Cursus JAVA

M2I Formations 2020

Christophe Guéroult
Xavier Tabuteau
Nicolas Haziza
Marc Gilbert
Olivier Blaivie

1.

Présentation

Objectifs de la formation

- Etre capable de faire une démonstration de l'application CRM en ligne.
- Etre capable de montrer le découpage du projet et la gestion de projet sur Jira.
- Être capable de présenter le projet (prototypes, designs, spécifications fonctionnelles et détaillées)
- Etre capable d'expliquer le code (Back et Front) et de justifier les choix (bdd, technos, langages, api et intégration continue avec Jenkins).
- Être capable d'expliquer les cycles de livraisons et la mise en production.
- Être capable de parler des notions présentes dans chaque plan de cours et pour chaque module.
- Etre capable d'expliquer les processus de montée en compétence appliqués durant la formation (pair programming, peer correcting et revues de code).
- Montrer aux futurs employeurs que vous êtes aptes à intégrer des projets d'entreprise, que vous êtes à l'aise avec les processus, technos et outils, que vous avez acquis les bases pour une montée en compétence à la fois en autodidacte et en entreprise.

Cursus Java

Mettre en place le côté « Back » d'une application

- Utiliser le langage **Java**
- Savoir mettre en place des **Tests Unitaires** (TU)
- Savoir mettre en place une **connexion** avec une base de données
- Comprendre et mettre en œuvre **JEE** (Java Entreprise Edition)
- Comprendre et mettre en œuvre **JSF** (Java Server Faces) → **MVC**
- Comprendre et mettre en œuvre **JPA** (Java Persistence API)
- Comprendre et mettre en œuvre les **Web Services** (SOAP et REST)
- Comprendre et mettre en œuvre **Spring**

Cursus Java

Utiliser des outils

- Environnement de développement (IDE) : **Eclipse**
- Base de données : **PostgreSQL**
- Utilitaire de gestion de projet Java : **Maven**
- Serveur d'application : **Tomcat**
- Utilitaire de test des Web Services : **SoapUI** et **Postman**
- Conteneurisation : **Docker**

Utiliser des « frameworks »

- Accès aux données
- Tests
- Web
- Utilitaires
- …

Cursus Java

Un projet fil rouge : l'application CRM

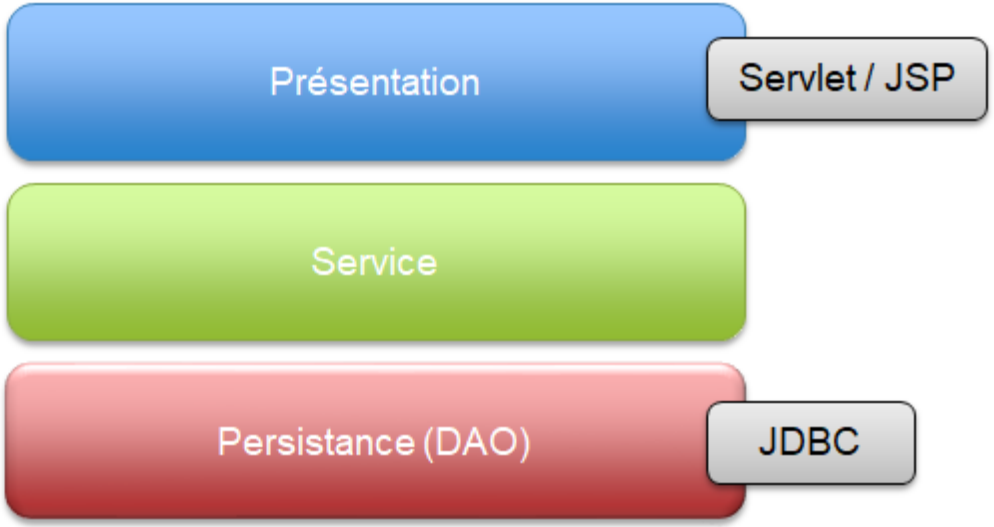
- Gestion de **Clients**
- Gestion de **Commandes**
- Liens entre les **Clients** et les **Commandes**
- Gestion des utilisateurs de l'application

