La gestion de version

Un système de gestion de versions (VCS en anglais pour *Version Control System* ou aussi SCM pour *Source Content Management*), ou encore système de versionning, est un système qui enregistre toutes les modifications apportées à une liste de fichiers. C'est un système qui permet de suivre précisément l'évolution du contenu des fichiers. Ce type de système est très utilisé en développement informatique, mais il n'y est pas limité puisque n'importe quelle activité utilisant des fichiers lisibles peut être suivie par un VCS. Par exemple, les systèmes Wiki (comme Wikipédia) utilisent un VCS. Il est également possible de citer le très connu site d'entraide informatique Stack Overflow qui suit les versions des questions et des réponses.

Le suivi des fichiers est effectué par sauvegarde des modifications. En effet, lorsque vous avez fini de modifier un fichier, vous allez indiquer au VCS que vous avez terminé ce travail et pour quelles raisons vous l'avez fait. Les modifications effectuées sur le fichier seront sauvegardées par le système.

Il ajoutera alors une nouvelle ligne (qu'on appellera révision) dans votre historique. Le terme révision (appelé également *commit* en anglais) définit un ensemble de modifications qui sont enregistrées par le système.

Les intérêts d'un tel système sont multiples et vont bien au-delà du suivi pur et simple d'un projet

Les intérêts de la gestion de version

1. Une véritable machine à remonter le temps

Imaginez une petite situation habituelle : vous êtes un indépendant et vous commencez votre semaine. Lundi matin à 9h, un de vos clients vous appelle : "Vous deviez modifier mon site pour enregistrer tout ce que faisaient les utilisateurs du site". Vous vous empressez de vous excuser et vous modifiez le code pour ajouter la fonctionnalité le plus rapidement possible. Une fois correctement réveillé - soit une heure et deux cafés plus tard -, vous vous rendez compte que vous n'êtes pas du tout parti dans la bonne direction. Le code que vous avez produit ne servira à rien et complexifie l'application alors qu'il y avait une manière beaucoup plus simple de faire. Votre VCS vous permet de revenir très simplement en arrière comme si ce début de lundi n'avait pas existé. Vous repartez donc avec un répertoire de travail propre.

Mais votre VCS va encore plus loin puisqu'il peut vous permettre de faire un retour en arrière de plusieurs années sans problème pourvu que vous l'ayez utilisé à ce moment-là.

Un système de versionning permet donc de revenir très facilement en arrière et à n'importe quel endroit de l'historique.

2. Une documentation détaillée et datée

Dans une équipe, lorsqu'on ajoute une fonctionnalité à un logiciel, il arrive régulièrement qu'on soit bloqué par une ou plusieurs lignes qu'on ne comprend pas. Le code est peu documenté et n'est pas du tout clair. Sans versionning, un développeur pourrait passer plusieurs heures à comprendre le code et à le documenter. Avec un VCS, vous pouvez obtenir différentes informations concernant les lignes de code. Les messages que les développeurs indiquent lorsqu'ils effectuent des modifications sont importants pour les prochains développeurs qui travailleront sur le code, c'est la raison pour laquelle ces messages doivent être rédigés avec soin et clarté. Ce message est communiqué par l'auteur du code où il explique les modifications qu'il a effectuées.

Le message permet d'avoir une idée de ce que le développeur a voulu réaliser dans son code. Par exemple, un message tel que "Prise en compte des remises dans le calcul de la TVA des factures" peut aider à comprendre un code.

Si le code reste encore obscur après la lecture du message, il est aussi possible d'aller voir son auteur si celui-ci est toujours dans l'équipe.

Un système de gestion de versions permet donc de garder une documentation à propos de chaque modification effectuée. Cela permet de simplifier la maintenance des applications et l'intégration de nouveaux collaborateurs.

3. Une pierre de Rosette pour collaborer

Prenons le cas où deux développeurs (Pierre et Paul) d'une équipe travaillent sur le même projet. Ils sont partis de la même version du projet et ont effectué leurs modifications dans leur propre répertoire de travail. Nous avons donc à un instant T deux versions différentes du projet.

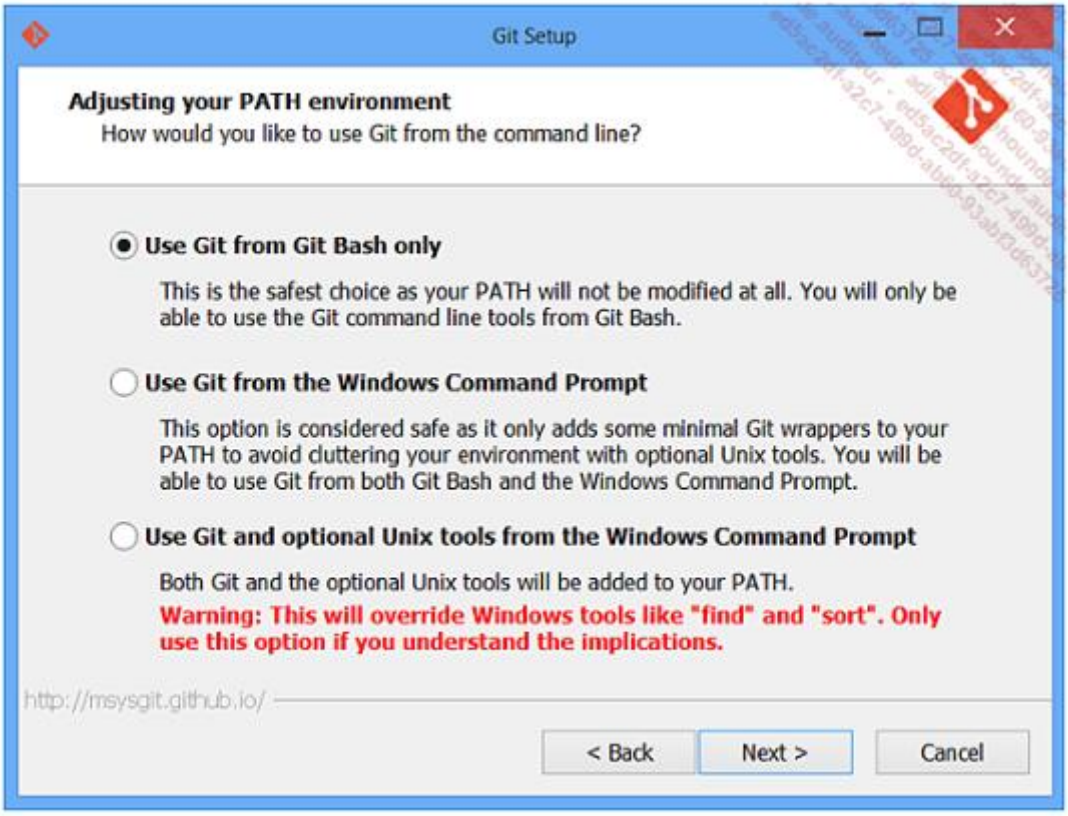
Grâce au système de gestion de version, la fusion de ces deux versions sera facile à effectuer, surtout s'ils ont travaillé sur des fichiers différents. La facilité de collaboration entre développeurs grâce au VCS est d'autant plus avérée lorsqu'on utilise des systèmes de gestion de versions décentralisés.

Installation sous Windows

Git n'a pas été conçu à l'origine pour fonctionner sur Windows. Pour utiliser Git efficacement sous Windows il faut l'utiliser au travers d'une interface en ligne de commande simulant un environnement UNIX. Cette interface se nomme Cygwin, elle est développée par RedHat et fournie directement avec l'installateur de Git.

