

TP - Deploy an app to Heroku

🎯 Objectifs :

- ❑ Découvrir le déploiement d'une application dans le Cloud
- ❑ Tester plusieurs technologies

Le TP est conçu pour être exécuté sur un environnement Linux. Si vous avez un autre OS, adaptez les commandes.

Présentation d'Heroku



Heroku is a cloud platform based on a managed container system, with integrated data services and a powerful ecosystem, for deploying and running modern apps. The Heroku developer experience is an app-centric approach for software delivery, integrated with today's most popular developer tools and workflows.

Les avantages :

- ❑ déploiement via Git
- ❑ Système de buildpacks ([pour plus d'informations](#)) permettant de gérer les particularités de différents types de langages/frameworks
- ❑ Système d'add-on intéressant
- ❑ Compte gratuit pour une utilisation limitée des ressources et du temps, parfait pour faire des petits prototypes (et pourquoi pas un projet d'AIOP ?)

Les inconvénients :

- ❑ Peut devenir rapidement chère pour une utilisation avec un fort trafic ou de fortes contraintes de bases de données

- ❑ [Lab 1 : Mise en place](#)
- ❑ [Lab 2 : Déploiement](#)
- ❑ [Lab 3 : Configuration de l'application](#)
- ❑ [Lab 4 : Lancement de l'application](#)
- ❑ [Lab 5 : Modifier et redéployer l'application](#)
- ❑ [Lab 6 : Executer des commandes](#)
- ❑ [Lab 7 : Définir les variables de configuration](#)
- ❑ [Lab 8 : Utiliser un add-on](#)
- ❑ [Lab 9 : Utiliser une base de données](#)
- ❑ [Lab 10 : Jouer avec différentes technologies](#)
- ❑ [Lab 11 : Pour aller plus loin : OAuth](#)

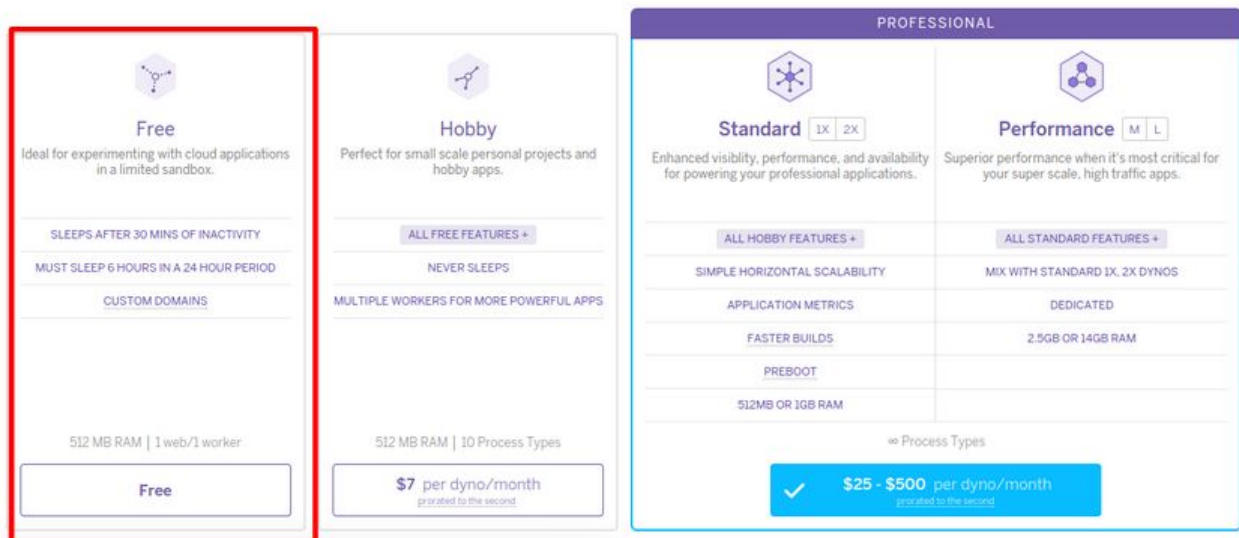
LAB 1 - Mise en place

Objectifs

- ☐ S'inscrire sur la plateforme heroku (utilisez votre adresse EPSI si vous voulez).
- ☐ Préparer l'environnement
- ☐ récupérer le projet exemple