→ En s'appuyant sur la base de données créée dans le module précédent

Cursus Java

Un projet fil rouge : l'application CRM

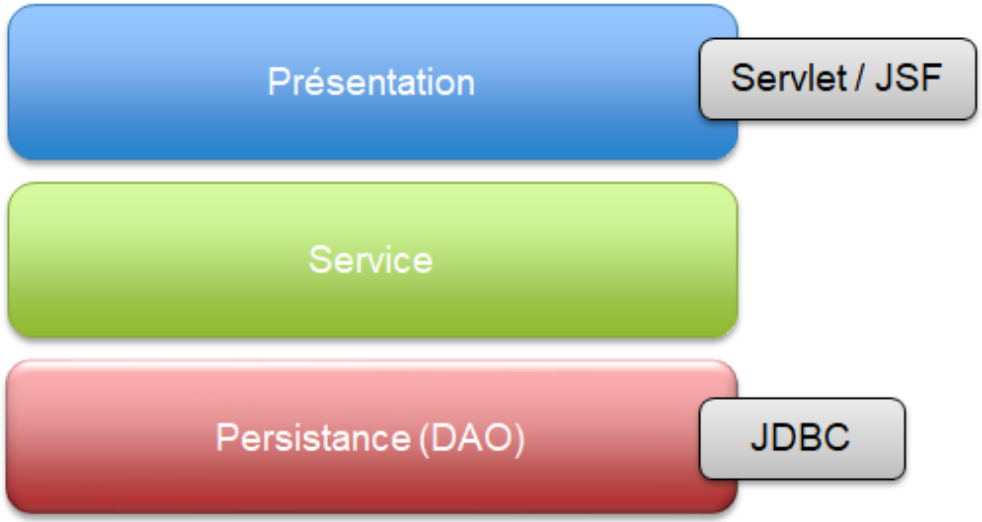
Etape 1



Cursus Java

Un projet fil rouge : l'application CRM

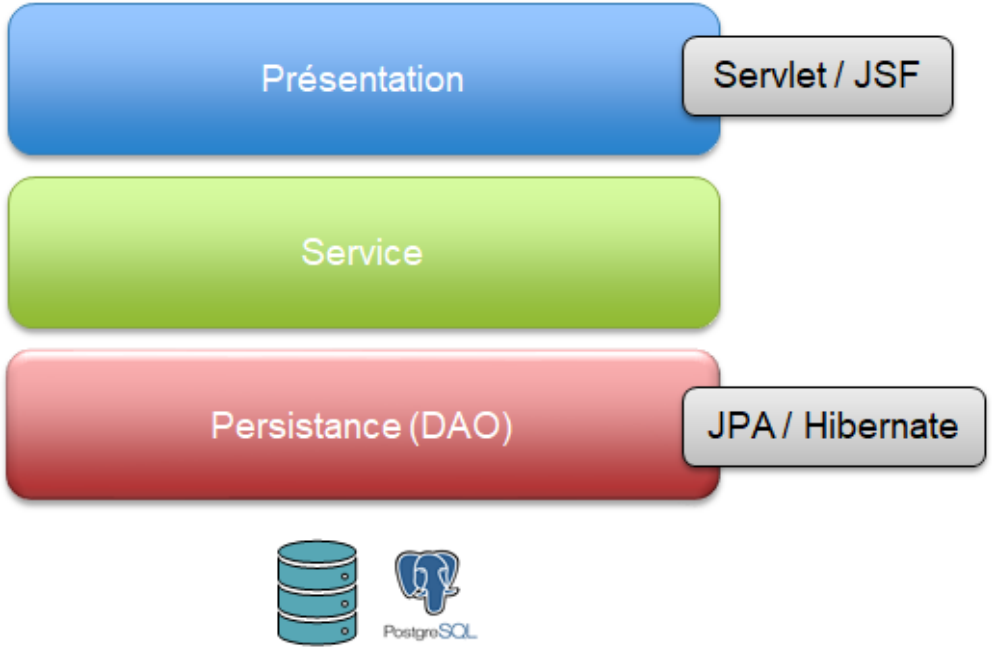
Etape 2



Cursus Java

Un projet fil rouge : l'application CRM

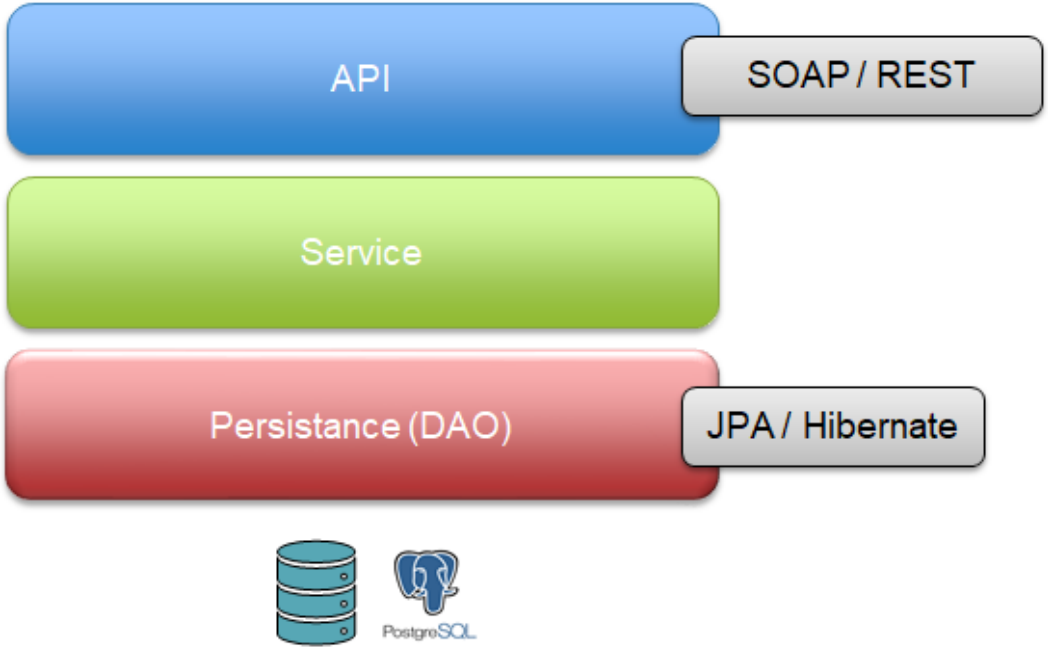
Etape 3



Cursus Java

Un projet fil rouge : l'application CRM

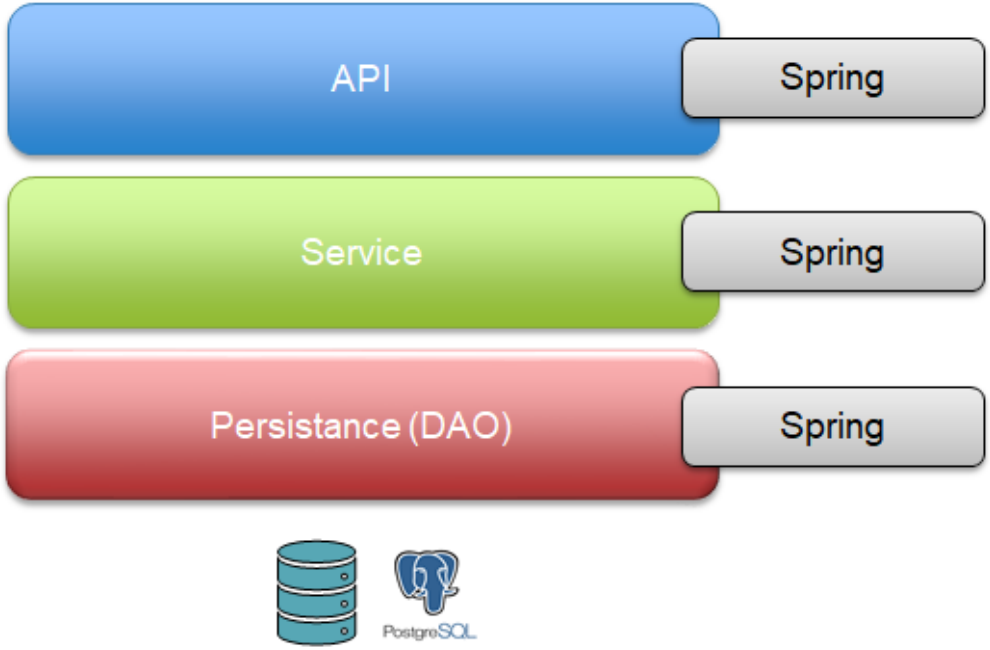