Pour télécharger l'exécutable d'installation, il faut aller sur le site officiel : https://git-scm.com. Là encore, à l'aide des informations envoyées par le navigateur, git-scm.com nous propose de télécharger l'exécutable qui correspond à notre système :

En double cliquant ensuite sur l'exécutable, la fenêtre d'installation s'affiche. Il faut suivre les étapes par défaut. À un moment, une fenêtre intitulée **Adjusting your PATH environment** se présente comme ci-dessous :



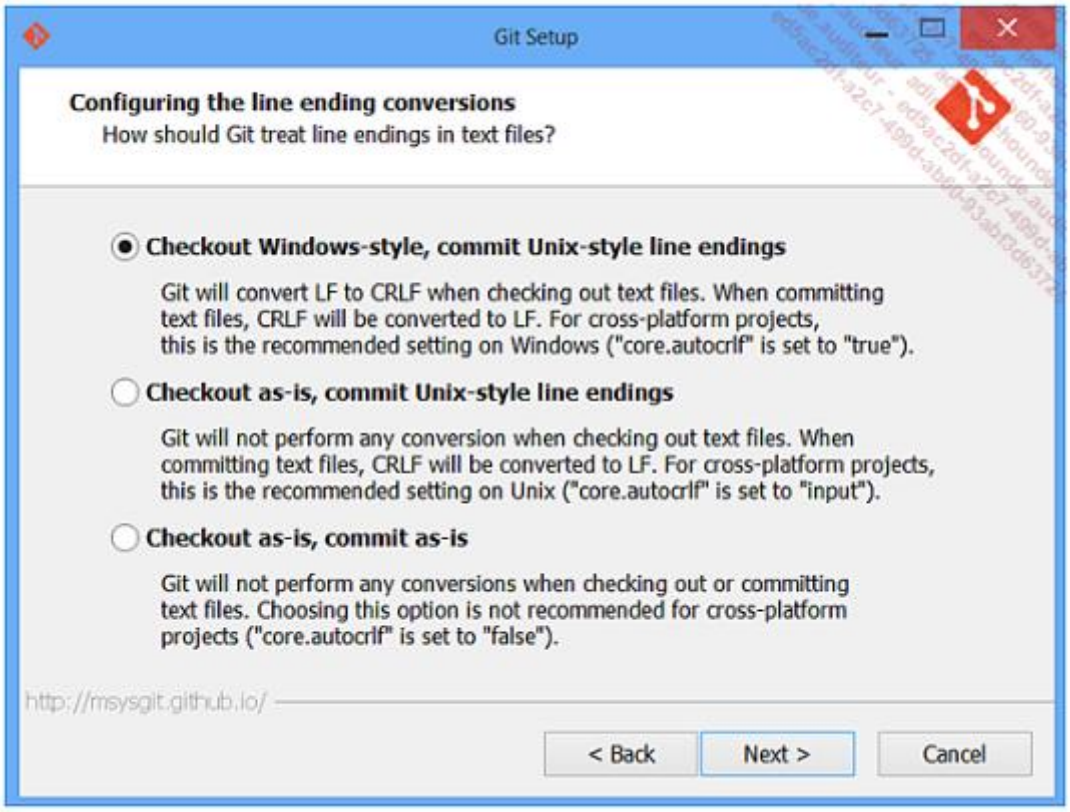
Cette fenêtre permet d'indiquer à Git que l'utilisateur utilisera Cygwin pour appeler Git à partir d'un mode *bash*. Sur cette fenêtre il faut également laisser le choix par défaut car Cygwin fournit des outils très pratiques pour utiliser Git.

Ensuite l'exécutable affiche une fenêtre permettant de configurer la gestion des sauts de ligne (capture ci-dessous). Il faut savoir que les systèmes UNIX et les systèmes Windows n'utilisent pas les mêmes caractères pour effectuer un saut de ligne. En effet, Windows utilise le mode CRLF (*Carriage Return Line Feed*), c'est-à-dire que dans un fichier de texte brut, Windows insère entre deux lignes un retour chariot puis un saut de ligne. Les

systèmes UNIX utilisent le mode LF, c'est-à-dire qu'ils utilisent uniquement un saut de ligne entre deux lignes.

Git permet d'effectuer certaines actions sur les sauts de ligne. En effet, si un utilisateur travaillant sur UNIX crée un fichier et l'ajoute au dépôt, puis qu'un utilisateur Windows ouvre le fichier, le fichier sera perçu comme ayant été entièrement modifié étant donné que chaque saut de ligne aura été converti par Windows. Git va donc proposer plusieurs modes pour permettre aux différents systèmes d'être interopérables avec Git.

- Le premier mode est le mode défini par défaut. Lorsque Git enregistrera des fichiers sur votre système Windows il convertira les sauts de ligne UNIX (LF) en sauts de ligne Windows (CRLF). Git fera l'inverse lorsqu'il enregistrera le contenu des fichiers dans le dépôt. C'est le mode recommandé pour les utilisateurs de Windows.
- Dans le deuxième mode, Git ne modifie pas les sauts de ligne des fichiers qu'il enregistre sur le système de fichiers. Inversement, lorsque l'utilisateur commitera des fichiers sur le dépôt, les sauts de ligne de type Windows seront convertis en sauts de ligne UNIX.
- Le dernier mode n'effectue aucune conversion de saut de ligne. Normalement, si tous les utilisateurs de Git utilisent le même système d'exploitation ce type ne pose pas de problème particulier. C'est tout de même se priver de nombreuses possibilités que de restreindre l'utilisation du dépôt à un seul type de système. C'est la raison pour laquelle cette option est fortement déconseillée lorsqu'on gère un projet développé sur des systèmes Windows et Unix.



Une fois que Git a été installé, deux raccourcis ont été créés dans la liste des programmes. Un raccourci appelé **Git Bash** redirige vers une interface en ligne de commande gérée par Cygwin. Un autre appelé **Git GUI** redirige vers une interface permettant d'utiliser Git.

Pour toute la suite de ce livre, il faudra utiliser **Git Bash** qui permet d'effectuer beaucoup plus d'actions et qui permet également de mieux comprendre le fonctionnement de Git.

Après avoir cliqué sur le raccourci **Git Bash**, une interface en ligne de commande s'affiche. Cette interface permet d'utiliser Git ainsi qu'un nombre important de commandes bash.

Par exemple, lorsque l'on utilise la commande pwd, qui sous UNIX permet d'obtenir le chemin courant, l'interface affiche la sortie suivante :

```
/c/Users/adimi
```

Le chemin est converti automatiquement en chemin de type UNIX avec des slashs au lieu des antislashs. Par exemple, pour changer de disque, il faut utiliser la commande :

```
cd /e
```

au lieu de la commande :

```
cd E:
```

Le site officiel du projet contient une documentation plus précise des fonctionnalités incluses dans Cygwin : https://cygwin.com

Pour vérifier l'installation de Git il est possible d'utiliser la commande suivante :

```
git --version
```

Cette commande affiche la sortie suivante :

```
git version 1.9.5.msysgit.1
```

Cette version n'est pas très récente. Il existe des versions plus à jour de Git sur des dépôts non officiels. L'installation est très simple et similaire à celle que nous venons d'effectuer. Le dépôt git-for-windows contient de nombreuses versions de Git pour Windows et propose même des versions portables (ne nécessitant pas d'installation). Les versions de Git installables sont disponibles sur l'adresse suivante : https://github.com/git-for-windows/git/releases

CONFIGURATION

Configurer les informations de l'utilisateur pour le dépôt courant ou pour tous avec l'option --global

git config --global user.name "[nom]"

Définit le nom de l'utilisateur

git config --global user.email [email]

Définit l'email de l'utilisateur

Cas pratique d'utilisation

Ce chapitre a pour but d'expliquer comment utiliser Git à l'aide d'un scénario. En effet, pour bien apprendre à utiliser Git, il ne s'agit pas forcément de connaître toutes ses commandes sur le bout des doigts : il faut surtout connaître les concepts et savoir à quel moment il faut exécuter une action précise.

Ce scénario est simplifié et concret pour expliquer le fonctionnement de Git. Ce chapitre sera donc à la fois théorique et pratique.

À la fin de ce chapitre, vous aurez appris à utiliser les principales commandes de Git avec le service en ligne Bitbucket. Vous serez autonome dans votre utilisation de Git.

Contexte du scénario

Raphaël est développeur web indépendant. Ses prestations principales sont le développement de sites web ou de sites intranet spécifiques à l'activité du client. Une fois qu'une prestation est réalisée, il assure la maintenance de son application. Il utilise depuis plusieurs années Git qui lui permet :

- de suivre ses développements et leur évolution,
- de collaborer avec des salariés du client,
- de proposer à ses clients un service mieux fini avec un historique complet.

Ses clients apprécient ce service notamment parce qu'il leur permettrait de changer de prestataire plus facilement si besoin, car la compréhension du développement est ainsi plus simple.

Cela permet de montrer à ses clients qu'il a confiance en son travail et qu'il ne cherche pas à les empêcher d'aller voir la concurrence.

Aujourd'hui, Raphaël vient de conclure un contrat pour un nouveau client : M. Airadun. Il a reçu un cahier des charges qui va lui permettre de commencer à travailler dessus. Pour les besoins de notre exemple, ce contrat va être extrêmement simple puisqu'il va consister en la création d'une liste de dates d'anniversaire. En effet, le client spécifie dans le cahier des charges :

"J'en ai marre ! Je n'arrive jamais à me souvenir de l'anniversaire de ma femme et de mes enfants !"