S'inscrire sur la plateforme Heroku

Heroku possède différentes formule de souscription :



Free	Hobby	Professional (Standard)	Professional (Performance)
Free Ideal for experimenting with cloud applications in a limited sandbox.	Hobby Perfect for small scale personal projects and hobby apps.	Standard (1X, 2X) Enhanced visibility, performance, and availability for powering your professional applications.	Performance (M, L) Superior performance when it's most critical for your super scale, high traffic apps.
SLEEPS AFTER 30 MINS OF INACTIVITY MUST SLEEP 6 HOURS IN A 24 HOUR PERIOD CUSTOM DOMAINS	ALL FREE FEATURES + NEVER SLEEPS MULTIPLE WORKERS FOR MORE POWERFUL APPS	ALL HOBBY FEATURES + SIMPLE HORIZONTAL SCALABILITY APPLICATION METRICS FASTER BUILDS PREBOOT 512MB OR 1GB RAM	ALL STANDARD FEATURES + MIX WITH STANDARD 1X, 2X DYNOS DEDICATED 2.5GB OR 14GB RAM
512 MB RAM 1 web/1 worker Free	512 MB RAM 10 Process Types \$7 per dyno/month <small>prorated to the second</small>	∞ Process Types \$25 - \$500 per dyno/month <small>prorated to the second</small>	

Aujourd'hui, nous allons utiliser la possibilité offerte par heroku de compte gratuit. Pour ce faire, il faut créer un compte gratuit [ici](#)



Attention : il ne vous est pas demandé votre numéro de carte ou toute autre information de prélèvement. Si tel est le cas, vous vous êtes trompé ne poursuivez pas.

Préparer l'environnement (TODO)

Pour mener à bien ce TP, nous allons utiliser quatre outils :

- ☐ Git
- ☐ Java 8
- ☐ Maven 3

❏ Heroku CLI : <https://devcenter.heroku.com/articles/heroku-command-line>

Récupérer le projet exemple

Pour ce faire taper les commandes suivantes :

```
1. $> cd votre_repertoire_a_projet  
2. $> git clone https://github.com/fteychene/heroku-epsi-cloud  
3. $> cd heroku-epsi-cloud
```

Le répertoire contient une application java basique et un fichier pom.xml utilisé par Maven pour gérer les dépendances de l'application et le cycle de vie de l'application.

LAB 2 - Déploiement

Objectifs

- ☐ Déployer l'application
- ☐ Vérifier le bon fonctionnement
- ☐ Analyser les journaux d'évènements

Déployer l'application

Tout d'abord, il est nécessaire d'indiquer à Heroku que nous voulons déployer une nouvelle application. Pour ce faire, il faut exécuter la commande suivante :

```
$> heroku create
```

Cette commande crée un dépôt git distant sur la plateforme Heroku. Ce dépôt est associé à votre dépôt local sous le terme **heroku**.

Heroku génère un nom aléatoirement pour votre application (vous pourrez changer ce nom par la suite). Vous pouvez aussi passer le nom spécifique que vous voulez lors de la création (e.g. *heroku create lapin*)

Pour déployer l'application il vous suffit de soumettre le code source sur le dépôt distant :

```
$> git push heroku master
```

Vérifier le bon fonctionnement de l'application

L'application est maintenant déployée. Vérifiez qu'il y a au moins une version de l'application qui s'exécute :

```
$> heroku ps:scale
```

Pour ouvrir l'application taper :

```
$> heroku open
```

Le navigateur par défaut de votre ordinateur va directement ouvrir l'application distante.



Checkpoint notation

Analyser les journaux d'évènements

Heroku traite les journaux d'évènements (logs) comme un flux d'évènements agrégés de tous vos composants déployés.

Pour voir les évènements survenant en direct dans votre application, taper :

```
$> heroku logs --tail
```

Pour écrire des informations dans les logs, il vous suffit d'utiliser une des deux sorties standard de Java, soit directement :

```
System.err.println("Hello, logs!");  
System.out.println("Hello, logs!");
```

Soit en utilisant un système de log de java (ce qui est ce que vous aviez prévu de faire n'est ce pas ? :)).