Etape 4



Cursus Java

Un projet fil rouge : l'application CRM

Etape 5



MODULE AGILE

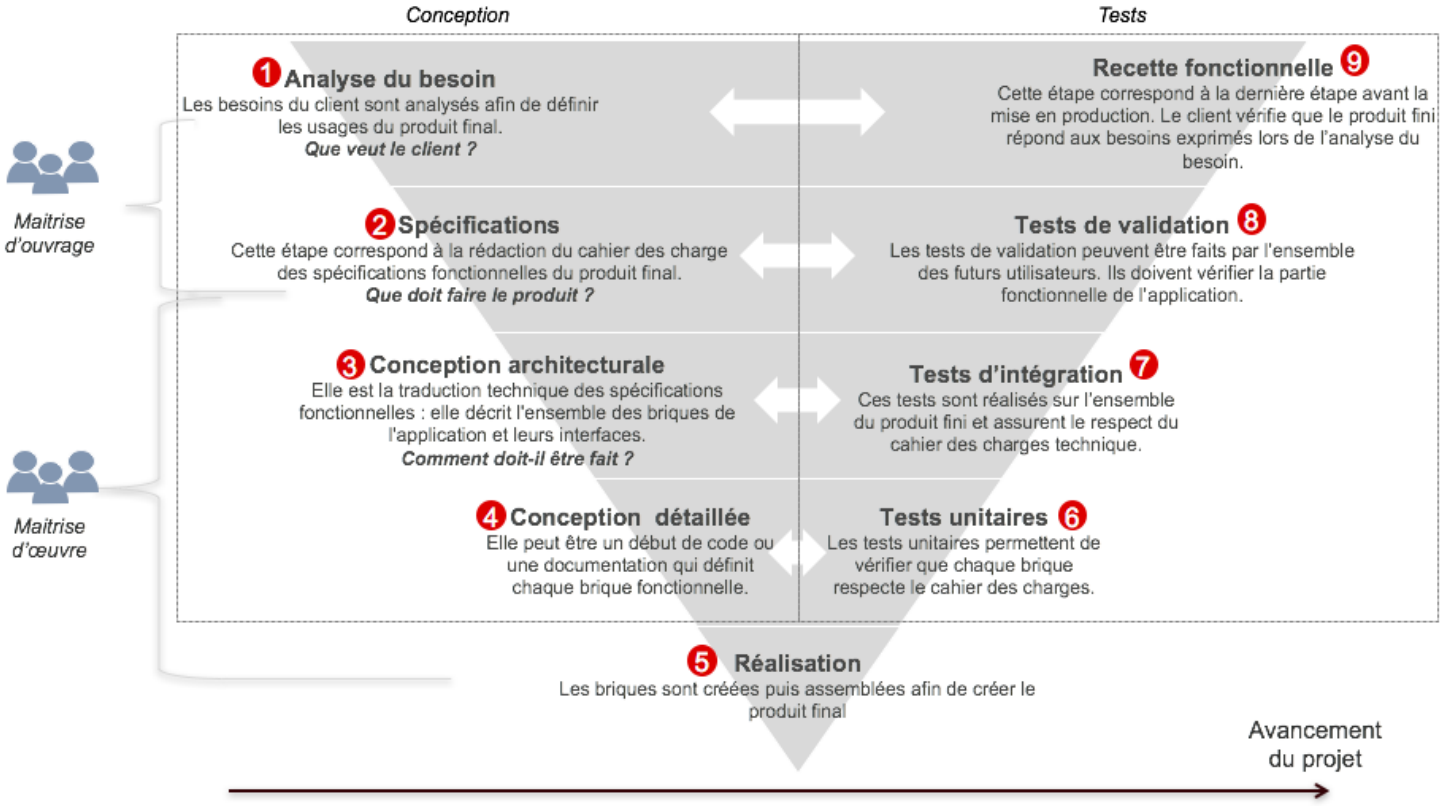
Comprendre la démarche Agile

Nicolas Haziza
nicolas.haziza@gmail.com

1.

Introduction à l’agilité

Le cycle en V : une méthode inadaptée



Le cycle en V : une méthode inadaptée

Avantage :

L'avantage principal de cette méthode est qu'elle présente des gains d'échelle (planning et/ou charge) en traitant d'un bloc l'ensemble du périmètre du projet.