Raphaël a donc décidé de commencer sans plus attendre le développement pour son client. Il a l'habitude d'utiliser un service en ligne lorsque le client accepte de voir son projet stocké par un tiers. Ce service en ligne s'appelle Bitbucket et sera abordé dans la suite de ce chapitre.

Création du dépôt

Après avoir travaillé sur la conception de son application, Raphaël est prêt pour démarrer le développement réel de celle-ci. Il va donc commencer par créer un dépôt local sur son poste. Il ouvre une interface de ligne de commande et se place dans le dossier *AnniversaireListe* et exécute la commande suivante :

```
git init
```

Cette commande va créer un nouveau dépôt dans le dossier courant. Cela signifie que Raphaël va maintenant pouvoir utiliser Git, et donc les commandes Git, à l'intérieur de ce répertoire.

Techniquement, lorsqu'on exécute git init, Git crée un dossier *.git* dans le dossier courant. Ce dossier caché va contenir tout le contenu du dépôt. Il faut tout de même savoir que c'est uniquement Git qui doit gérer les fichiers de ce dossier. Une simple modification dans ce dossier pourrait corrompre le dépôt et l'empêcher de fonctionner correctement.

Pour l'instant, Raphaël n'a créé aucun fichier, donc il n'a rien à suivre. Il va donc pouvoir commencer son développement.

Début du développement

Pour l'instant, Raphaël n'a pas besoin de suivre des fichiers, car il n'en a pas encore. Il va donc commencer le développement en créant un fichier HTML intitulé *anniversaire.html* dans le dossier *AnniversaireListe* avec le contenu suivant :

```
<html>
<head>
   <title>Liste des anniversaires</title>
   <meta http-equiv="Content-Type" content="text/html;charset=utf-8">
</head>
<body>
   <h1>Liste des anniversaires</h1>
   <p>
      <button onclick="javascript:anniv_liste();">Générer la liste</button>
      <button onclick="javascript:localStorage.clear();">Supprimer</button>
   </p>
   <p>
      <br />Ajouter un anniversaire
      <br />Personne : <input type="text" id="anniv_personne" />
      <br />Date : <input type="text" id="anniv_date" />
      <br/><button onclick="javascript:anniv_ajout();">Ajouter</button>
   </p>
   <p id="anniv_liste" style="background-color:#DDD;">


   </p>
   <script type="text/javascript" src="static/js/anniversaire.js"></script>
   <script type="text/javascript">
```

```
    (function() {
      anniv_liste();
    })();
  </script>
</body>
</html>
```

Après avoir terminé ce fichier, Raphaël ne va pas enregistrer ses modifications dans Git pour le moment. En effet, un commit représente un ensemble de modifications dépendantes les unes des autres et elles doivent former un ensemble cohérent et fonctionnel. Or, s'il exécute son code maintenant, Raphaël obtiendra des erreurs, car les fonctions JavaScript utilisées n'existent pas encore. Il va donc ajouter un fichier JavaScript nommé *anniversaire.js* stocké dans le dossier *static/js* avant de commiter :

```
function anniv_ajout() {
  anniv_personne = document.querySelector('#anniv_personne').value;
  anniv_date = document.querySelector('#anniv_date').value;
  anniversaire = {
    'anniv_personne': anniv_personne,
    'anniv_date': anniv_date
  };
  nb_anniv = anniv_get_nb_anniv();
  localStorage.setItem(nb_anniv.toString(), JSON.stringify(anniversaire));
  nb_anniv++;
  localStorage.setItem('nb_anniv', JSON.stringify(nb_anniv));
  document.querySelector('#anniv_personne').value = "";
  document.querySelector('#anniv_date').value     = "";
  anniv_liste();
}

function anniv_liste() {
  nb_anniv = anniv_get_nb_anniv();
  result = "";
  if (nb_anniv == 0) {
    result = '<h3>Aucun anniversaire enregistré<h3>';
  }
  for (i=0;i<nb_anniv;i++) {
    anniversaire = JSON.parse(localStorage.getItem(i.toString()));
```

```
        result+='<br />'+anniversaire.anniv_date+' : '+
        anniversaire.anniv_personne;
    }
    document.querySelector('#anniv_liste').innerHTML = result;
    return nb_anniv;
}

function anniv_get_nb_anniv() {
    nb_anniv = localStorage.getItem('nb_anniv');
    if (typeof nb_anniv == "undefined" || nb_anniv == null) {
        nb_anniv = 0;
    }
    return nb_anniv;
}
```

Pour que Git prenne en compte ces fichiers, il va falloir les lui indiquer explicitement. Pour cela, Raphaël va utiliser la commande git add :

```
git add anniversaire.html
git add static/js/anniversaire.js
```

Ces commandes auront pour effet d'ajouter les fichiers dans une zone d'attente nommée *index* avant que les fichiers ne soient réellement ajoutés au dépôt par un commit.

Il est possible de vérifier que les fichiers ont correctement été ajoutés dans l'index avec la commande suivante :

```
git status
```

Cette commande affiche la sortie suivante :

```
On branch master


Initial commit


Changes to be committed:
  (use "git rm --cached <file>..." to unstage)


    new file:    anniversaire.html
```

```
    new file:   static/js/anniversaire.js
```

Cette sortie indique que les fichiers se trouvent dans l'index car ils sont prêts à être commités.

Enregistrer des modifications

Une fois que les fichiers sont ajoutés à l'index, il est possible de les commiter pour les ajouter dans l'historique de Git. Un commit ne doit être effectué que sur des modifications qui représentent une tâche ou un travail précis. Il est important de rappeler que Git n'est pas un système de sauvegarde de code, mais un gestionnaire de versions. Les commits doivent représenter une modification unique qui va combler un besoin précis (correctif d'un bug, module d'une nouvelle fonctionnalité, etc.). Il ne faut pas penser qu'un commit qui ne modifie qu'une seule ligne est inutile : s'il répond à un besoin précis avec cette seule et unique ligne (comme la correction d'une syntaxe erronée), il ne faut pas le regrouper avec d'autres modifications pour former un commit plus conséquent.

Chaque commit doit contenir un message de commit qui explique l'origine des modifications. Raphaël commite à présent ses modifications avec la commande git commit :

```
git commit -m "Liste des anniversaires et ajout en localstorage"
```

Il est conseillé de limiter la taille des messages à 49 caractères. Lorsque le commit est important et que 49 caractères ne sont pas suffisants pour écrire un message simple et explicite, il est conseillé d'utiliser l'éditeur de texte pour ajouter le message en n'utilisant pas d'argument -m.

Un nouveau git status indiquera cette fois-ci qu'aucun fichier n'a été modifié depuis le dernier commit. Cela signifie que le commit a bien été pris en compte et que les fichiers ne diffèrent plus de la dernière version du dépôt.

Après avoir effectué le premier commit, Raphaël va s'assurer que son commit s'affiche bien dans la liste des commits en exécutant la commande suivante :

```
git log
```

Cette commande affiche la sortie suivante :

```
commit ccabf27

Author: Samuel DAUZON <git@dauzon.com>

Date:   Thu Sep 24 22:56:39 2015 +0200


    Liste des anniversaires et ajout en localstorage
```

Cette commande permet de lister les commits du dépôt. D'ailleurs, grâce à cette commande, le développeur obtient la chaîne d'identification (sous forme de hash SHA-1) du commit. La chaîne ici est ccabf27 (version raccourcie de l'identifiant qui en réalité est

long de 40 caractères hexadécimaux). Cet identifiant est très utile lorsqu'on souhaite effectuer des actions en relation avec un commit précis.

Maintenant que cette modification a été prise en compte, Raphaël veut centraliser son dépôt sur un dépôt en ligne.

Bitbucket

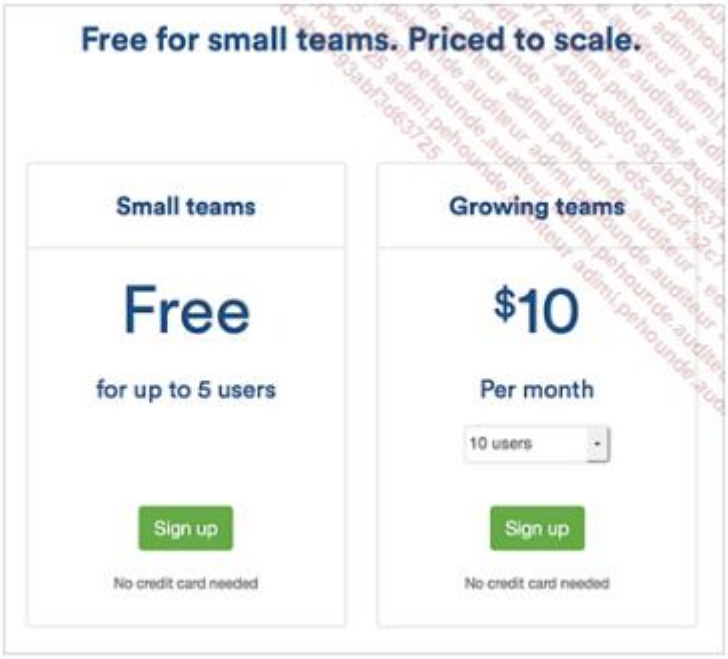
Lorsque les développeurs évoquent des services d'hébergement de dépôt, le principal site dont ils parlent est GitHub, notamment grâce aux très populaires projets libres qu'il héberge (le noyau Linux, Django, Bootstrap, etc.). Cependant, GitHub n'est pas le seul, il existe également Bitbucket qui se fait beaucoup plus discret alors qu'il possède de nombreux points communs avec GitHub.

