

Persistence objet

Prise en main

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

18 février 2016

L'enseignant

- Olivier Cailloux
- olivier.cailloux@dauphine.fr
- Coordonnées : cf. [annuaire](#) de Dauphine

Objectifs pédagogiques

- Apprentissage de la persistance : pourquoi faire ?

Objectifs pédagogiques

- Apprentissage de la persistance : pourquoi faire ?
 - Sauvegarde de l'état de l'application
 - Object / Relational Mapping : paradigme haut-niveau pour maintenance facilitée
 - Peut simplifier le développement (prq ?)

Objectifs pédagogiques

- Apprentissage de la persistance : pourquoi faire ?
 - Sauvegarde de l'état de l'application
 - Object / Relational Mapping : paradigme haut-niveau pour maintenance facilitée
 - Peut simplifier le développement (prq ? Navigation dans réseau d'objets peut être facilitée !)

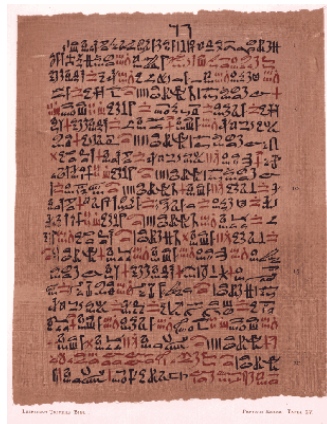
Objectifs pédagogiques

- Apprentissage de la persistance : pourquoi faire ?
 - Sauvegarde de l'état de l'application
 - Object / Relational Mapping : paradigme haut-niveau pour maintenance facilitée
 - Peut simplifier le développement (prq ? Navigation dans réseau d'objets peut être facilitée !)
- Prise en main d'outils de dév avancés :
 - eclipse ;
 - Maven ;
 - git...
- Apprendre à se débrouiller
 - Installation d'outils,
 - debug...

Aperçu des bienfaits (?) de la complexité !

Les bienfaits de la complexité

- Activité plus difficile : souvent plus attrayante
- Évite les activités répétitives : complexité amène diversité
- Récompenses plus grandes
- Recherche d'un accomplissement personnel
- Cercle vertueux : accès à activités plus complexes
- Pas un bien positionnel : accessible à tous



Écriture hiératique, égypte ancienne

Approche pédagogique

- Peu de solutions clé en main
- Il vous faudra *comprendre*
- Chercher dans la documentation (liens fournis)

Deux astuces importantes

- Vous aurez *presque* toutes les clés
- Posez des questions !

Implémentation sur applications illustratives

- Implémentation des techniques sur une *application illustrative*
- Une application \neq par groupe : 3 ou 4 membres
- Chaque membre responsable d'au moins une classe persistante non triviale
- Ajout éventuel de classes artificielles pour implémentations requises (t.q. héritage)

Demande

- Implémentation au fur et à mesure, par chacun ou le groupe
- Je vous indiquerai les contenus et dates de remise
- Remise exclusivement par serveur git

Évaluation

Contrôle continu

- 50%
- qualité du code concernant JDBC ou JPA en isolation

Soutenance

- 50%
- mise en œuvre adéquate des technologies dans l'application
- qualité de la présentation finale
- fonctionnalités
- qualité générale de l'application

Possibilité de rendre l'application finale fin avril.

Git

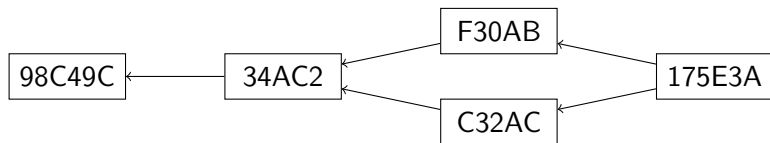
- Contrôle de version (VCS, SCM) : conserver l'historique
- Pour tous types de projet : code, images, présentations, article...
- VCS local, centralisé, distribué ?