Risques :

- Manque de communication : la communication est peu facilitée du fait du cloisonnement des rôles de chaque acteur : ils ont chacun leur tour un rôle à jouer mais ils ne le jouent jamais tous ensemble et communiquent principalement au travers de documents.
- Manque de souplesse : pour passer à l'étape suivante, il est important de terminer la précédente. Cela peut donc entraîner des dérives planning et donc de coûts.
- Péremption du produit : la durée de ce type de projet est potentiellement longue et le résultat final peine souvent à s'adapter aux évolutions du besoin.
- Manque de feedback : le produit fini ne répondra pas entièrement aux besoins du client et la méthode ne prévoit pas d'opportunité de vérification intermédiaire.

Qu’est ce que l’agilité

L’agilité est né au début du XXième siècle, lorsque les industries ont commencé à découper la production en micro-tâche.

Toyota met en place en premier la gestion de projet se rapprochant de l’agilité d’aujourd’hui en introduisant :

- une culture de l’amélioration en continu et d’autonomisation des salariés (**le kaizen**),
- l’étiquetage (**kanban**) pour la gestion des pièces à produire ou à livrer et éviter ainsi le gaspillage,
- l’adaptation face au changement,
- le respect des salariés,
- la force du travail d’équipe,
- l’importance d’une philosophie à long terme plutôt que les objectifs financiers à court terme.

En 2001 le manifeste agile est créé

Agile ou cycle en V : comment choisir ?

Voici les deux questions à se poser :

1- Suis-je en mesure de mobiliser une équipe qui pourra participer de manière active à mon projet (pour définir les besoins et tester les développements) ?

2- Suis-je complètement en maîtrise du besoin auquel le projet doit répondre et suis-je capable décrire son fonctionnement précis de façon certaine et complète ?

Si la réponse à la question 1 est positive, optez pour une méthode SCRUM.

Si la réponse à la question 2 est positive alors optez pour une méthode Cycle en V.

Qu’est ce que l’agilité : les 4 valeurs

- Les individus et leurs interactions plus que les processus et les outils :
  Communiquer à travers des processus et des outils nécessite un formatage des messages ainsi que le respect de leurs rythmes particuliers : structurons plutôt un contexte où les individus interagissent en direct de manière plus efficace.

- Du logiciel qui fonctionne plus qu’une documentation exhaustive :
  Documenter l’action de façon bureaucratique est généralement une perte de temps : réduisons cela au strict minimum nécessaire et choisissons plutôt l’avancée opérationnelle de notre produit comme repère.

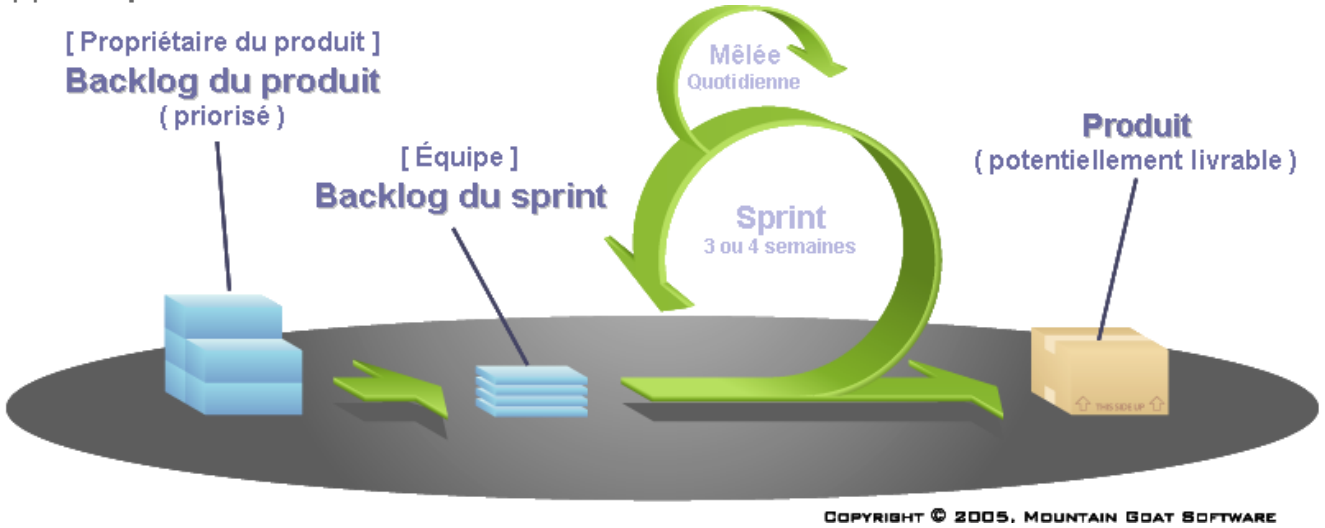
- La collaboration avec les clients plus que la négociation contractuelle :
  Impliquons l’utilisateur/client à chaque étape du développement – et pas seulement avant et après : c’est l’unique solution pour accompagner dynamiquement ses besoins réels.

- L’adaptation au changement plus que le suivi d’un plan :
  Procédons par itérations courtes plutôt que sur la base d’un plan : cela permettra de mieux nous adapter

L'agilité : SCRUM

SCRUM est un framework de la méthode Agile. Il décrit un cadre de travail permettant la mise en place des méthodes agiles.

Ce framework se base sur un découpage du projet en boîte de temps (Time-Box) appelé **Sprint.**



L’agilité : eXtrem Programming

eXtrem Programming est une méthode agile permettant l’amélioration de la qualité du code produit grâce au travail en binôme. L’idée est de pousser chaque concept à l’extrême :

- La revue de code est une bonne pratique, elle sera faite en permanence (par un binôme) ;
- Les tests sont utiles, ils seront faits systématiquement avant chaque mise en œuvre ;
- La conception est importante, le code sera retravaillé tout au long du projet (*refactoring*) ;
- La simplicité permet d'avancer plus vite, la solution la plus simple sera toujours celle qui sera retenue;
- La compréhension est importante, des métaphores seront définies et évolueront en concomitance;
- L'intégration des modifications est cruciale, celles-ci seront faites plusieurs fois par jour ;
- Les besoins évoluent vite, des cycles de développement très rapides faciliteront l'adaptation au changement.

