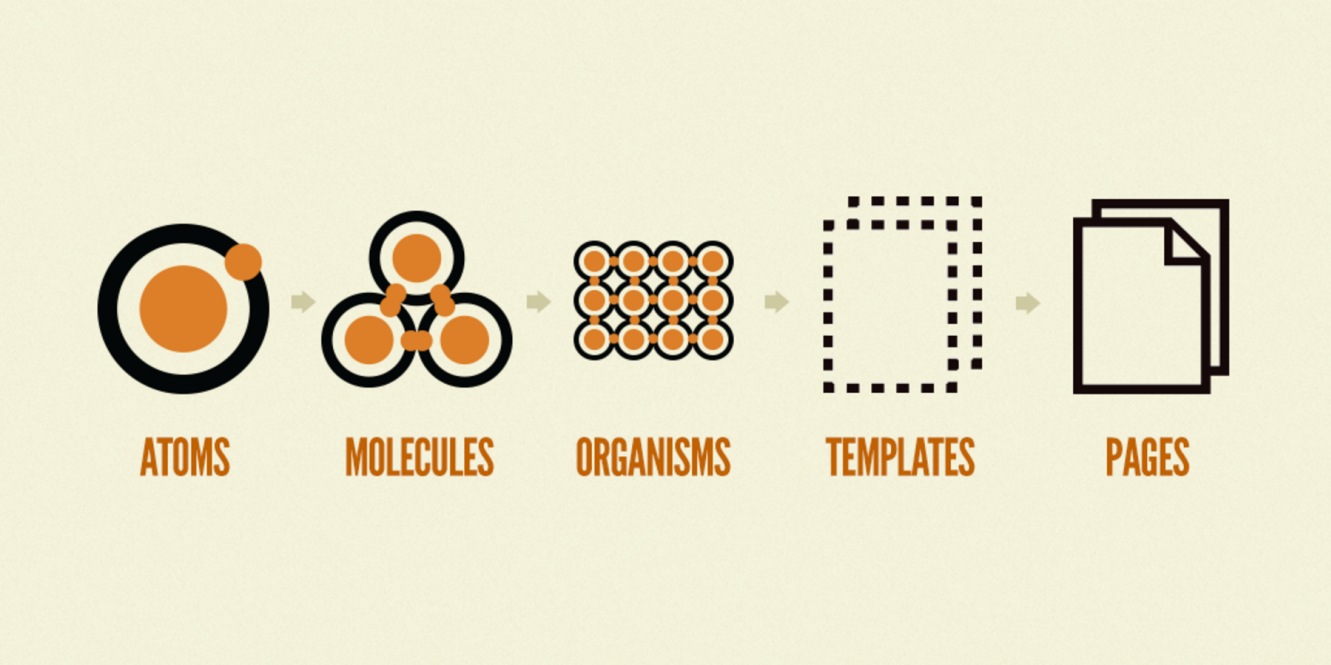
Le test ne doit jamais reproduire de comportement ou de logic annexe a la fonctionnalité testée.

Il est possible de créer des Spy qui permette de savoir si une fonctionnalité tierce a ete appelée mais en gardant la logic en dehors du spy.

Organiser son code en React

Atomic Design à la rescousse

l'atomic design est une methode pour decouper et organiser ses composants React



Pour plus d'information sur atomic design vous pouvez le faire directement sur leur site : http://atomicdesign.bradfrost.com/table-of-contents/

Il existe un exemple de code montrant le decoupage avec atomic design dans le repertoire `atomic`

Points important :

- le fichier .env a la racine du projet change le path par defaut pour etre a l'interieur du repertoire `src` ce qui s'implifie les path d'import
- le repertoire component contient un fichier index.js qui s'implifie encore l'import des composants.
- chaque composant est placé dans son propre repertoire a l'interieur d'un des repertoires atoms - molecules - organisms - templates - pages

- le nom du repertoire du composant sera celui exposé au niveau de components

React et Redux tips and tricks

Dumb and Smart component

L'approche composant de React permet d'identifier 2 types de composants:

- dumb : qui gere l'affichage, "le rendu HTML"
- smart : qui ne gere que la logic et les evenements venant du HTML

State Management

Utiliser Redux pour gérer un etat global à l'application.

Il y a 2 exemples de l'utilisation de redux dans le repository `minimal-redux` et `minimal-redux-thunk` les 2 exemples sont constuit avec tous les composants dans le meme fichier (src/index.js)

redux permet de gerer l'etat de l'applicatoin thunk permet d'ameliorer redux pour que redux gère les actions asynchrone, comme une requete a une API ou un reauete a indexeddb - le 2eme exemple de code concerne uniquement l'utilisation des actions asynchrones.

Thunk fonctionne de la maniere suivante : - les actions thunk sont asynchrones, elles prennent en parametres le state et la fonction dispath - thunk va traiter les actions par leur type,
si ce sont des objets (action synchrone) il laisse passer l'action au reducer sans rien faire,
si le type est un fonction, il passe a la fonction 2 function (getState et dispatch) le role de l'action est d'executer sa fonction asynchrone, et de dispatcher une action synchrome quand le callback ou la promesse est resolu.