Git

- Contrôle de version (VCS, SCM) : conserver l'historique
- Pour tous types de projet : code, images, présentations, article...
- VCS local, centralisé, distribué ?
- Centralisé : seulement sur un serveur distant
- Distribué : copie locale et distante
- Git : distribué

Commits et historique

- Blob : capture d'un fichier à un moment donné
- Commit : identifié par un hash SHA-1
 - Contient : structure de répertoires ; *blobs* ; auteur...
- Histoire : un DAG de « commits »



Work dir (WD)

- Histoire conservée *localement* dans `.git` à la racine du projet
- WD (« work dir ») : version du projet (fichiers et sous-répert.)
- Interaction avec sous-rép. `.git` : *uniquement* via outils git

/root

 /.git

 /rép1

 /fich1

 /fich2

Préparer un commit

Work dir

/rép1

/fich1

/fich2

/fich3

Index

/rép1

/fich1'

/fich2

HEAD

/rép1

/fich1

/fich2'

- *Index* : changements à apporter au prochain commit
- *HEAD* : commit d'où le work dir actuel est issu
- Initialisation nouveau répertoire : Index et HEAD vide
- Juste après un commit : index vide

Préparer un commit : définitions

Work dir	Index	HEAD
/rép1	/rép1	/rép1
/fich1	/fich1'	
/fich2	/fich2	/fich2'

Définitions (étant donné un fichier)

∈ **index** blob ds index, peut être ≠ WD

∈ **HEAD** blob ds HEAD, peut être ≠ WD

untracked [∉ index] et [∉ HEAD]

tracked [∈ index] ou [∈ HEAD] (ou les deux)

modified [∈ index ⇒ ≠ index] ; [∉ index ⇒ ≠ HEAD]

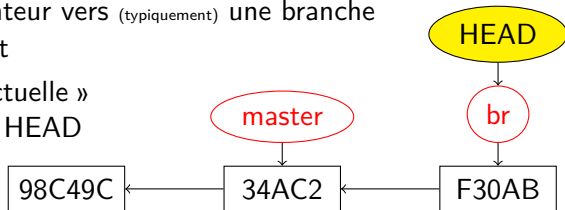
unmodified [∈ index ⇒ = index] ; [∉ index ⇒ = HEAD]

Préparer un commit : commandes

- `git add fichier` : blob mis dans index (« staged »)
- `clean WD` : tous (sauf fichiers dans `gitignore`) `tracked` et `unmodified`
- `git status` : liste `untracked`, `tracked-modified`, `staged`
- `git status --short` (sauf `merge conflict`) : `idx` VS `HEAD` ; `WD` VS `idx`.
- `git diff` : `WD` VS `index`
- `git diff --staged` : `index` VS `HEAD`
- `git commit` : commenter et expédier ! (Renvoie son id `SHA-1`)
- `git commit -v` : voir l'index en détail

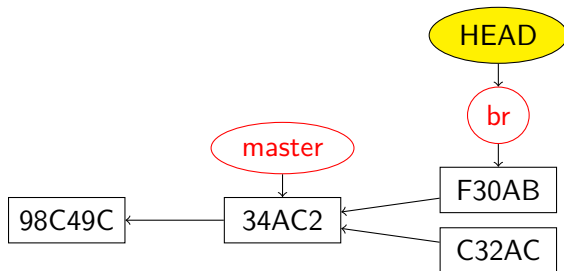
Branches et HEAD

- Branche : pointeur vers un commit
- HEAD : pointeur vers (typiquement) une branche et un commit
- Branche « actuelle » désignée par HEAD



- commit : avance HEAD et branche actuelle
- `git branch truc` : crée branche truc. HEAD inchangé !
- `git checkout truc` : change HEAD et met à jour WD
- Conseil : WD clean avant checkout !
- `git log --graph --decorate --oneline --all`

Fusion de branches



- `git merge autrebranche` : fusionne changements de autrebranche dans branche actuelle
- Si autrebranche est en avant de l'actuelle : « fast-forward »
- Sinon, « merge conflict » possible. Modifier les fichiers à la main et les ajouter à l'index puis commit pour créer un merge.
- `checkout d'un commit (ou tag) sans branche (detached head state)` : lecture !