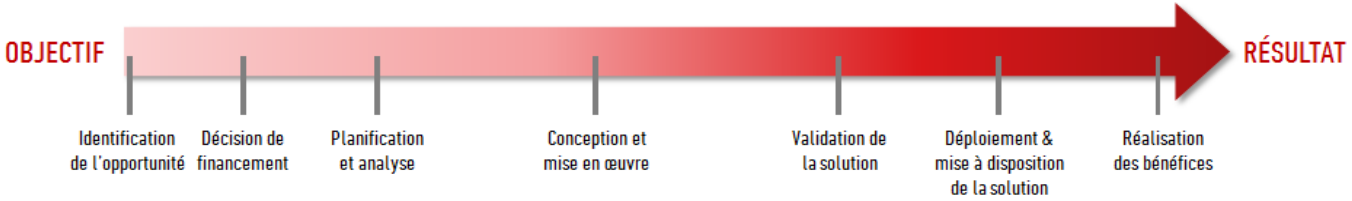
L'agilité : kanban

Le **Kanban** est un système visuel de gestion des processus qui indique quoi produire, quand le produire et en quelle quantité.

L'agilité et le devops

LES ENTREPRISES DOIVENT ÊTRE CAPABLE DE MODERNISER LEURS PROCESSUS DE GESTION DE PROJET

HIER

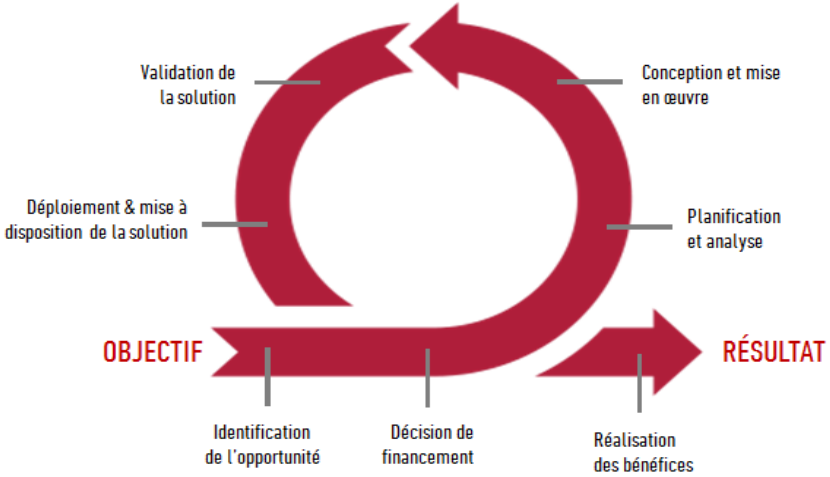


© SQLI GROUP | 2016

L'agilité et le devops

LES ENTREPRISES DOIVENT ÊTRE CAPABLE DE MODERNISER LEURS PROCESSUS DE GESTION DE PROJET

DEMAIN



© SQLI GROUP | 2016

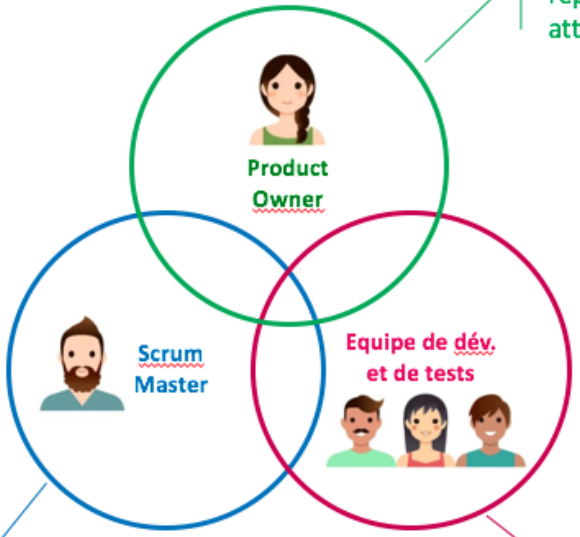
2.

L’agilité : les pratiques via SCRUM

SCRUM : les rôles & responsabilités

Le Product Owner (PO)	**Le stratège**. Il recueille les besoins, estime leur valeur. Il **porte la vision du produit**. Il est en charge de la construction et la mise à jour du backlog. Il est **garant du produit et de sa valeur**.
Le Scrum Master	**Le facilitateur**. Il **élimine les obstacles** dûent à l'environnement de travail. Il est **garant de l'application des principes et pratiques SCRUM**.
L'équipe de développement	**Les réalisateurs**. Ils **analysent et estiment les user stories**. Ils déterminent **la meilleure façon de réaliser leur travail**. Ils sont **garants de la qualité du livrable produit**.

SCRUM : les rôles & responsabilités



Définir le produit répondant aux attentes utilisateurs

Faciliter les échanges et l'amélioration

Réaliser le meilleur produit

SCRUM : l’expression du besoin

Les exigences du produit seront listés dans un **artefact** : le **Backlog**.

Celui-ci sera unique pour un produit, même si plusieurs équipe Scrum travaille dessus.

Il contiendra les **user-stories**, les **fonctionnalités**, leur **description** et **priorité**.

SCRUM : ATELIER expression besoin

Durant cet atelier chaque binôme / trinôme désignera un spécificateur et un/des artiste(s). Bien-sûr les rôles seront échangés entre chaque phase. Le projet du client sera représenté par un dessin et seul les spécifieurs y auront accès.

Le spécifieur a pour rôle de rédiger les spécifications du projet du client.

L’artiste lui réalise le projet du client.