Bitbucket est un service en ligne qui offre beaucoup de fonctionnalités :

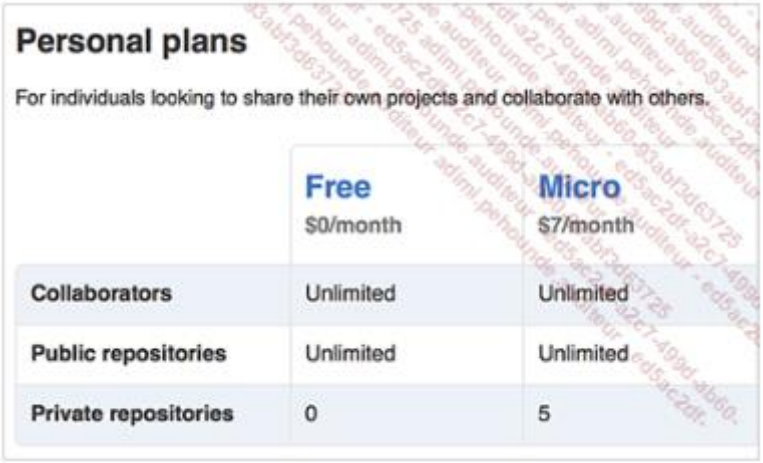
- Stockage de dépôts en ligne : cela permet de travailler à plusieurs sur le dépôt stocké chez Bitbucket.
- Visualisation du dépôt : l'interface web permet de prendre connaissance des modifications effectuées dans le dépôt. L'interface est claire et est plus agréable à utiliser que la ligne de commande pour la consultation, bien qu'elle puisse être déconcertante pour les inconditionnels de la console !
- Réseau social : avec un système de compte personnel, il est possible de participer à des projets libres. Le profil du compte affiche les participations de l'utilisateur aux projets.
- Modification en direct : si un développeur doit faire une modification très simple dans un fichier, il peut le faire directement à partir de l'interface web. Bitbucket permet d'éditer les fichiers et de les commiter à partir d'un compte utilisateur. Bien évidemment, pour des modifications conséquentes il vaut mieux passer par le dépôt local.

Bitbucket possède en outre quelques avantages comparé à GitHub :

- Business model favorisant les indépendants et les petites entreprises travaillant sur de nombreux projets, car GitHub et Bitbucket segmentent les dépôts en deux parties : les dépôts publics qui sont accessibles par n'importe qui et les dépôts privés qui sont accessibles par une équipe définie par l'administrateur du dépôt. Bitbucket favorise les petites équipes, car il ne pose aucune limite au nombre de dépôts privés stockés, mais il facture les équipes supérieures à cinq développeurs en fonction de leur nombre. GitHub, racheté par Microsoft en juin 2018, a longtemps refusé de proposer des dépôts privés gratuits. En janvier 2019, GitHub a commencé à proposer des dépôts privés gratuits limités à trois collaborateurs. Il faut donc choisir en fonction de la taille de l'équipe, du prix et des fonctionnalités proposées. Voici un exemple des tarifs de Bitbucket et GitHub (en septembre 2015) :



Tarifs de Bitbucket



Tarifs de GitHub

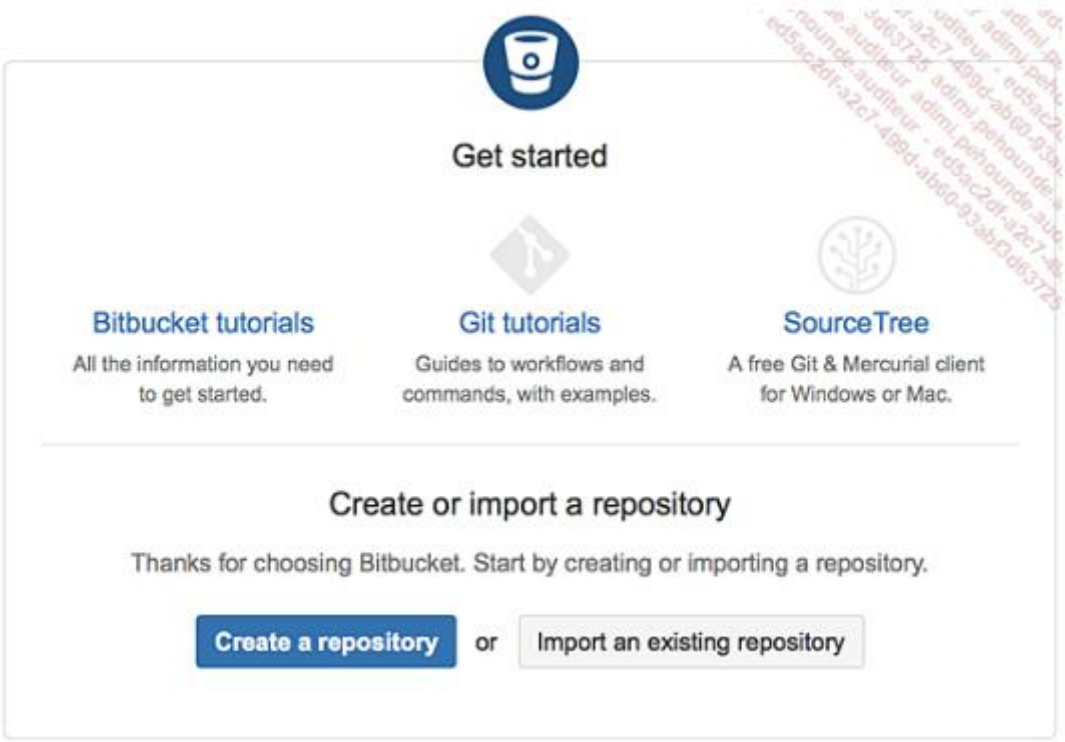
- Compatibilité avec Mercurial : Bitbucket offre également une compatibilité avec Mercurial. Il permet donc d'utiliser les deux types de SCM.

1. Création d'un compte

Pour les besoins de ce cours, Raphaël va créer un compte sur Bitbucket comme s'il ne possédait aucun compte. Il se connecte donc sur https://bitbucket.org, clique sur le bouton **Get started for free**, puis entre ses informations personnelles nécessaires à la création du compte.