Serveurs distants

- Réf. distante (« remote ref ») : pointeurs vers branches et tags sur dépôts distants
- « Remote-tracking branch » : branche locale correspondant à une branche distante et qui connaît l'état de la branche distante correspondante la dernière fois qu'on l'a vue
- Remote « origin » supposé configuré ici
- `git branch -vv` : voir branches et correspondants distants
- `git fetch` : récupère les commits distants ; met à jour (ou crée) les références distantes
- `git push origin mabranche` : sinon, nouvelles branches restent locales
- `git remote show origin` : voir les réf. distantes
- Suivre une branche distante : `checkout origin/branche` ; créer branche locale ; `git branch --set-upstream-to origin/branche`

Divers

- Utilisez gitignore ([modèles](#))
- Créez-vous une paire clé publique / privée
- Raccourcis : à éviter au début
- `git init` : dépôt vide dans rép. courant (rien n'est traqué)
- `git clone url` : cloner un dépôt (et non checkout!)
- `git stash` : WD \leftarrow HEAD
- `git tag -a montag` (tag annoté, recommandé) puis `git push origin montag`
- `git config --global` : écrit dans `~/.gitconfig`
- Indiquez propriété `user.name` (et `user.email`)
- Déterminer des [révisions](#) exemple : `HEAD~1` pour parent de HEAD
- [Alias](#)
- [Documentation](#)
- GUI pour diff : `git difftool`
- GUI pour merge : `git mergetool`

Références

- [Téléchargement](#) officiel
- [Livre](#) Pro Git
- [tryGit](#)

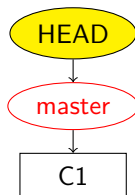
Exercices : Git I

Git en local

- Définir globalement (au moins) `user.name`. Vérifier avec `git config --list`.
- Créer un répertoire projet et dedans un fichier `début.txt` contenant "coucou".
- Initialiser un dépôt git dans ce projet.
- Placer `début.txt` dans l'index. Modifier `début.txt` pour qu'il contienne "coucou2". Visualiser la différence sur ce fichier entre la version WD, index, et dépôt. Faire en sorte que le blob dans l'index contienne bien "coucou2".

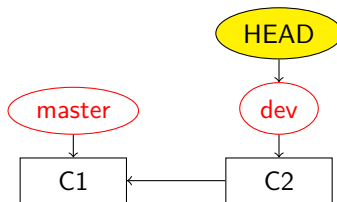
Exercices : Git II

- Effectuer un premier commit, qui contiendra uniquement `début.txt`. À l'issue de ce commit, vérifier que vous obtenez l'historique suivant.



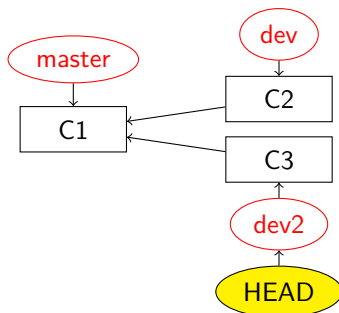
Exercices : Git III

- Vous avez maintenant une idée audacieuse pour résoudre un problème dans votre projet. Comme vous n'êtes pas sûr de sa pertinence, vous désirez placer vos changements dans une nouvelle branche en attendant d'y réfléchir. Créer une branche "dev" ; y commettre un fichier `audacieux.txt` (en plus de `début.txt`, inchangé) contenant "approche 1". Votre historique doit maintenant être celui-ci (vérifier!).