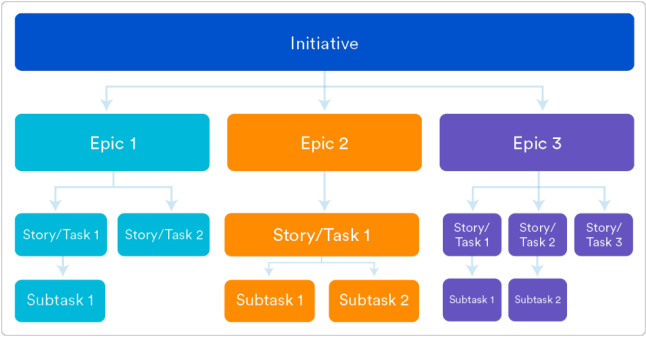
Contraintes :

- Les spécifieurs doivent écrire leurs instructions de manière littérale et linéaire. Aucun dessin ni échange vocal.
- Le seul moyen de communiquer entre un spécifieur et un artiste est l’écrit.

SCRUM : user-stories et tâches

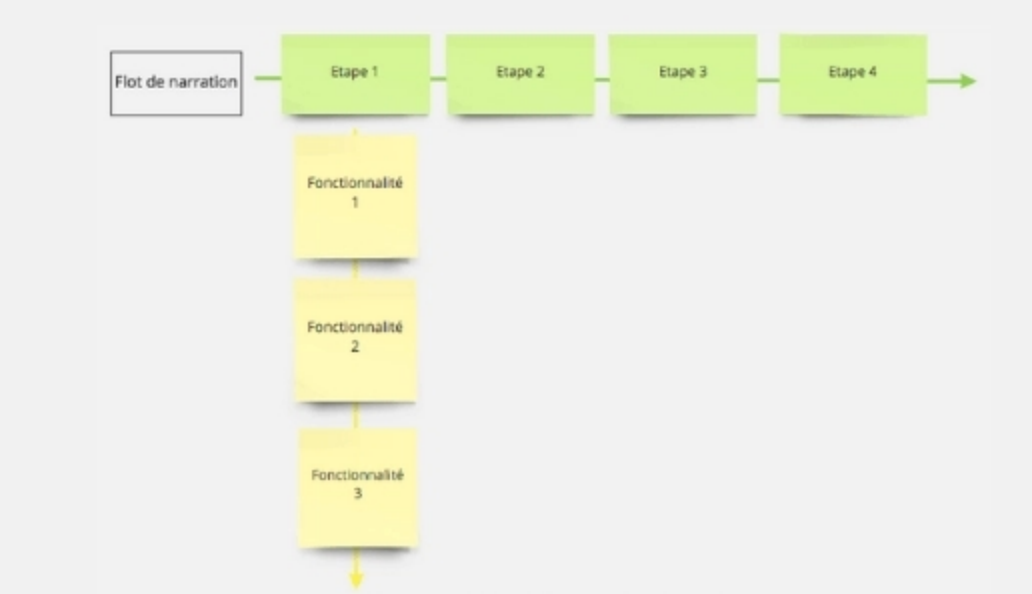
En SCRUM le rôle du PO est de garantir la valeur du produit. Pour se faire il va (en collaboration avec le client) réaliser des user-stories. Une story est un élément de backlog (aussi appelé élément du backlog de produit) qui apporte de la valeur aux utilisateurs, aux parties prenantes (Stakeholders) et parfois à l'équipe elle-même.

Celle-ci sera formulée de façon fonctionnelle puis sera **découpée** en **tâches** afin de faciliter son estimation et sa réalisation.



SCRUM : le story mapping

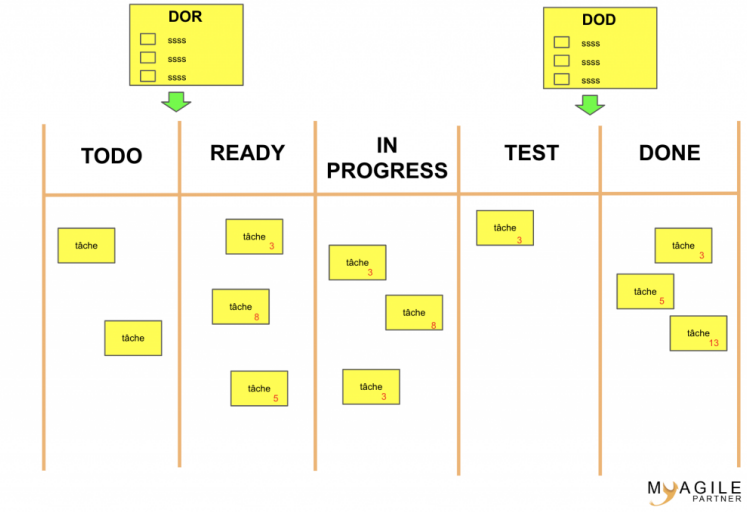
Le but d’un story mapping est d’accompagner le client dans son expression du besoin.



SCRUM : la DOR et la DoD

La **Definition Of Ready (DOR)** est la liste (définie par l'équipe) d'éléments attendus qu'une user story doit rassembler pour être candidate au développement.

La **Definition Of Done (DoD)** est la liste (définie par l'équipe) de critères à vérifier, afin de déterminer si les user stories ou tickets sont vraiment terminés.



SCRUM : Estimer les tâches

En SCRUM l’estimation des tâches se déroulent généralement autour d’un Scrum Poker. Le principe est simple, chaque participant possède une suite de carte chacune comportant un nombre de la suite de Fibonacci.