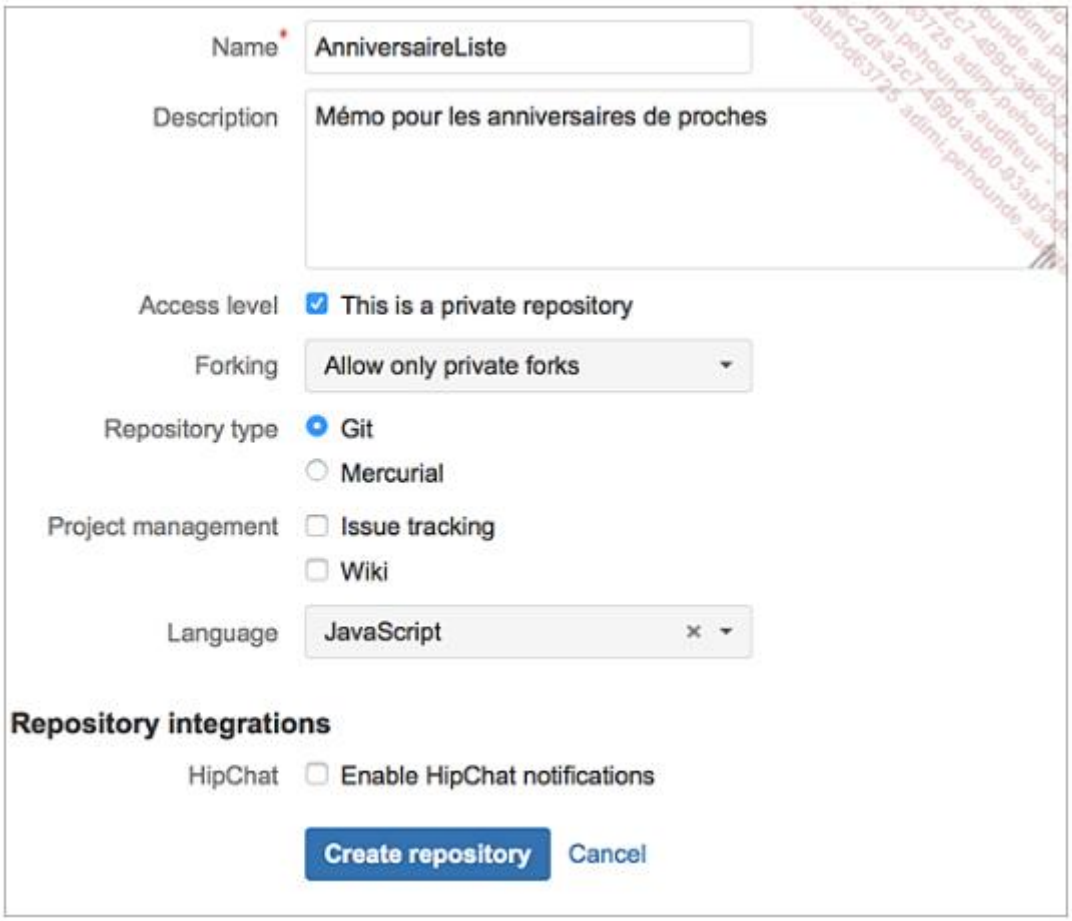
La traduction française du site n'est pas complète à l'heure où ces lignes sont écrites. Pour éviter d'afficher les deux langues simultanément et de créer de la confusion, les exemples de ce cours seront présentés uniquement en anglais.

Une fois le compte créé, Bitbucket envoie un mail contenant un lien permettant de valider la création du compte. Après avoir suivi ce lien, une page d'accueil s'affiche :

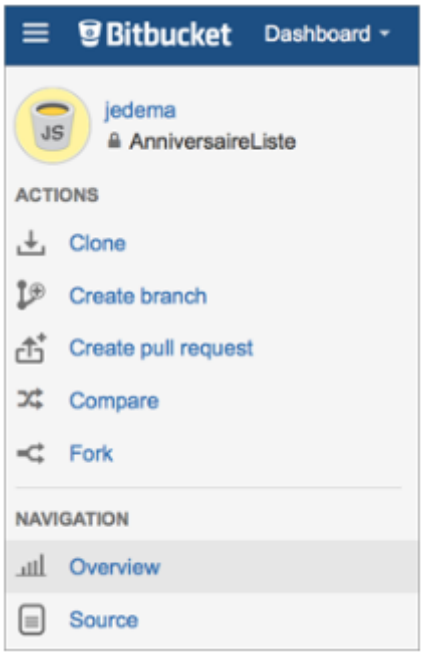


Après avoir cliqué sur le bouton **Create a repository**, un formulaire s'affiche alors dont il faudra remplir plusieurs champs :

- **Name** : définit le nom du projet. Dans cet exemple : **AnniversaireListe**.
- **Description** : c'est une description simple du projet. Il ne faut pas copier tout le contenu du fichier *README* dans ce champ. Dans cet exemple : **Mémo pour les anniversaires de proches**.
- **Access level** : lorsque ce champ est coché, le projet est privé, il ne sera donc accessible qu'aux personnes autorisées. Dans cet exemple, la case sera cochée.
- **Forking** : le fait de forker un dépôt correspond à cloner un dépôt, à la différence que le fork est effectué par le biais d'un hébergeur de dépôts tel que GitHub, Bitbucket ou encore GitLab (qui permet d'héberger soi-même ses propres dépôts).
- **Repository type** : permet de choisir si le dépôt est de type Git ou de type Mercurial. Dans cet exemple, **Git**.
- **Project management** : permet d'utiliser des outils supplémentaires. Ces outils sont facultatifs. Dans cet exemple, aucun outil supplémentaire ne sera utilisé.
- **Language** : permet de définir le langage principal utilisé dans le projet (ici : JavaScript).
- **HipChat** : permet d'utiliser un outil de chat proposé par Atlassian, le même éditeur que Bitbucket.



Une fois le formulaire validé, Bitbucket redirige vers la page d'accueil du dépôt nouvellement créé. Cette page est composée d'un menu sur la gauche qui permet d'effectuer un certain nombre d'actions sur le dépôt.



2. Envoyer un dépôt local vers Bitbucket

Pour importer les éléments de son dépôt local, Raphaël va utiliser les commandes suivantes à partir du dossier de son dépôt :

```
git remote add origin
https://example@bitbucket.fr/rdauzon/anniversaireliste.git
git push -u origin --all
```

La première commande permet de renseigner dans la configuration du dépôt local le dépôt distant auquel il faut se référer. La deuxième commande permet d'envoyer les éléments du dépôt local vers le dépôt distant. Lors de l'exécution de la deuxième commande, le mot de passe du compte Bitbucket est demandé. Cette mesure de sécurité permet de ne pas autoriser n'importe qui à envoyer ses données.

La documentation de Bitbucket conseille d'exécuter également une commande permettant d'envoyer les tags du dépôt. Étant donné que le dépôt ne contient aucun tag, cette commande n'est pas utilisée. Voici néanmoins la commande qui permet d'envoyer les tags du dépôt :

```
git push -u origin --tags
```

Après avoir envoyé les éléments du dépôt vers Bitbucket, il est possible de consulter la liste des commits en cliquant sur le menu **Commits**. La liste s'affiche alors avec le seul commit actuel :

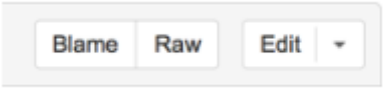
3. Éditer un fichier sur Bitbucket

Pour cela, Raphaël doit cliquer sur le lien **Source** présent dans le menu. Les fichiers et dossiers présents à la racine du dépôt s'affichent. Il est possible de naviguer dans cette arborescence par l'intermédiaire des liens présents sur les noms des dossiers. Pour éditer ou consulter un fichier, il faut cliquer sur le nom de ce fichier.

Le contenu du fichier s'affiche alors dans un encadré en haut duquel se situent plusieurs boutons :

- Le bouton de suivi d'historique contenant les informations sur les commits ayant modifié le fichier.
- Le bouton **Blame**, qui permet d'effectuer l'équivalent d'un git blame directement à partir de l'interface web. Ce bouton permet d'afficher les informations sur les lignes du fichier (développeur qui a modifié la ligne et commité).
- Le bouton **Raw** permet de télécharger ou d'afficher le fichier sous forme brute, c'est-à-dire sans aucune mise en forme ni coloration syntaxique.

- Le bouton **Edit** permet d'éditer le fichier.
- À droite du bouton **Edit** se situe un bouton composé d'un petit triangle, qui permet de renommer ou de supprimer le fichier.



Il faut ajouter le commentaire suivant avant la définition de la fonction JavaScript anniv_liste() :

```
/* Cette fonction affiche sur la page la liste des anniversaires
stockés dans le localstorage du navigateur */
```

Une fois cette modification effectuée, Raphaël décide de la commiter en cliquant sur le bouton **Commit**. Il entre alors le message de commit suivant :

```
Documentation de la fonction anniv_liste()
```

Pour finaliser le commit, Raphaël clique sur le bouton **Commit**.

Ce commit a uniquement été ajouté sur Bitbucket. Si Raphaël exécute à partir de son interface la commande git log, il ne trouvera aucune trace de ce commit ; c'est parce que le dépôt local n'a aucune trace de ce commit.

4. Récupérer les modifications du dépôt distant

La modification qui vient d'être commitée sur Bitbucket existe uniquement sur le dépôt distant. Le dépôt local n'a aucune information sur cette dernière modification. Pour que le dépôt local connaisse ce commit, il va falloir l'intégrer à partir du dépôt distant.

Pour cela, Raphaël exécute la commande suivante :

```
git pull
```

La commande git log affiche alors les deux commits : le premier commit et celui effectué à partir de l'interface de Bitbucket.

Intégrer un nouveau développement

Suite aux demandes de son client, Raphael va devoir modifier l'application. En effet, le client souhaite pouvoir ajouter une couleur de son choix pour chaque anniversaire. Le client veut payer le moins possible et demande donc la solution la moins onéreuse et la plus simple.