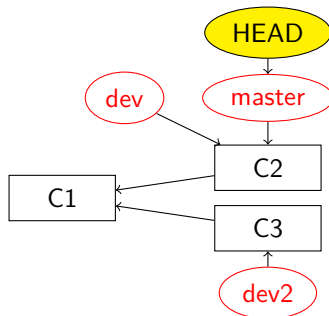
Exercices : Git IV

- À l'issue de ce travail harrassant, il vous vient une idée alternative. N'étant toujours pas sûr de la valeur de votre première idée (dans `dev`), vous repartirez de `master` pour l'implémenter. Depuis `master`, créer une branche `dev2`, et y commettre (en plus de `début.txt`, inchangé) un fichier `audacieux.txt` contenant "approche alternative". Vérifier ensuite votre historique.



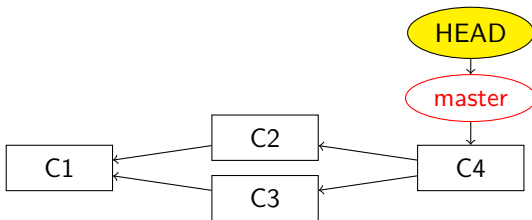
Exercices : Git V

- À la réflexion, votre première idée est bonne. L'intégrer dans `master` pour obtenir l'historique suivant. Prédire si vous obtiendrez un fast-forward et vérifier.



Exercices : Git VI

- Tout bien réfléchi vous aimez également votre deuxième idée. L'intégrer à son tour dans `master` et obtenir cet historique. Quel problème allez-vous rencontrer, ce faisant ?



- Imaginons qu'on aurait d'abord intégré `dev2` à `master` (ceci aurait-il produit un fast-forward ?) puis `dev` au résultat. Quel aurait été le résultat final ?

Git distant

- Cloner votre dépôt central pour le projet de ce cours.

Exercices : Git VII

- Le clonage vous a créé un pointeur vers un serveur distant `origin`, et une « remote-tracking branch » `master`. Voir où pointent `origin`, `master` et `origin/master`.
- Ajouter un fichier "`macontrib.txt`" contenant votre prénom à votre index local. Commettre dans votre dépôt git local. L'envoyer au dépôt distant. Vérifier (via l'interface web) qu'il s'y trouve et que le commit est associé à votre nom.
- Quand un collègue a fait de même, vous pouvez rapatrier sa modification en local. Après l'avoir fait, prédire où vont pointer `master` et `origin/master` et vérifier.
- Si un collègue a publié sa modification avant la vôtre, quel va être votre problème ? Comment le résoudre ? Si vous êtes le premier à publier la modification, allez aider vos collègues à publier la leur !

Présentation

- Apache Maven
- Outil de gestion de projet
- Principalement gestion de dépendances
- Description de votre projet via POM (Project Object Model)
- Convention over configuration : peu de configuration grâce aux valeurs par défaut
- Dépôt central avec publications open source
- Fortement basé sur plugins

Le POM

- Fichier XML
- Décrit un projet ou module et comment le construire (*build*)
- Un projet *peut* être composé de modules

Exemple de POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="..." xsi:schemaLocation="...">
  <modelVersion>4.0.0</modelVersion>
  <groupId>myGroupId</groupId>
  <artifactId>myArtifactId</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

Structure du projet

Structure déterminée normalement par
conventions

`/src/main/...` fichiers code et
ressources
« normales »

`/src/test/...` fichiers code et
ressources pour
tests

`.../java` code java

`.../resources` images, etc.,
devant être dans
classpath

```
/root
  /src
    /main
      /java
      /resources
    /test
      /java
      /resources
```


Accès aux ressources en Java

Accès aux ressources

(Ceci est indépendant de Maven)

- MyClass dans package com.mypkg
- URL `url = MyClass.class.getResource("ploum.txt")`
- `url` désigne la ressource de chemin `/com/mypkg/ploum.txt`
- Ensuite : `url.openStream()` (ou toute autre exploitation)