Pour de plus amples informations <https://devcenter.heroku.com/articles/logging>

LAB 3 - Configuration de l'application

Objectifs

- ☐ Explication du fichier "ProcFile"
- ☐ Information sur les dynos
- ☐ Gestion de dépendances et compilation de l'application

Explication du fichier "ProcFile"

Le ProcFile est un fichier texte qui doit se trouver à la racine de l'application pour expliquer à Heroku les commandes qu'il doit exécuter au démarrage de l'application.

Le fichier Procfile de l'application exemple que vous avez déployé contient les informations suivantes :

web: `java -jar target/java-getting-started-1.0.jar`

Ce fichier déclare un seul type de processus, web, et la commande nécessaire pour le lancer. Le terme **web** est important. Il permet à la plateforme Heroku de comprendre que ce processus doit se voir attacher le routeur HTTP d'Heroku et le trafic web qui y est associé.

Il existe différents types de processus, par exemple vous pouvez déclarer un processus en tâche de fond qui consomme le contenu d'un système de files d'attente.

Pour plus d'informations sur le ProcFile : <https://devcenter.heroku.com/articles/procfile>

Information sur les dynos

Actuellement, votre application utilise un seul **dyno**. Un dyno peut être vu comme un conteneur léger qui exécute la commande spécifiée dans le ProcFile. Vous pouvez vérifier le nombre de dyno entrain de s'exécuter avec la commande ps :

```
$>heroku ps
```

Par défaut, votre application est déployée sur un "free dyno". Les free dynos rentrent en hibernation au bout de 30 minutes sans sollicitations et ne peuvent pas rester allumés plus de 18 heures par jour. Si un free dyno devient inactif mais n'a pas encore atteint son quota (18

heures) d'activité journalière, n'importe quelle requête web va le réveiller. Cela peut entraîner un délai de quelques secondes entre la première requêtes et les requêtes suivantes (le temps du réveil).

Gestion de dépendances et compilation de l'application

Heroku détecte l'application comme étant une application Java grâce à la présence d'un fichier pom.xml dans le répertoire racine.

L'application de démo contient un pom.xml comme ceci :

```
<dependencies>
  <dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-core</artifactId>
    <version>2.2</version>
  </dependency>
  <dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-template-freemarker</artifactId>
    <version>2.0.0</version>
  </dependency>
  ...
</dependencies>
```

Le fichier pom.xml spécifie les dépendances qui doivent être installée avec votre application. Lorsque l'application est déployé sur Heroku ce dernier lit le fichier pom.xml et installe les dépendances grâce à la commande mvn clean install.

Le fichier **system.properties** détermine la version de java à utiliser :

java.runtime.version=1.8

Taper la commande ci-dessous pour construire l'application en local :

```
$> mvn clean install
```



Checkpoint notation

LAB 4 - Lancement de l'application en local

Objectifs

- ☐ Lancer l'application localement
- ☐ Simuler Heroku localement

Lancer l'application localement

L'application de démonstration peut être lancée comme toute autre application localement sur votre poste avec les outils habituels.

Simuler Heroku localement

Pour lancer l'application localement de la même manière que sur Heroku, il suffit de lancer la commande suivante :

```
$> heroku local web
```

Le programme va parcourir le fichier ProcFile pour déterminer les actions à opérer (comme sur le cloud Heroku). Votre application est disponible sur <http://localhost:5000>. Pour arrêter l'application, appuyez sur Ctrl-C.

 heroku local ne fait pas que lancer votre application ; il met aussi en place les variables d'environnement qui seront vues plus loin dans le TP.

LAB 5 - Modifier et redéployer l'application

Objectifs


- ☐ Utiliser le gestionnaire de dépendances
- ☐ Redéployer une application

L'objectif est de vous apprendre à propager une modification de votre application. Pour ce faire nous allons utiliser l'exemple de la gestion de dépendances.

Ajouter une dépendance

Nous allons ajouter une dépendance supplémentaire à l'application. Pour ce faire, modifier le fichier pom.xml pour y inclure la dépendance jscience. Ajouter à la section <dependencies> les informations suivantes :

```
<dependency>
  <groupId>org.jscience</groupId>
  <artifactId>jscience</artifactId>
  <version>4.3.1</version>
</dependency>
```

 Le triplet **groupId**, **artifactId** et **version** permettent d'identifier de manière unique ce que l'on appelle un artefact (dans notre cas un jar). Ici nous demandons à maven de télécharger la librairie jscience de org.science en version 4.3.1.