Raphaël modifie donc le fichier *anniversaire.html* de la sorte :

- En dessous du champ date, il ajoute un champ pour définir la couleur :

```
<br />Couleur (ex : 0F0): <input type="text" id="couleur" />
```

- Il modifie ensuite les fonctions anniv_ajout() et anniv_liste() du fichier *static/js/anniversaire.js* :

```
function anniv_ajout() {
    anniv_personne = document.querySelector('#anniv_personne').value;
    anniv_date = document.querySelector('#anniv_date').value;
    couleur = document.querySelector('#couleur').value;
    anniversaire = {
        'anniv_personne': anniv_personne,
        'anniv_date': anniv_date,
        'couleur': couleur
    };
    nb_anniv = anniv_get_nb_anniv();
    localStorage.setItem(nb_anniv.toString(), JSON.stringify(anniversaire));
    nb_anniv++;
    localStorage.setItem('nb_anniv', JSON.stringify(nb_anniv));
    document.querySelector('#anniv_personne').value = "";
    document.querySelector('#anniv_date').value = "";
    document.querySelector('#couleur').value = "";
    anniv_liste();
}

/* Cette fonction affiche sur la page la liste des anniversaires
stockés dans le localstorage du navigateur */
function anniv_liste() {
    nb_anniv = anniv_get_nb_anniv();
    result = "";
    if (nb_anniv == 0) {
        result = '<h3>Aucun anniversaire enregistré<h3>';
    }
    for (i=0;i<nb_anniv;i++) {
        anniversaire = JSON.parse(localStorage.getItem(i.toString()));
        result+='<br />
        <span style="background-color:#'+anniversaire.couleur+'">
        '+anniversaire.anniv_date+' :
        '+anniversaire.anniv_personne+'</span>';
    }
```

```
    document.querySelector('#anniv_liste').innerHTML = result;
    return nb_anniv;
}
```

1. Vérifier son code avant l'indexation

Maintenant qu'il a effectué ces modifications, Raphaël peut commiter. Il a cependant un doute sur les modifications effectuées. Il va donc regarder les modifications entre son répertoire de travail et ce qui est présent dans la dernière version du dépôt. Il va utiliser git diff pour cela.

La sortie d'écran de cette commande affiche toutes les lignes qui ont été ajoutées et supprimées. La coloration de la console permet de facilement repérer les lignes qui ont été ajoutées et les lignes qui ont été supprimées. C'est une bonne pratique de relire le code ajouté ou supprimé avant de commiter. Cela permet de déceler certaines erreurs ou certains oublis.

Raphaël a vérifié les modifications et va donc ajouter ces deux fichiers à l'index :

```
git add anniversaire.html
git add static/js/anniversaire.js
```

2. Commiter le nouveau développement

Avant de commiter quoi que ce soit, il est conseillé de vérifier les fichiers qui vont être intégrés dans le commit. Cette vérification peut sembler superflue étant donné que Raphaël vient d'ajouter uniquement deux fichiers à l'index, mais elle s'avère d'autant plus utile que le développement est complexe et implique un nombre important de fichiers. Raphaël va donc vérifier les fichiers présents dans l'index et, après validation des fichiers, il va exécuter la commande suivante :

```
git commit -m "Ajout d'une couleur pour chaque anniversaire"
```

Pour vérifier que le commit a été ajouté au dépôt, Raphaël peut effectuer un git log -1 qui va afficher les détails du dernier commit. En effectuant un git status, Raphaël peut remarquer que l'index est vide, ce qui est tout à fait normal car la version du dépôt est à présent la même que celle de l'index.

Annuler les modifications d'un fichier

Pour essayer d'améliorer l'exemple de couleur donnée par le libellé du champ couleur, Raphaël modifie dans le fichier *anniversaire.html* la ligne contenant la couleur du champ :

```
<br />Couleur (ex : 0F0 ou 00FF00, le nom des couleurs n'est pas
pris en charge) : <input type="text" id="couleur" />
```

Après avoir testé cette modification, Raphaël se rend compte qu'elle n'est pas réellement très utile et qu'elle peut créer de la confusion. Il décide donc de la supprimer. Il va pour cela utiliser la commande suivante :

```
git checkout anniversaire.html
```

Cette commande permet de demander à Git de restaurer l'état du fichier *anniversaire.html* tel qu'il est dans HEAD.

.gitignore : ignorer une bibliothèque

Lorsque Raphaël a fait une démonstration de la page web à son client, celui-ci lui a rétorqué : "Mais ce n'est pas pratique, comment puis-je me souvenir de tous ces codes de couleurs ? Je ne peux pas avoir quelque chose de mieux pour pas cher ?".

Raphaël va donc lui proposer un champ qui lui permettra de sélectionner une couleur à partir d'une palette. Le code couleur sera automatiquement généré à partir de la couleur choisie. En proposant cela, Raphaël sait qu'il va utiliser une bibliothèque existante qu'il a déjà rencontrée dans d'autres développements. Cela lui permet aussi de garantir un tarif très attractif, car il va réutiliser un code déjà existant.

Après que Raphaël lui a donné le tarif de la prestation, M. Airadun a directement accepté tout en marmonnant quelques réticences : "Encore une facture... Si ça avait directement été fait...".

Après avoir quitté son client, Raphaël a commencé par télécharger la bibliothèque JSColor sur le site officiel (http://jscolor.com). Il modifie ensuite le fichier *anniversaire.html* et ajoute la ligne suivante juste avant la balise fermante </head> :

```
<script type="text/javascript"
src="static/lib/js/jscolor/jscolor.js"></script>
```

Et il va modifier la ligne contenant le champ permettant de définir la couleur avec le contenu suivant :

```
<br />Couleur (ex : 0F0) : <input type="text" id="couleur"
class="color" />
```

Il va ensuite créer un dossier *lib* (pour *library*) à la racine de son projet. Dans ce dossier *lib*, il va créer un dossier *js* qui va servir à stocker toutes les bibliothèques JavaScript qu'il va utiliser.

Il va ensuite copier le contenu de la bibliothèque dans le dossier *static/lib/js*. Après avoir vérifié que le code est bien fonctionnel, Raphaël ne va pas commiter immédiatement. En effet, il ne va pas avoir besoin de suivre les évolutions de la bibliothèque car il ne va pas la modifier. Il va donc indiquer à Git de ne pas suivre les fichiers contenus dans le dossier *lib*. Si le projet avait été plus conséquent et le client moins pressant, Raphaël aurait très certainement utilisé les dépôts intégrés (ou sous-modules).

Raphaël va utiliser la commande git status pour vérifier que la bibliothèque est présente et non suivie. Cette commande lui affiche la sortie suivante (tronquée) :

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
    static/lib/


no changes added to commit (use "git add" and/or "git commit -a")
```

Cette commande affiche uniquement le dossier *static/lib* car Git ne suit aucun fichier présent dans ce répertoire. Pour vérifier l'état de ses fichiers, Raphaël exécute alors la commande git ls-files -o et obtient la sortie suivante :

```
static/lib/js/jscolor/arrow.gif
static/lib/js/jscolor/cross.gif
static/lib/js/jscolor/demo.html
static/lib/js/jscolor/hs.png
static/lib/js/jscolor/hv.png
static/lib/js/jscolor/jscolor.js
```

La commande précédente permet de lister les fichiers non suivis. Au total, six fichiers sont présents dans l'inclusion de la bibliothèque.

Raphaël souhaite que Git ignore ces fichiers, c'est-à-dire que Git ne s'en occupe jamais, comme s'ils n'existaient pas.

Pour cela, il va créer le fichier *.gitignore* à la racine du projet (dans le dossier qui contient le dossier *.git* masqué sur la plupart des systèmes).

Ce fichier *.gitignore* va permettre d'indiquer à Git les fichiers qu'il ne doit pas versionner. En effet, il existe des fichiers qu'il vaut mieux ne pas versionner :

- les fichiers générés ou compilés, car le contenu des fichiers ne peut pas être suivi et est illisible. Les fichiers qui doivent être versionnés sont les fichiers lisibles et modifiables par des humains.
- Les fichiers de configuration contenant des mots de passe, car les collaborateurs ne doivent pas forcément avoir accès à vos mots de passe. Cette règle est d'autant plus vraie lorsque le projet est libre.

Raphaël va ajouter la ligne suivante dans le fichier *.gitignore* :

```
lib/
```

Cette ligne signifie qu'aucun des fichiers contenus dans le dossier *lib* ne sera pris en compte par Git. Cela ne veut pas seulement dire qu'ils ne seront pas versionnés, mais également que lorsqu'on exécutera des commandes comme git status, Git ignorera totalement ces fichiers comme s'ils n'existaient pas.

Raphaël va vérifier que les fichiers de la bibliothèque sont correctement ignorés par Git avec la commande git status. La sortie indique que le fichier *.gitignore* n'est pas suivi, mais ne donne plus aucune indication sur le dossier *static/lib*, ce qui signifie qu'il est correctement ignoré.

La commande git-ls-files -o indiquera toujours la présence des fichiers, car c'est une commande de plomberie. Elle passe outre les fichiers ignorés.

Raphaël va ensuite ajouter le fichier *.gitignore* à son dépôt Git pour que les autres développeurs ne soient pas pollués par les fichiers de la bibliothèque.