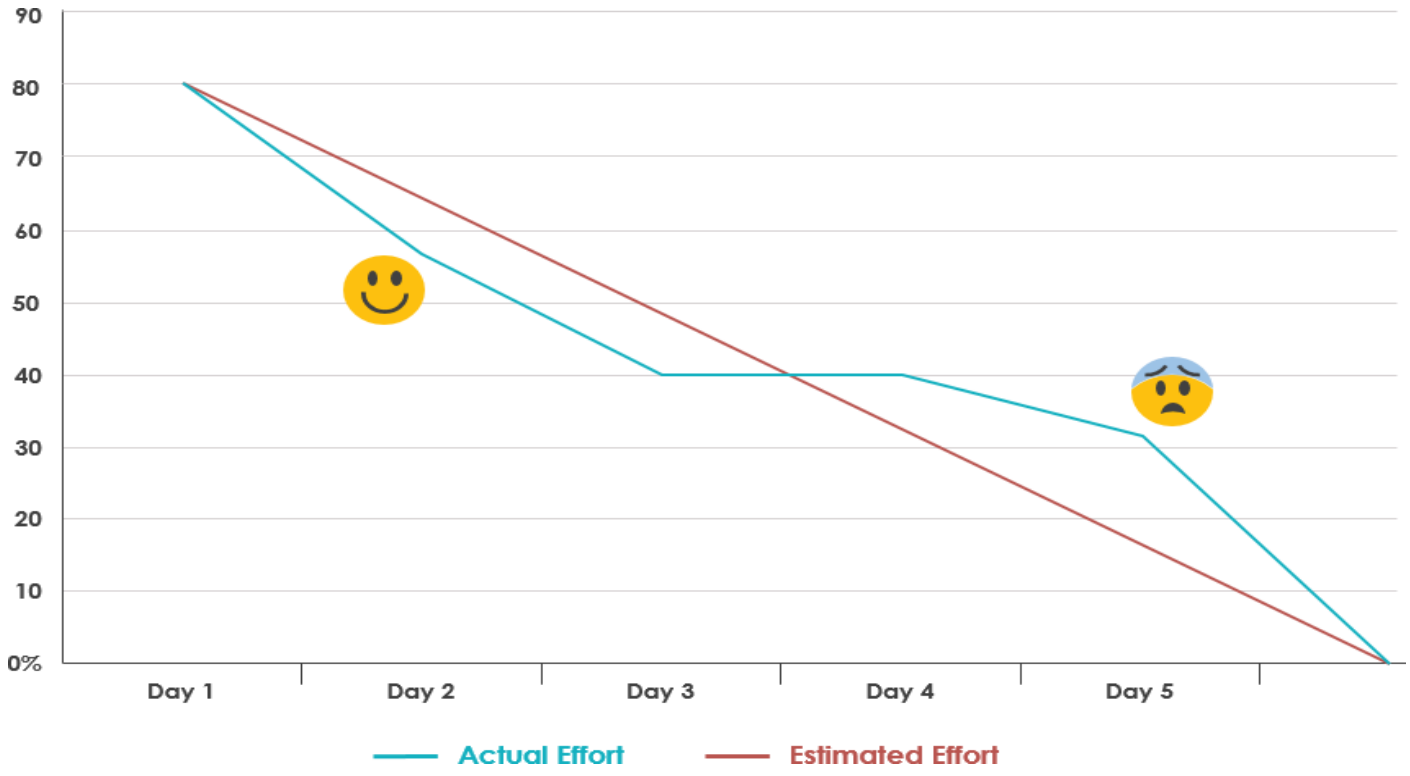
L’objectif est que **l’équipe développement** puissent estimer les tâches par un consensus. En effet les tâches ne sont pas à la responsabilités d’un développeur mais à l’ensemble de l’équipe. Il faut donc bien faire attention à ce que les tâches soient estimer à la hauteur de l’équipe et non pas de l’individu.

SCRUM : Atelier SCRUM Poker

Pour cet atelier nous allons estimer le temps de préparation d’une salade de fruits.

Je vous proposerai à chaque tour un fruit à découper (représentant une tâche) et vous pourrez alors voter avec la pondération qui vous semble la plus juste pour le temps nécessaire au découpage de ce fruit.

SCRUM : Burndown



SCRUM : les cérémonies

Le Sprint Planning

Objectif : **Organiser le sprint qui débute**

Déroulement :

Le Product Owner définit les objectifs du sprint

L'équipe estime les users stories à réaliser

Le Product Owner propose une liste de tâches à réaliser durant le sprint

SCRUM : les cérémonies

Le Dayli Scrum

Objectif : **Faire une état des lieux de l'avancement et relever les points bloquants**

Déroulement :

Chaque membre doit expliquer ce qu'il a fait depuis le dernier daily

Chaque membre doit expliquer ce qu'il va faire avant le prochain daily

Chaque membre doit lever les alertes s'il y en a

SCRUM : les cérémonies

Le Sprint Review

Objectif : **Point final d'un sprint, utilisé également pour présenter une itération aux parties prenantes**

Déroulement :