Positionnement de ressources avec Maven

Placer le fichier dans

`/src/main/resources/com/mypkg/ploum.txt`

Cycles de vie Maven

- Maven utilise des cycles de vie
- Cycles embarqués : default ; clean ; site
- Cycle : ensemble ordonné de *phases*
- Cycle “clean” contient essentiellement phase “clean”
- Cycle “site” contient essentiellement phase “site”
- Lors exécution de Maven, préciser une phase (Maven en déduit le cycle)

Cycle “default”

Phases (non exhaustif) dans cycle “default” :

`validate` valide informations du projet

`process-resources` copie vers destination

`compile` compilation du code source

`test` lancement des tests

`package` création d'un paquet

`integration-test` tests d'intégration

`verify` vérification de la validité du paquet

`install` installation en local

`deploy` déploiement dans dépôt configuré

Phases

- Chaque phase associée à un ensemble de plugins et d'objectifs (*goals*)
- Phase process-resources associée par défaut à plugin **Resources**, objectif resource
- Phase test associée par défaut à plugin **Surefire**, objectif test
- Phase package associée par exemple à plugin **JAR**, objectif JAR

Exécution

Lancement de Maven avec `mvn phasechoisie` :

- Maven détecte de quel cycle il s'agit
- Maven exécute toutes les phases jusqu'à "phasechoisie"
- Exemple : exécution systématique de test avant package

Dépendances

- Maven permet de gérer les « dépendances »
- Bibliothèques dont votre code dépend
- Pour compiler (dépendance statique) ; s'exécuter ; pour tests uniquement...
- Une bibliothèque dont vous dépendez peut elle-même avoir des dépendances
- Maven gère ces dépendances transitives pour vous !
- Dépendances prises par défaut dans Maven Central Repository
- Dans POM : section `<dependencies>`
- Dans cette section : ajouter une section `<dependency>` pour chaque dépendance à gérer

Dépendances : exemples

Exemple : dépendance vers Google Guava

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>19.0</version>
</dependency>
```

Exemple : dépendance vers junit

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Dépendances dans POM

- Trouver groupId et artifactId : voir site du projet
- Trouver version : voir [Central](#)
- Presque tous les projets Java récents font une release Maven

Portées (liste non exhaustive) :

`compile` Par défaut

`test` Bibliothèque incluse uniquement lors phase tests

`runtime` Bibliothèque incluse uniquement lors exécution, pas lors compilation

Configuration des plugins

- Voir [Liste](#) pour plugins de Apache
- Configuration parfois utile
- Dans POM : ajouter section

```
<build><plugins><plugin>...</plugin></plugins></build>
```
- Exemple : pour configurer la compilation, voir la page “Apache Maven Compiler Plugin”

Propriétés

- Propriété `ma propriété` : accessible via `${ma propriété}`
- Nommage souvent hiérarchique :
`catégorie.sous-catégorie.nom-propriété`

Dans POM :