Utiliser la librairie jscience

Modifier le fichier **src/main/java/com/example/Main.java**:

1. Ajouter les imports suivant en haut du fichier :

```
import static javax.measure.unit.SI.KILOGRAM;
import javax.measure.quantity.Mass;
import org.jscience.physics.model.RelativisticModel;
import org.jscience.physics.amount.Amount;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestParam;
```

2. Ajouter une méthode pour rajouter une route :

```
@RequestMapping(value = "/hello")
@ResponseBody
String hello() {
    RelativisticModel.select();
    Amount<Mass> m = Amount.valueOf("12 GeV").to(KILOGRAM);
    return "E=mc^2: 12 GeV = " + m.toString();
}
```

3. Tester localement les modifications :

```
$> mvn clean install
$> heroku local web
```

Visiter la page <http://localhost:5000/hello> et vérifier que vous voyez :
E=mc^2: 12 GeV = (2.139194076302506E-26 ± 1.4E-42) kg

Redéployer une application

Redéployer une application consiste simplement à **pousser** les nouveaux commits Git que vous avez effectués. Le déroulement est le suivant :

1. effectuer du travail, en faire des commits sur git (git add ... ; git commit ...)
2. lorsqu'une version correspond à ce que vous voulez deployer sur le cloud, pousser les modifications

Effectuer un déroulement basique :

1. Ajouter des modifications du travail

```
$> git add .
```

2. Réalisation d'un commit

```
$> git commit -m "Démo"
```

3. Pousser les modifications sur Heroku

```
$> git push heroku master
```

4. Vérifier le bon fonctionnement de votre application

```
$> heroku open
```



Vous pouvez aussi avoir une approche par feature branche ou pousser un commit qui

n'est pas le plus récent sur Heroku.



Checkpoint notation

LAB 6 - Exécuter des commandes



Objectifs

- ☐ Apprendre à exécuter des commandes

Exécuter des commandes sur un dyno

Vous pouvez lancer des commandes, par exemple des scripts shell, dans un dyno (c'est un dyno particulier voir [one-off dyno](#) pour plus d'informations).

Pour se faire il suffit d'utiliser la commande **heroku run <command>**

Lancer :

```
$> heroku run bash  
$> java -version
```



Ne pas oublier de taper exit our quitter le shell et mettre fin à l'exécution du dyno.

LAB 7 - Définir les variables de configuration

Objectifs

- ❑ Définir les variables d'environnement

Heroku permet de définir une configuration externalisée. A l'exécution, la configuration sera exposée sous forme de variables d'environnement à l'application.

1. Modifier le fichier **src/main/java/Main.java** pour qu'il récupère une variable *energy* depuis la variable d'environnement **ENERGY** :

```
@RequestMapping(value = "/conversion")
@ResponseBody
String conversion() {
    RelativisticModel.select();
    String energy = System.getenv().get("ENERGY");

    Amount<Mass> m = Amount.valueOf(energy).to(KILOGRAM);
    return "E=mc^2: " + energy + " = " + m.toString();
}
```

2. Compiler l'application

```
$> mvn clean install
```

3. Modifiez le fichier .env contenant la ligne suivante :
ENERGY="40 GeV"



Pour créer un fichier en ligne de commande et y insérer de l'information, vous pouvez utiliser différents logiciels dont nano ou vi.

Exemple : *nano nomdufichier*

4. lancer l'application en local grâce à la commande heroku

```
$> heroku local
```

5. Visiter l'adresse <http://localhost:5000/conversion>

La page affiche la conversion pour 40 GeV

Lorsque vous démarrez votre application en utilisant la commande **heroku local**, heroku lit le fichier .env présent à la racine du projet et insert comme variable d'environnement toutes les paires clefs/valeurs qu'il y a trouvé.

Maintenant que vous avez vérifié que cela fonctionne correctement en local, créer la variable

de configuration sur Heroku en exécutant la commande suivante :

```
$> heroku config:set ENERGY="20 GeV"
```

Pour vérifier que la variable a bien été ajoutée à la configuration :

```
$> heroku config
```

Déployer les changements que vous avez effectué sur Heroku et vérifier le bon fonctionnement avec `heroku open`.

