* Travail d’integration : camera, environement
* **Présentation des livrables**: Vous présenterez votre travail à votre évaluateur en expliquant vos choix techniques. (10 minutes)

J’avais reçu une proposition en tant que développeur web pour un stage chez Reservia- une petite entreprise proposant un outil de planification de vacances. L’idée c’était de transformez une maquette en site web.

Le site va permettre aux usagers de trouver des hébergements et des activités dans la ville de leur choix. Les hébergements peuvent également être filtrés par thématique, par exemple leur budget ou leur ambiance.

* A l’étape d’intégration de la maquette responsive en HTML et CSS. J’avais reçu l’email avec des les nouvelles [maquettes desktop et mobiles du site Reservia](https://s3-eu-west-1.amazonaws.com/course.oc-static.com/projects/Front-End+V2/P2+HTML+%26+CSS/Projet+2+-+Reservia+FR.zip). Le dossier avait aussi les images. Moi, j’avais intégré dans mon projet les image s avec la plus petite taille, j’avais choisi le format le plus adapté par rapport à la résolution et au temps de chargement. (le projet a 46 MegaBytes).
* Comme environnement de travail j’utilise Visual Studio Code (plusieurs plugins pourront te simplifier la vie, par exemple Live Server ou Prettier.)

**Fonctionnalités**

* Les usagers pourront rechercher des hébergements dans la ville de leur choix. Le champ de recherche est donc un champ de saisie, dont le texte peut être édité par l’usager. En revanche, le bouton de recherche ne reste pas fonctionnel.
* Chaque carte d’hébergement ou d’activité devra être cliquable dans son intégralité, mais les liens seront vides.
* Les filtres ne sont pas fonctionnels, mais, il faut qu’ils changent d’apparence au survol.
* Dans le menu, les liens “Hébergements” et “Activités” sont des ancres qui mènent aux sections de la page.

**Contraintes techniques**

* J’avais reçu deux maquettes : l’une desktop et l’autre mobile. Le site devra également être adapté aux tablettes. Resolution 1080 px
* Les icônes proviennent de la bibliothèque [Font Awesome](https://fontawesome.com/).
* J’avais recu 3 couleur de base : bleu claire, bleu , gris pour le fond . Les couleurs de la charte sont le bleu #0065FC, et sa version plus claire #DEEBFF ainsi que le gris pour le fond #F2F2F2.
* La police du site est [Raleway](https://fonts.google.com/specimen/Raleway).

Pendant la réalisation de ce projet j’avais tester mon code via au validateur W3C pour m’assurer que j’utilise les bonnes balises sémantiques et pour m’assurer qu’il as pas des erreurs dans mon code HTML et CSS. En ce qui consiste le versioning de mon code, j’avais comité vers mon hébergeur GIT le code

Description du code : body-out-foot + footer

Body-out-foot : top (jusqu’au hébergement) + main(la partie centrale) + section (activités a Marseille)

* L’objectif du GIT est de produire un historique clair. Ça facilite le travail en équipe, ça optimise le travail, ça gagne en qualité.

**Discussion** : votre évaluateur, jouant toujours le rôle de votre manager, va vous questionner sur la façon dont vous avez intégré les éléments ainsi que sur votre utilisation de Git et GitHub. Comme vous venez d’arriver dans l’équipe, il est important que vous vous familiarisiez avec les pratiques de l’équipe ! (15 minutes)

**Git** est un [logiciel de gestion de versions](https://fr.wikipedia.org/wiki/Logiciel_de_gestion_de_versions) [décentralisé](https://fr.wikipedia.org/wiki/Gestion_de_versions#Gestion_de_versions_d.C3.A9centralis.C3.A9e). Le code informatique développé est stocké non seulement sur l’ordinateur de chaque contributeur du projet, mais il peut également l'être sur un serveur.

Git possède deux [structures de données](https://fr.wikipedia.org/wiki/Structure_de_donn%C3%A9es) : une base d'objets et un cache de [répertoires](https://fr.wikipedia.org/wiki/R%C3%A9pertoire_(informatique)). Il existe quatre types d'objets :

* l'objet **blob** (pour [binary large object](https://fr.wikipedia.org/wiki/Binary_large_object" \o "Binary large object) désignant un ensemble de données brutes[7](https://fr.wikipedia.org/wiki/Git#cite_note-:0-7)), qui représente le contenu d'un [fichier](https://fr.wikipedia.org/wiki/Fichier_informatique) ;
* l'objet **tree** (mot anglais signifiant arbre), qui décrit une arborescence de fichiers. Il est constitué d'une liste d'objets de type *blob*s et des informations qui leur sont associées, tel que le nom du fichier et les [permissions](https://fr.wikipedia.org/wiki/Droits_sur_un_fichier). Il peut contenir [récursivement](https://fr.wikipedia.org/wiki/Type_r%C3%A9cursif) d'autres *trees* pour représenter les sous-répertoires ;
* l'objet **commit** (résultat de l'opération du même nom signifiant « valider une transaction »[8](https://fr.wikipedia.org/wiki/Git#cite_note-8)), qui correspond à une arborescence de fichiers (*tree*) enrichie de métadonnées comme un message de description, le nom de l'auteur, etc. Il pointe également vers un ou plusieurs objets *commit* parents pour former un graphe d'historiques[7](https://fr.wikipedia.org/wiki/Git#cite_note-:0-7) ;
* l'objet **tag** (étiquette) qui est une manière de nommer arbitrairement un commit spécifique pour l'identifier plus facilement. Il est en général utilisé pour marquer certains commits, par exemple par un numéro ou un nom de version (2.1 ou bien [Lucid Lynx](https://fr.wikipedia.org/wiki/Liste_des_versions_d%27Ubuntu" \l "Ubuntu_10.04_LTS_.28Lucid_Lynx.29" \o "Liste des versions d'Ubuntu)).