```
<properties>  
  <cat.etc.prop1>valeur1</cat.etc.prop1>  
  <cat.etc.prop2>valeur2</cat.etc.prop2>  
</properties>  
(...)  
<balise-quelconque>${cat.etc.prop1}</balise-quelconque>
```

Conventions et configurations classiques

- Utiliser comme groupId un nom unique : généralement un nom de domaine inversé
- Le paquet de base de toutes les classes doit être ce nom
- Indiquer propriété `project.build.sourceEncoding` avec valeur UTF-8
- Configurer maven-compiler-plugin avec valeurs `source` et `target` à 1.8

Conventions pour ce cours

- Utiliser groupId :
`fr.dauphine.lamsade.hib.2016.nomapp`
- Utiliser artifactId : nomapp
- Canevas simple disponible [ici](#)

Installation

- Installer Java 1.8 pour ce cours
- Installer [Eclipse IDE for Java EE Developers](#) : contient maven embarqué
- Facultatif : [télécharger](#) et installer maven indépendamment d'eclipse

Maven et Eclipse

- M2Eclipse (m2e) fournit support Maven pour Eclipse
- Maven embarqué
- Wizards pour démarrer ou importer un projet maven
- Conseil : utiliser l'option Maven / Update project / Update project configuration from pom.xml pour configuration correcte du projet dans Eclipse

Références

- [Tutoriel](#) Apache Maven
- [Apache Maven Cookbook](#), Raghuram Bharathan, 2015

Omis dans cette présentation :

- Archetypes ([maven-archetype-plugin](#))
- Packaging
- Assembly ([maven-assembly-plugin](#))

Présentation

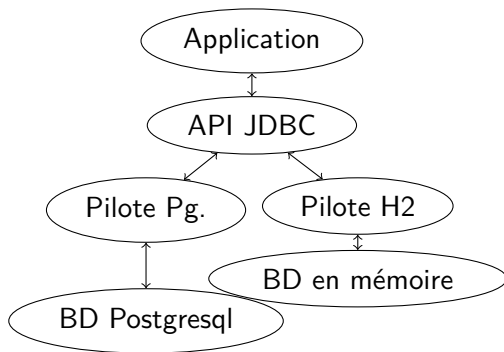
- JDBC ?

Présentation

- JDBC ? Java Database Connectivity

Présentation

- JDBC ? Java Database Connectivity
- Une API pour se connecter à des données relationnelles
- Programmation indépendante du fournisseur de BD
- App. programmée via API JDBC
- App. inclut pilotes du fournisseur
- Ces pilotes font la traduction



Instanciation

- Souhait : instancier pilote adéquat avec minimum de code spécifique à un fournisseur
- API JDBC nous fournit [DriverManager](#)
- Appeler `DriverManager.getConnection(String url)`
- url au format `jdbc:subprotocol:subname`
- Exemple : `jdbc:postgresql:mydb` (cf. [Doc JDBC Postgresql](#))

Mais comment ça marche ?

Fonctionnement de l'instanciation

- Le pilote fournisseur est inclus aux bibliothèques runtime de l'application
- Le JAR pilote inclut un fichier nommé (par convention) `META-INF/services/java.sql.Driver`
- Ce fichier nomme la classe que `DriverManager` doit charger
- `DriverManager` charge toutes ces classes (si plusieurs pilotes accessibles)
- Ayant l'URL, `DriverManager` cherche un pilote enregistré qui peut la lire
- Il instancie ce pilote et le renvoie à l'appelant ou l'utilise en arrière-plan

Remarques concernant l'instanciation

- Avec DriverManager on peut aussi obtenir le **Driver** (utile pour avoir n° de version par exemple)
- Beaucoup de tutoriels sur le net suggèrent d'enregistrer explicitement le pilote par exemple avec `Class.forName()`. Ce n'est plus nécessaire depuis longtemps (cf. explication précédente).

Exercices JDBC

Objectif : accéder à une BD en mémoire, à l'aide de H2

- Créer un projet Maven dans eclipse
- Indiquer H2 comme dépendance (cf. site [H2](#))
- Bien choisir la portée de votre dépendance
- Trouver l'URL JDBC à laquelle vous connecter (cf. site)
- Faites un simple `main` dans lequel vous vous connectez au pilote via JDBC et obtenez le numéro de version

Applications

- Application multi-utilisateur
- En Java SE : simulé par multiples instances en parallèle
- OU Java EE (au choix)
- À gérer de façon agile car fonctionnalités ajoutées au fil du cours : cycles courts, refactoring...
- Fonctionnalités de l'application décrites de manière vague : à vous de compléter
- Fin d'année : présentation collective de vos applications
- Vote pour la meilleure application

Bd locale d'annonces intéressantes (ads)