Pour plus d'informations sur le fonctionnement d'Heroku local [cliquer ici](#)



Checkpoint notation

LAB 8 - Utiliser un add-on (ne pas effectuer ce lab seulement le lire)

Objectifs

- ☐ Ajouter une extension à Heroku

Les add-ons sont des services cloud tierces (third-party cloud services) qui fournissent des services supplémentaires “out-of-the-box” pour votre application, de la persistance de données , de l’exploitation de logs, du monitoring, etc.

Par défaut, Heroku stocke 1500 lignes de log de votre application. Cependant, il rend le flux du journal complet disponible en tant que service - et plusieurs fournisseurs de services d'exploitation ont écrits des add-ons qui fournissent des choses telles que la persistance du journal, la recherche, et le courrier électronique ou l’alerte SMS quand certaines conditions sont remplies.

Ajout du service papertrail :

Pour éviter les abus, Heroku requière une vérification de compte pour installer des add-ons. Visiter le [verification site](#) afin de valider votre compte. **La vérification implique le fait de renseigner son numéro de carte bancaire. NE LE FAITE PAS DANS LE CADRE DE CE TP. La suite de ce lab est écrite à titre indicatif.**

Papertrail est un service de gestion des journaux d’évènement pour l’ajouter à votre application taper la commande :

```
$> heroku addons:create papertrail
```

Pour lister la liste des add-ons de votre application, taper :

```
$> heroku addons
```

Pour voir le fonctionnement de cet add-on, accéder plusieurs fois à votre application. Chaque visite va générer plus d’évènements qui devraient être acheminé sur papertrail. Pour voir la console paper trail et lire les logs taper :

```
$>heroku addons:open papertrail
```

Une console va s’ouvrir, montrant les derniers évènements, et fournissant une interface de

recherche et de configuration d'alertes :

```
Aug 08 07:20:50 warm-eyrie-9006 heroku/router: at=info method=GET path="/" host=warm-eyrie-9006.herokuapp.com request_id=a396a7dc-41d4-4fda-ab66-225262711f43  
fwd="94.174.204.242" dyno=web.1 connect=1ms service=21ms status=200 bytes=605  
Aug 08 07:20:50 warm-eyrie-9006 heroku/router: at=info method=GET path="/favicon.ico" host=warm-eyrie-9006.herokuapp.com request_id=e072bd72-8163-4cc4-9bcc-  
8eb05d387034 fwd="94.174.204.242" dyno=web.1 connect=1ms service=3ms status=200 bytes=519  
Aug 08 07:22:11 warm-eyrie-9006 heroku/router: at=info method=GET path="/" host=warm-eyrie-9006.herokuapp.com request_id=67308c79-07eb-4131-a5fd-5b32b1c60488  
fwd="94.174.204.242" dyno=web.1 connect=1ms service=5ms status=200 bytes=605  
Aug 08 07:22:11 warm-eyrie-9006 heroku/router: at=info method=GET path="/favicon.ico" host=warm-eyrie-9006.herokuapp.com request_id=41e97e9d-45c0-41c6-93ca-  
6c9c7182f401 fwd="94.174.204.242" dyno=web.1 connect=1ms service=4ms status=200 bytes=519
```


LAB 9 - Utiliser une base de données

Objectifs

- ☐ Utiliser des variables d'environnements
- ☐ Communiquer avec une base de données

Heroku est doté d'une marketplace pour les add-on (extensions). Cette dernière contient un grand nombre de systèmes de stockages de données de Redis à MongoDB en passant par Postgres et Mysql.

Dans ce lab nous allons utiliser l'add-on gratuit Postgres pour Heroku. Cet add-on est déjà installé par défaut sur les comptes gratuits (mais avec une base de données limitée).

Pour savoir les add-on présents dans votre application taper :

```
$> heroku addons
```

Voir la connexion vers la base postgres

En regardant la configuration de votre application, vous trouverez la connexion vers votre base de données sous la variable d'environnement DATABASE_URL :

```
$> heroku config
```

Heroku fournit aussi une commande pg qui donne plus d'informations :

```
$> heroku pg
== HEROKU_POSTGRESQL_BLUE_URL (DATABASE_URL)
Plan:          Hobby-dev
Status:        Available
Connections:    0
PG Version:    9.3.3
Created:        2014-08-08 13:54 UTC
Data Size:     6.5 MB
Tables:        0
Rows:          0/10000 (In compliance)
Fork/Follow:   Unsupported
Rollback:      Unsupported
```

Cela indique que le plan utilisé est hobby-dev (gratuit), tournant sur Postgres 9.3.3
L'application exemple que vous avez déployé a déjà une base de données fonctionnelle qui peut être atteint via /db de votre application.