Git dispose notamment des commandes suivantes :

* git init​ crée un nouveau dépôt ;
* git clone​ clone un dépôt distant ;
* git add​ ajoute de nouveaux objets *blob*s dans la base des objets pour chaque fichier modifié depuis le dernier *commit*. Les objets précédents restent inchangés ;
* git commit​ intègre la somme de contrôle [SHA-1](https://fr.wikipedia.org/wiki/SHA-1) d'un objet *tree* et les sommes de contrôle des objets *commit*s parents pour créer un nouvel objet *commit* ;
* git branch​ liste les branches ;
* git merge​ fusionne une branche dans une autre ;
* git rebase​ déplace les commits de la branche courante devant les nouveaux commits d’une autre branche ;
* git log​ affiche la liste des commits effectués sur une branche ;
* git push​ publie les nouvelles révisions sur le *remote*. (La commande prend différents paramètres) ;
* git pull​ récupère les dernières modifications distantes du projet (depuis le *Remote*) et les fusionne dans la branche courante ;
* git stash​ stocke de côté un état non commité afin d’effectuer d’autres tâches.
* [GitHub](https://fr.wikipedia.org/wiki/GitHub) est un service web d'hébergement et de gestion de développement de logiciels développé en [Ruby on Rails](https://fr.wikipedia.org/wiki/Ruby_on_Rails) et [Erlang](https://fr.wikipedia.org/wiki/Erlang_(langage)), qui appartient à [Microsoft](https://fr.wikipedia.org/wiki/Microsoft) depuis juin 2018 ;

[GitHub](https://github.com/) est un site de partage de code, sur lequel on peut publier des projets dont le code est géré avec le système de gestion de version [Git](http://git-scm.com/)**.**

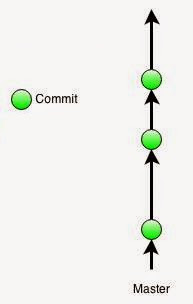
git remote add origin https://github.com/chrisdavidmills/mon-premier-depot.git

1. git add --all
2. git commit -m 'ajout des fichiers au dépôt' //envoyez le code sur GitHub
3. git push -u origin master

Le git-flow est probablement le plus connu, voire le seul d’ailleurs.  
Il s’agit d’un modèle de branches standard qui semble s’adapter à n’importe quel projet, pas trop complexe à prendre en main, avec quelques subtilités cependant.

* Il n'existe pas de modèle unique de workflow Git adapté à toutes les circonstances. Un workflow doit être simple et améliorer la productivité de votre équipe.

#### Master-only

Pour des projets simples, avec peu de contributeurs, l'utilisation de la seule branche 'master' peut convenir.

Il y a plusieurs implications :  
- une seule personne travaille sur une évolution donnée  
- il faut répartir le travail afin de limiter au maximum les risques de conflits  
- la durée d'écriture d'une évolution doit être courte, là aussi pour éviter les risques de conflits

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)
* develop est la branche sur laquelle s’effectue le développement proprement dit.

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

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

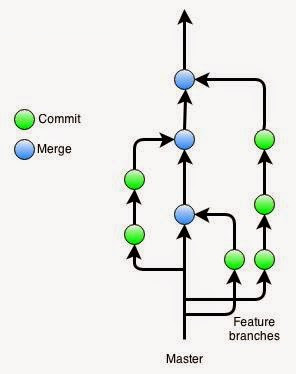
On crée une branche **release/xxx** à partir de develop lorsque celle-ci reflète l’état désiré de la release (l’ensemble des fonctionnalités du scope ont été mergées).

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.

La création de branches de fonctionnalité est une extension logique du workflow centralisé. Le principe de base du [workflow de branche par fonctionnalité](http://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow) est que chaque fonctionnalité est développée dans une branche dédiée plutôt que dans la branche master. Cela signifie également que la branche master ne contiendra jamais de code bogué : un avantage non négligeable pour les environnements d'intégration continue.