- Annonces collectées manuellement ou automatiquement
- Par exemple : recherche d'appartements, d'emploi
- Classement par attractivité subjective de chaque utilisateur
- Différents aspects à prendre en compte (taille appartement, proximité aux lieux d'intérêt, prix...)
- Utilité partielle associée à chaque aspect : un nombre entre 0 et 1
- Chaque utilisateur peut définir ses fonctions d'utilités partielles
- L'application montre les annonces attractives

Gestion de biblio collaborative (biblio)

- Solution à apporter : les bibliographies sont souvent incorrectes
- Entrées de bibliographies : livres ou musiques
- Chaque utilisateur peut commenter chaque entrée
- Chaque utilisateur peut confirmer ou informer les commentaires des autres
- Dès lors, un utilisateur a une réputation calculée
- Chaque utilisateur peut accorder sa confiance à d'autres utilisateurs (par priorité)
- L'application fournit donc une bibliographie personnalisée

Définir et analyser des QCMs (QCMs)

- Des enseignants peuvent définir des questions
- Des enseignants peuvent définir des réponses
- Des enseignants peuvent associer des points aux réponses (\neq points pour \neq enseignants)
- Un enseignant peut définir un QCM en choisissant des questions, des réponses, des coefficients
- Un enseignant peut définir un ensemble de questions à partir desquelles piocher des questions au hasard
- Un étudiant peut répondre au QCM
- Ou un groupe d'étudiants (points partagés pour le groupe)
- Un étudiant peut tenter de répondre seul ou demander de l'aide au groupe...

Élection du meilleur projet (elections)

- Un projet associé à un groupe de participants
- Un administrateur établit les règles de vote
- Chaque électeur (participant ou autre) peut voter pour un projet
- Ou ranger les projets
- L'administrateur peut interdire de voter pour son propre projet
- Option anonyme : l'identité des électeurs est oubliée

Comparaison de prédictions météo (predictions)

- Enregistrement des prédictions de différents sites
- Des utilisateurs indiquent la météo réelle
- Des utilisateurs peuvent confirmer les indications d'autres utilisateurs
- Calcul d'un degré de sincérité de chaque utilisateur
- Calcul de fiabilité des indications
- Calcul de fiabilité des prédictions

À faire

À faire *avant* le 3 mars

- Enregistrez-vous chacun dans un groupe sur [MyCourse](#)
- Min 3 membres, max 4 (pour 3 projets) (1^{ers} arrivés prioritaires)
- Nom groupe MyCourse = nom application = nom entre ()
- Créez un projet sur [GitHub](#) (ou [Bitbucket](#)) pour le groupe
- Nom projet git = nom application
- Ajoutez-moi à votre projet git
- Créez-vous chacun un compte sur le site en question
- Indiquez sur votre groupe MyCourse l'url git de votre projet

Avant le x signifie : avant qu'on soit le jour x. Donc au plus tard la veille à 23h59 + 1 minute.

À faire : cours suivant

À faire *avant* le 16 mars

- Projet eclipse (configuration sur serveur git)
 - Dépendance Maven : H2
 - Le prg affiche le numéro de version du pilote, puis quitte
 - À faire par chaque membre de chaque groupe
 - Envoyez au git de votre groupe dans rép. `user.name`
-
- Choisir un nom d'utilisateur *une fois pour toutes* et l'indiquer dans `user.name`
 - Indiquer sur votre groupe MyCourse le lien entre identité réelle et `user.name` (si non évident)
 - Je considère que l'auteur du commit (tel qu'indiqué par git) est l'auteur du travail

Licence

Cette présentation, et le code LaTeX associé, sont sous [licence MIT](#). Vous êtes libres de réutiliser des éléments de cette présentation, sous réserve de citer l'auteur.

Le travail réutilisé est à attribuer à [Olivier Cailloux](#), Université Paris-Dauphine.

(Ceci ne couvre pas les images incluses dans ce document, puisque je n'en suis généralement pas l'auteur.)