Vous pouvez vous connecter sur votre application connecté sur la page /db pour voir que la connection marche.

Modifier le code de la page DB

En prenant en compte que il est possible de récupérer un query param *from* avec une valeur par défaut *unknow* en rajoutant un paramètre à la méthode *db* en utilisant le code suivant :

```
... (... , @RequestParam(name = "from", defaultValue = "unknow") String from, ...)
```

Changez le code de la fonction *db* pour qu'elle :

1. créer une table ask avec une colonne tick de type timestamp et une colonne *usr* de type *varchar*.
2. Insère à chaque appel à la page *db* une ligne dans la table ask avec le timestamp courant et un user qui est le query param *unknow* si non définit.
3. Charge la liste des tuples de la table ask
4. Affiche le résultat sur le modèle

Read from DB: 2017-11-05 19:10:32.553775 asked by unknow

Read from DB: 2017-11-05 19:10:43.326118 asked by fteychene

Déployer la modification sur Heroku

Pour déployer les changements, sauvegarder vos modifications dans git puis taper la commande :

```
$> git push heroku master
```

Accéder à la route /db en mettant un paramètre from et en l'enlevant vous devriez voir quelques chose comme ceci :

Database Output

* Read from DB: 2017-11-05 19:10:32.553775 asked by unknow

* Read from DB: 2017-11-05 19:11:10.873736 asked by test

* Read from DB: 2017-11-05 19:13:33.220346 asked by unknow



Checkpoint notation

Accéder à distance à la base Postgres grâce à Heroku

A partir du moment où vous avez le pgclient d'installé (il est installé en même temps que la base de données), vous pouvez utiliser la commande **heroku pg:psql**. Cette commande vous permet de vous connecter à votre base de données distante. Taper :

```
$> heroku pg:psql
heroku pg:psql
psql (9.3.2, server 9.3.3)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.
=> SELECT * FROM ticks;
      tick
-----
2014-08-08 14:48:25.155241
2014-08-08 14:51:32.287816
2014-08-08 14:51:52.667683
2014-08-08 14:51:53.1871
2014-08-08 14:51:54.75061
2014-08-08 14:51:55.161848
2014-08-08 14:51:56.197663
2014-08-08 14:51:56.851729
(8 rows)
=> \q
```

Pour plus d'information sur PostgreSQL sur Heroku: [Heroku PostgreSQL](#).

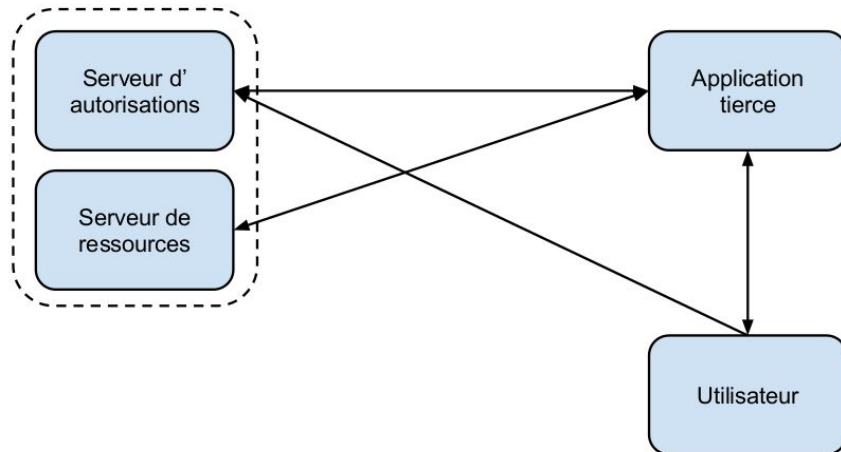
Une technique similaire peut être utilisée pour installer les add-ons pour [MongoDB](#) ou [Redis](#).

LAB 10 - Pour aller plus loin : OAuth

Objectifs

- ☐ Initiation à OAuth

Rappel OAuth



OAuth est un protocole de délégation d'autorisation d'accès à des APIs.