**Discussion** : votre évaluateur, jouant toujours le rôle de votre manager, va vous questionner sur la façon dont vous avez intégré les éléments ainsi que sur votre utilisation de Git et GitHub. Comme vous venez d’arriver dans l’équipe, il est important que vous vous familiarisiez avec les pratiques de l’équipe ! (15 minutes)

#### GitHub-flow



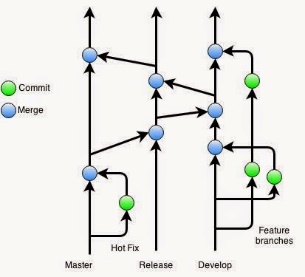
Le schéma précédent présente un inconvénient certain pour des développements longs, et ne permet pas de collaborer sur une évolution, ne serait-ce que pour permettre à un responsable de projet de valider des modifications avant de les intégrer.

Il ne permet pas non plus de laisser temporairement de côté une évolution pour travailler sur une fonctionnalité plus urgente à intégrer (il y a bien sûr la possibilité de faire du remisage, mais nous parlons ici de principes de méthode de travail).

Un workflow plus courant consiste à utiliser des 'feature branches'.  
Il se base sur la méthode suivante :  
- à tout moment la version disponible sur la branche 'master' est stable et donc déployable  
- une branche est créée à partir de 'master' pour toute modification, quelle que soit son ampleur  
- régulièrement cette branche est poussée sur une branche du même nom sur le serveur  
- quand nécessaire, un responsable du projet valide les modifications  
- la branche est fusionnée sur 'master' et est poussée sur le serveur

Cette méthode a pris le nom de 'github-flow' car c'est celle utilisée par les développeurs du service GitHub, et celle qui est naturellement mise en oeuvre lorsque l'on utilise un service tel que GitHub offrant les fonctions de fork/pull request.

#### Gitflow



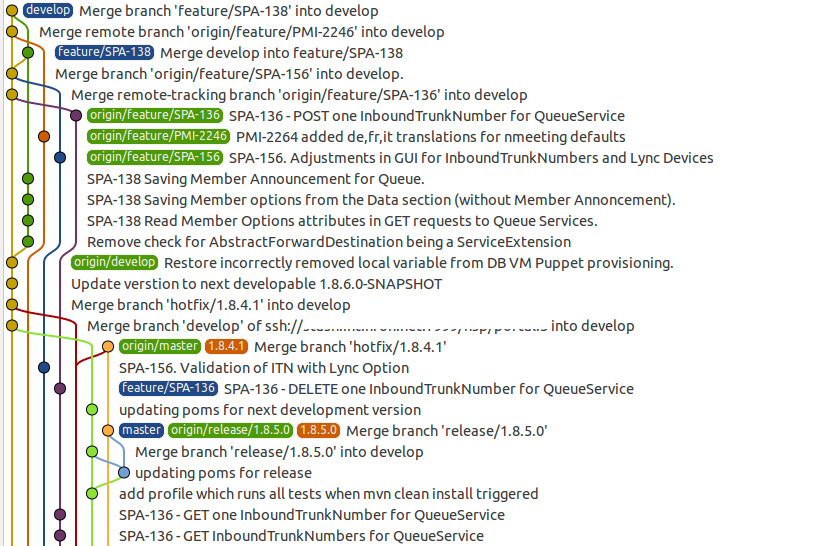
L'inconvénient, pour certains projets, du schéma précédent est la présence sur la branche 'master' de micro-évolutions. Il peut être préférable de ne déposer sur 'master' que des versions livrables complètes (des releases).

Un schéma plus complexe est dans ce cas nécessaire. Le workflow portant le nom de **gitflow** a été proposé en ce sens par [Vincent Driessen](http://nvie.com/posts/a-successful-git-branching-model/). Il contient :  
- une branche 'develop' qui correspond à la branche 'master' du 'github-flow'  
- les releases sont préparées sur une branche spécifique (fusion depuis la branche 'develop' jusqu'à ce que la release soit validée)  
- lorsqu'une release est prête, elle est fusionnée sur la branche 'master'

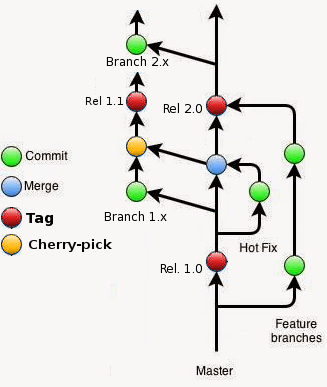
La correction d'un bug critique est fusionnée directement sur la branche 'master' et déployée sur les branches 'release' et 'develop'.

Ce workflow est utilisé par de nombreux projets (d'une certaine ampleur...) et a donné lieu au développement d'une extension Git pour faciliter sa mise en oeuvre (voir <https://github.com/nvie/gitflow>).

Si il a été largement adopté, ce workflow a également ses détracteurs, à cause de la complexité de l'historique qu'il génère (ci-dessous). Chacun se fera son avis...



#### Workflow intermédiaire



De nombreux projets adoptent un workflow de complexité intermédiaire, conservant la simplicité du GitHub-flow tout en prenant en compte la gestion des releases :  
- la branche 'master' contient la version (stable !) en cours de développement  
- l'ajout/modification de fonctionnalités et la correction de bug sont réalisées sur des branches spécifiques  
- lorsque la version en cours peut-être diffusée (création d'une release), une étiquette est apposée sur la branche 'master' et une 'release-branch' est créée.