```
git add .gitignore
git commit -m "Ajout du .gitignore"
```

Il existe de meilleurs moyens de ne pas polluer l'historique avec les bibliothèques. Pour avoir plus d'informations sur les dépôts intégrés, il faut consulter la section Dépôts intégrés avec submodules. Les dépôts intégrés permettent de gérer beaucoup plus facilement les bibliothèques externes en suivant leurs versions sans polluer l'historique du projet. Dans cet exemple, Raphaël a choisi la simplicité pour ne pas polluer son historique (et peut-être aussi pour ne pas trop complexifier ce scénario). Si le projet évolue, Raphaël utilisera les dépôts intégrés.

Commiter tous les fichiers ajoutés ou modifiés

Depuis le début, Raphaël ajoute les fichiers un à un dans l'index. C'est une opération répétitive, surtout lorsqu'on doit ajouter tous les fichiers ajoutés ou modifiés.

Il existe un argument à la commande git add qui permet d'ajouter tous les fichiers ajoutés/modifiés à l'index : -A.

Tout d'abord, Raphaël doit vérifier tous les fichiers ajoutés/modifiés avec la commande git status. La commande lui indique que seul le fichier *anniversaire.html* a été modifié. En utilisant la commande suivante, Git va ajouter les fichiers modifiés à l'index (ici uniquement *anniversaire.html*) :

```
git add -A
```

Avant de commiter, Raphaël va vérifier que le fichier ne se trouve pas dans l'index avec la commande git status. Il commite alors la modification qui permet de choisir la couleur avec la bibliothèque :

```
git commit -m "Ajout fenetre choix couleur"
```

Cette commande permet de gagner beaucoup de temps, mais elle ne doit pas empêcher de respecter deux règles :

- Il y a des fichiers qu'il ne faut pas versionner : si Raphaël avait utilisé la commande git add -A avant de mettre en place le fichier *.gitignore*, il aurait versionné tous les fichiers de la bibliothèque. C'est grâce à *.gitignore* que le dernier commit contient uniquement les fichiers nécessaires.
- L'atomicité des commits : les commits doivent représenter une modification indépendante des autres. Avec la commande git add -A, il ne faut pas commiter un ensemble de modifications là où plusieurs commits seraient plus adaptés.

Il est également possible de ne pas du tout passer par l'index pour commiter tous les fichiers déjà suivis. En effet, en utilisant la commande suivante, tous les fichiers modifiés déjà suivis seront ajoutés directement dans le prochain commit.

```
git commit -a -m "message de commit"
```

Après avoir vérifié que son dernier commit a correctement été ajouté, Raphaël vérifie alors sur Bitbucket si le commit est présent en rafraîchissant la page **Commits**, et il se rend compte que le commit n'a pas été envoyé. En effet, le dernier commit a été effectué en local et n'est présent que sur le dépôt local. Raphaël doit donc l'envoyer de manière explicite à son dépôt distant chez Bitbucket.

Envoyer les commits au dépôt distant

En validant ses modifications, Raphaël a ajouté un commit à son dépôt local, mais pour l'instant le dépôt distant n'a aucune information sur ce dernier commit. Il va donc envoyer les modifications de sa branche courante vers le dépôt distant.

```
git push
```

Le résultat indique que l'envoi a bien été effectué. Si un développeur est consciencieux comme Raphaël, il va également vérifier sur l'interface web de Bitbucket que son commit a été correctement pris en compte.

Afficher les différences entre deux commits

Imaginons le cas où cela fait plusieurs semaines que Raphaël n'a pas pu travailler sur son projet pour diverses raisons (attente de documents du client, vacances, etc.). Pour appréhender à nouveau le but du projet et ce qui a déjà été fait, il consulte la liste des commits (avec git log) et il décide de regarder les modifications du dernier commit. Pour cela, il doit exécuter la commande :

```
git diff 4702 40a42
```

L'identifiant du dernier commit commence par 40a42 et l'avant-dernier commence par 4702. La commande git diff fonctionne en spécifiant deux commits.

La réponse de Git prend la forme d'une liste de lignes. Les lignes supprimées par le commit sont préfixées d'un - et sont en rouge. Les lignes ajoutées sont préfixées d'un + et sont en vert. Les autres lignes, de couleur noire, entourent chaque modification.

Il est aussi possible de voir les modifications apportées par un commit directement à partir de Bitbucket. L'avantage de passer par la ligne de commande est que vous pouvez travailler sans être dépendant du client Git.

Cloner le dépôt distant

Pendant quelques jours, Raphaël va partir en vacances. Il ne compte pas abandonner son projet pour autant et va emmener un petit ordinateur pour le continuer tranquillement. Pour avoir un dépôt propre et fonctionnel, il va cloner son dépôt sur son PC portable. Pour cela, il lui faut Git sur son PC portable. Il faut également qu'il récupère l'URL de clonage

sur Bitbucket. Cette URL se trouve sur la page principale du dépôt, après avoir cliqué sur le lien **Clone** du menu comme le montre la capture ci-dessous :

Ensuite, il va utiliser l'interface en ligne de commande pour se placer dans le dossier qui contiendra le projet et utiliser la commande :

```
git clone https://rdauzon@bitbucket.org/rdauzon/
anniversaireliste.git anniversaireVacances
```

Cette commande aura pour effet de copier le dépôt dans un nouveau dossier *anniversaireVacances*. Cela signifie que Raphaël a accès à tous les précédents commits et aux modifications qu'ils contiennent. Cloner un dépôt ne revient pas uniquement à copier le répertoire de travail, mais à copier l'ensemble des éléments d'un dépôt.

D'ailleurs, les fichiers configurés dans *.gitignore* n'ont pas été copiés. En effet, ils sont totalement ignorés par Git. Il faudra donc les copier manuellement pour le moment. Ce comportement peut ne pas sembler très pratique étant donné qu'il est malgré tout nécessaire de copier manuellement une liste de fichiers pour avoir un dépôt fonctionnel. Pourtant, cette limitation est très pratique lorsque *.gitignore* doit ignorer des fichiers sensibles comme ceux contenant des mots de passe d'API. D'ailleurs, Git est un système de gestion de versions, pas un système de déploiement. Il n'a donc pas pour vocation de déployer une application pour la rendre fonctionnelle.

Une branche, ça sert à quoi ?

Les branches représentent l'une des fonctionnalités les plus intéressantes de Git. Dans chaque dépôt Git, une branche est présente par défaut : la branche *master*. Cette branche *master* correspond généralement à la branche principale du projet.

Les branches correspondent à la déviation d'une autre branche (comme peut l'être une branche d'arbre vis-à-vis du tronc qui la porte). En d'autres termes, une branche est utilisée pour maintenir une version particulière du projet dans un but précis et fixé à l'avance.

Elles peuvent servir à :

- développer une nouvelle fonctionnalité sans polluer la branche supérieure avec des modifications non validées,
- garder une branche correspondant à une version stable définie. Par exemple, un développeur a créé une version 1.0 d'un logiciel et il ne souhaite pas y intégrer les dernières modifications.

Raphaël a envie de proposer une nouvelle fonctionnalité à son client, à savoir une proposition des couleurs utilisées. C'est-à-dire qu'en dessous du champ texte permettant de définir la couleur, Raphaël va ajouter des boutons correspondant aux couleurs déjà utilisées sur le site. Lorsque l'utilisateur cliquera sur l'un de ces boutons, le champ **couleur** sera rempli avec la valeur choisie. Il va tout d'abord créer sa nouvelle branche (dans son dépôt *anniversaireVacances*) :

```
git branch color_buttons
```

```
git checkout color_buttons
```

Cette commande va indiquer à Git que Raphaël souhaite travailler dans une nouvelle branche. En réalité, la branche ne sera réellement créée qu'à partir du premier commit réalisé dans celle-ci.

Ensuite, il va positionner la balise div qui contiendra les boutons en dessous du champ couleur :