Les avantages :

- ☐ l'application tierce ne connaît pas le mot de passe de l'utilisateur;
- ☐ son accès à l'api est restreint et validé par l'utilisateur;
- ☐ l'utilisateur peut révoquer l'accès de l'application tierce.

Connecter une application avec le serveur OAuth2 d'Heroku

Heroku a sa propre implémentation d'un serveur OAuth2 valide. Bien que dans le cadre de vos projets vous aurez à vous authentifier auprès de serveurs OAuth 2 majeurs tels que celui de google, ce tutoriel peut vous faire une bonne introduction :

<https://devcenter.heroku.com/articles/oauth>

LAB 11 - Jouer avec différentes technologies

Objectifs

- ☐ Tester de nouvelles technologies
- ☐ Déployer différentes technologies dans le cloud

Choisissez les technologies que vous souhaitez tester dans la liste ci-dessous et deployez les dans le cloud d'Heroku.

Tester Spring MVC + Hibernate

Une architecture classique de déploiement dans le monde java est l'utilisation de l'écosystème Spring en conjonction du framework d'ORM (Object Relational Mapping) Hibernate.

Dans ce lab votre objectif est de déployer une application d'exemple dans Heroku puis de jouer avec pour mieux appréhender la technologie.

- ☐ Documentation de Spring
- ☐ Réaliser ce tutoriel :
<https://devcenter.heroku.com/articles/getting-started-with-spring-mvc-hibernate>

Tester avec [Jax-RS](#)

Jax-RS est la technologie que vous avez utilisé lors du lab 8 du TP#1.

- ☐ Exemple à déployer : <https://github.com/heroku/template-java-jaxrs>

Tester avec [Play!Framework](#)

“Play is a high-productivity Java and Scala web application framework that integrates the components and APIs you need for modern web application development.

Play is based on a lightweight, stateless, web-friendly architecture and features predictable and minimal resource consumption (CPU, memory, threads) for highly-scalable applications thanks to its reactive model, based on Iteratee IO.”

Play!1 est un framework reconnu pour sa rapidité de développement; Play2 est le framework de référence pour faire du web en Scala, bien qu'utilisable en Java. Il est utilisé par de nombreuses entreprises prestigieuses grâce à ses qualités (lire la document pour en savoir plus).

- ☐ Play!1 en java (facile)

- ❑ <https://devcenter.heroku.com/articles/getting-started-with-play>
- ❑ Play!2 en Scala (peut aussi être utilisé avec Java) - attention plus difficile
- ❑ <https://devcenter.heroku.com/articles/getting-started-with-scala#introduction>

Tester avec [Grails](#)

“**Grails** is a **powerful** web framework, for the Java platform aimed at multiplying developers’ productivity thanks to a Convention-over-Configuration, sensible defaults and opinionated APIs. It integrates smoothly with the JVM, allowing you to be immediately productive whilst providing powerful features, including integrated ORM, **Domain-Specific Languages**, runtime and compile-time **meta-programming** and **Asynchronous** programming.”

Grails est un framework de développement rapide. Il associe ensemble plusieurs technologies clefs (Spring, Hibernate, etc.) et offre une approche convention over configuration ce qui permet de masquer beaucoup de détails liés à la configuration de ses frameworks au développeur. Ce framework utilise pour ses templates des GSP (Groovy Server Pages) qui à la différence des JSP utilisent Groovy comme langage.

- ❑ Tutoriel : <https://devcenter.heroku.com/articles/getting-started-with-grails>
- ❑ Grails getting started : <https://grails.org/documentation.html#gettingstarted>

Tester avec [DropWizard](#)

“Dropwizard is a Java framework for developing ops-friendly, high-performance, RESTful web services.

Dropwizard pulls together stable, mature libraries from the Java ecosystem into a simple, light-weight package that lets you focus on getting things done.

Dropwizard has out-of-the-box support for sophisticated configuration, application metrics, logging, operational tools, and much more, allowing you and your team to ship a production-quality web service in the shortest time possible.”

DropWizard a pour objectif d’assembler de manières cohérentes des technologies classique de Java afin de pouvoir fournir un framework de développement rapide orienté RESTful.

- ❑ Exemple et explications : <https://github.com/alexroussos/dropwizard-heroku-example>
- ❑ DropWizard Getting started : <http://www.dropwizard.io/getting-started.html>