```
<div id="couleurs_proposees"></div>
```

Raphaël va ajouter une nouvelle bibliothèque à son projet : jQuery. En effet, jQuery est un framework JavaScript qui permet de simplifier un certain nombre d'actions, par exemple écouter un certain nombre d'événements. jQuery sera utilisé pour surveiller les clics effectués sur les boutons. Pour ajouter la bibliothèque, il faut la télécharger sur le site de jQuery (https://jquery.com/download/), la placer dans le dossier *static/lib/js/jquery.1.13.min.js* et, pour finir, il faut ajouter la ligne d'import de jQuery juste avant la ligne d'import de *anniversaire.js* :

```
<script type="text/javascript"
src="static/lib/js/ jquery.1.13.min.js"></script>
```

Il faut ensuite créer la fonction qui va générer les boutons dans le fichier *anniversaire.js* :

```
function charge_couleurs() {
    nb_anniv = anniv_get_nb_anniv();
    // On vide la div contenant les couleurs si elle en contient
    // pour recréer les couleurs.
    couleur_div = document.querySelector('#couleurs_proposees');
    couleur_div.innerHTML = '';

    couleur_list = Array();
    for (i=0;i<nb_anniv;i++) {
        anniversaire = JSON.parse(localStorage.getItem(i.toString()));
        couleur_list.push(anniversaire.couleur);
    }
    couleur_list_unique = couleur_list.filter(function(elem, pos) {
        return couleur_list.indexOf(elem) == pos;
    })
    // Génération et affichage des boutons
    couleur_list.filter(function(elem, pos) {
        couleur_div.innerHTML = couleur_div.innerHTML +
```

```
        '<button style="color:#'+elem+'">'+elem+'</button>';
    })
    // Écouteurs de boutons pour remplir la couleur choisie
    $("#couleurs_proposees button").bind("click", function() {
        $('#couleur').val($(this).text());
    });
}
```

La fonction est créée, mais pour l'instant elle n'est jamais exécutée. Raphaël décide donc de charger les couleurs lors du chargement de la page en ajoutant le code suivant dans le fichier JavaScript :

```
$(document).ready(function() {
    charge_couleurs();
});
```

Pour que la page soit plus dynamique, Raphaël va également appeler la fonction charge_couleurs() à la fin de la fonction anniv_ajout().

Après avoir testé ses modifications, Raphaël peut créer un nouveau commit avec le message **Proposition des couleurs déjà utilisées**. C'est avec ce commit que la branche aura une réelle existence dans Git car il faut un commit pour qu'une branche soit réellement intégrée à l'historique. Pour créer le commit, il utilisera les commandes suivantes :

```
git add anniversaire.html
git add static/js/anniversaire.js
git commit -m "Proposition des couleurs déjà utilisées"
```

Raphaël va alors passer à la deuxième partie de sa modification : lorsque l'utilisateur commencera à entrer le code d'une couleur, les boutons des couleurs qui ne correspondent pas à ce qu'il entre dans le champ seront masqués.

Raphaël va ajouter un écouteur sur le champ couleur_proposees dans son fichier *anniversaire.js* :

```
$("#couleur").bind( "keyup", function() {
    masque_couleur($(this).val());
});
```

Il va ensuite créer la fonction qui va lui permettre de cacher les boutons des couleurs ne correspondant pas au texte du champ et l'ajouter au même fichier :

```
function masque_couleur(color) {
```

```
    $('#couleurs_proposees button').each(function( index ) {
        if ($(this).html().indexOf(color) > -1) {
            $(this).show();
        }
        else {
            $(this).hide();
        }
    });
}
```

Une fois qu'il a terminé ses modifications, Raphaël va tester le code avant de commiter.

```
git add static/js/anniversaire.js
git commit -m "Suggestions des boutons couleurs"
```

Raphaël va vérifier que ses commits ont bien été pris en compte avec la commande git log puis il peut mettre à jour le dépôt distant avec la commande suivante :

```
git push --set-upstream origin color_buttons
```

Changer de branche

Raphaël reçoit un appel urgent de son client. En effet, celui-ci n'aime pas du tout le titre de la page. Il demande donc à Raphaël de modifier le titre qui était **Liste des anniversaires** en **Page des anniversaires**.

Raphaël ne va pas effectuer cette modification sur la branche contenant les boutons des couleurs. En effet, le client refusera peut-être les boutons ou demandera une autre modification avant la mise en production. Raphaël va donc revenir sur la branche principale (la branche par défaut de Git s'appelle *master*) pour modifier le titre. Pour cela, il va utiliser la commande suivante :

```
git checkout master
```

Cette commande permet de changer de branche. Un changement de branche signifie que le répertoire de travail va être mis à jour à partir des données de la branche spécifiée dans la commande. Dans ce cas, cela signifie que les modifications effectuées dans la branche *color_buttons* seront effacées du répertoire de travail, mais conservées par Git.

Après avoir exécuté cette commande, il est possible de vérifier la branche sur laquelle le prochain commit sera effectué avec la commande suivante :

```
git branch
```

Cette commande affiche la sortie suivante :

```
  color_buttons
* master
```

Cette sortie indique qu'il y a deux branches dans le dépôt et que l'état du répertoire de travail correspond à la branche *master*.

Si Raphaël consulte les fichiers, il ne verra plus les modifications qu'il a effectuées pour mettre en place le système de boutons.

En regardant plus précisément les fichiers, Raphaël remarque que la bibliothèque jQuery qu'il a ajoutée dans la branche *color_buttons* est toujours présente. C'est normal, étant donné qu'elle est ignorée par Git (dans le fichier *.gitignore*), elle n'a aucune existence pour Git quelle que soit la branche dans laquelle Raphaël était au moment de l'ajout.

Raphaël change donc le titre dans le fichier *anniversaire.html* et commite sa modification avec les commandes suivantes :

```
git add anniversaire.html
git commit -m "Titre page : Page des anniversaires"
```

Lorsque son commit est terminé, il est situé uniquement dans la branche *master*, la branche *color_buttons* n'ayant reçu aucune modification.

Fusionner deux branches

Pour présenter une démonstration du système de boutons à son client, Raphaël va se replacer dans la branche *color_buttons* :

```
git checkout color_buttons
```

Il retrouve alors les modifications qu'il avait effectuées dans cette branche. Seulement, il n'y a pas la modification de titre qu'il vient de faire dans la branche *master*. Son client ne va pas être très ouvert à sa proposition s'il n'y a pas les modifications demandées. Raphaël va donc récupérer les modifications effectuées sur la branche *master* à l'aide de la commande git merge :

```
git merge master
```

Lorsque Raphaël exécute la commande précédente, Git lui propose un message pour le commit qui va représenter les modifications de *master* qui seront ajoutées dans la branche *color_buttons*. Il garde ce message pour son commit.

Cette commande va intégrer les modifications de la branche *master* dans la branche *colors_buttons*. Implicitement, la commande git merge signifie que les modifications de la branche spécifiée seront ajoutées dans la branche sur laquelle le dépôt est positionné. Cette étape de fusion engendre parfois un conflit lorsque les mêmes parties du fichier ont été modifiées sur les deux branches. Les conflits doivent alors être résolus manuellement par le développeur.

Après avoir fait la démonstration du système de boutons à son client, celui-ci a accepté la mise à jour pour une mise en production. Il était très satisfait d'avoir un système qui lui permette de gagner autant de temps et de se souvenir de tous les anniversaires de sa famille.

Raphaël va donc inclure les modifications de la branche *color_buttons* vers sa branche de production *master*. Il va donc tout d'abord se placer sur sa branche *master* :

```
git checkout master
```

Il va ensuite fusionner les deux branches :

```
git merge color_buttons
```

Cette commande affiche la sortie suivante :

```
Updating 7a5372c..01555d8
Fast-forward
```

L'affichage du texte **Fast-forward** indique que Git n'a pas eu besoin d'appliquer les modifications contenues dans la branche *color_buttons*. En effet, Git a détecté que le commit le plus récent de la branche *color_buttons* représente exactement la même version (en tenant uniquement compte du contenu des fichiers du dépôt) que la version à laquelle aurait dû arriver la branche *master* après la commande git merge color_buttons.

Après cela, la branche *color_buttons* n'est plus nécessaire. Pour laisser son dépôt propre, Raphaël décide de la supprimer. Pour cela, il utilise la commande suivante :

```
git branch -d color_buttons
```

Cette commande supprime la branche spécifiée. Par sécurité, Git empêche le développeur d'exécuter cette commande si la branche possède des modifications non présentes dans une autre branche.

Ce scénario permet de visualiser l'utilisation de Git lorsqu'un développeur travaille seul. Il permet de mieux appréhender les actions simples de Git et comment ces actions s'enchaînent dans le flux de travail quotidien. Ce scénario couvre beaucoup de concepts de Git rapidement.