Le Product Owner présente le travail livré pendant le Sprint

Démonstration d'une nouvelle fonctionnalité

Le SCRUM Master présente les indicateurs d'avancée du projet

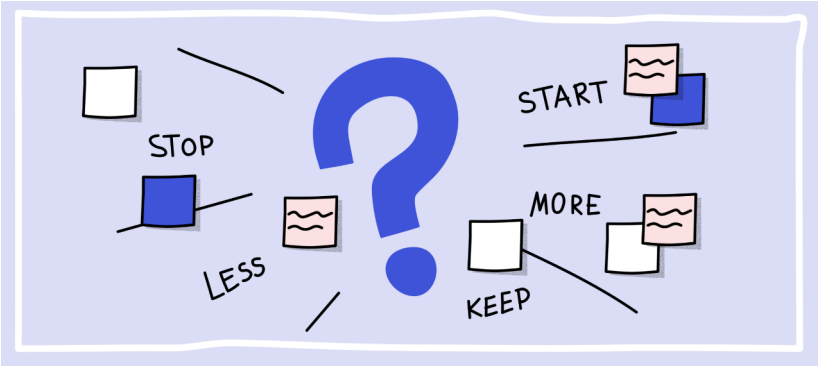
Echange avec les parties prenantes

SCRUM : les cérémonies

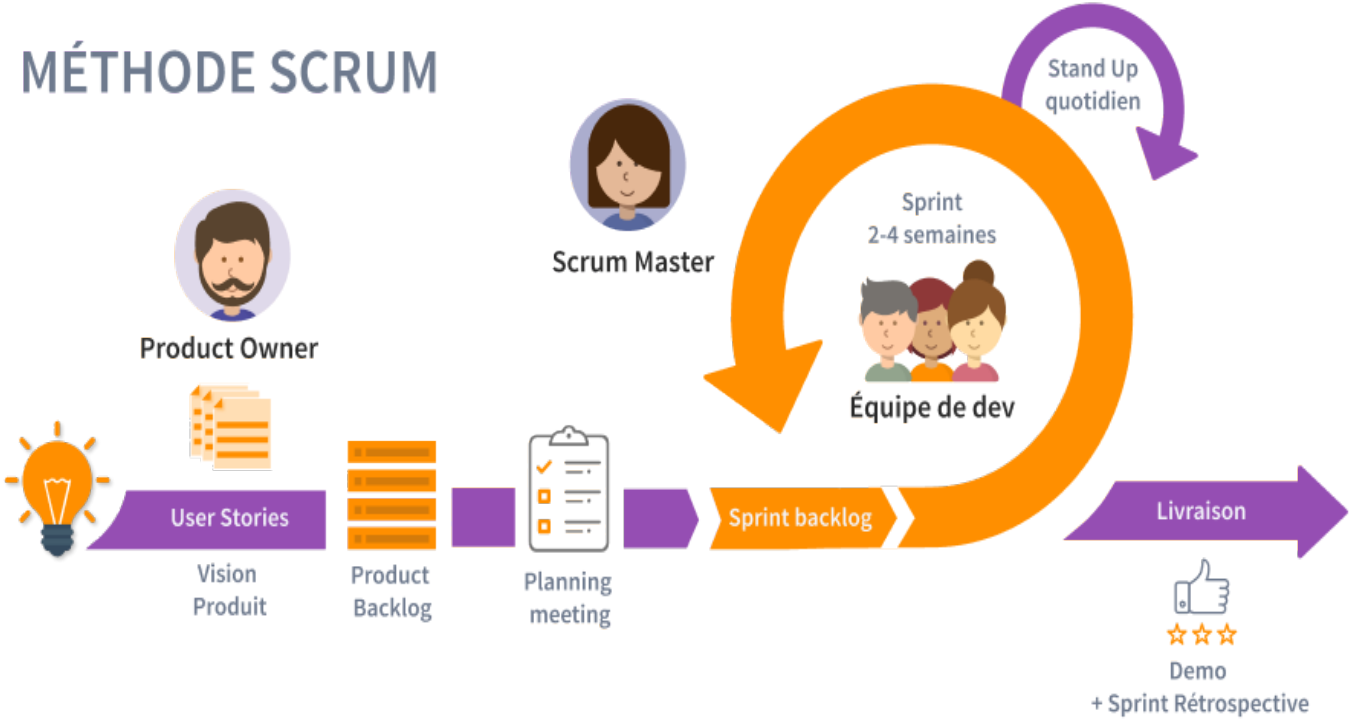
Le Sprint Retrospective

Objectif : **Travailler sur l'amélioration continue des process de l'équipe**

Déroulement : à définir en équipe !



SCRUM : au quotidien



SCRUM : QCM

SCRUM : outil JIRA

jira.teamsinspace.com

Teams in Space
Software project

- Backlog
- Board
- Reports
- Releases
- Components
- Issues
- Repository
- Add item
- Settings

Board

Release

Quick Filters

TO DO 5	IN PROGRESS 5	CODE REVIEW 2	DONE 8
Engage Jupiter Express for outer solar system travel SPACE TRAVEL PARTNERS 5 TIS-25	Requesting available flights is now taking > 5 seconds SEESPACEEZ PLUS 3 ~~TIS-8~~	Register with the Mars Ministry of Revenue LOCAL MARS OFFICE 3 TIS-11	Homepage footer uses an inline style - should use a class LARGE TEAM SUPPORT TIS-68
Create 90 day plans for all departments in the Mars Office LOCAL MARS OFFICE 9 TIS-12	Engage Saturn Shuttle Lines for group tours SPACE TRAVEL PARTNERS 4 TIS-15	Draft network plan for Mars Office LOCAL MARS OFFICE 3 TIS-15	Engage JetShuttle SpaceWays for travel SPACE TRAVEL PARTNERS 5 TIS-23
Engage Saturn's Rings Resort as a preferred provider SPACE TRAVEL PARTNERS 3 TIS-17	Establish a catering vendor to provide meal service LOCAL MARS OFFICE 4 TIS-15		Engage Saturn Shuttle Lines for group tours SPACE TRAVEL PARTNERS TIS-15
Enable Speedy SpaceCraft as the preferred	Engage Saturn Shuttle Lines for group tours		Establish a catering vendor to provide meal service

SCRUM : atelier Jira

Pour cet atelier l’objectif est de créer un projet représentant votre week-end. Celui-ci doit prévoir deux sprints : un samedi et un dimanche.

Vous devez créer au moins 2 epics, 2 user-stories et 4 tâches. Bien-sûr vous pouvez allez plus loin si vous le souhaitez.

Les tâches devront être estimées.

MODULE SQL

Langage SQL ou PL/SQL

Nicolas Haziza
nicolas.haziza@gmail.com

1.

Base de données : les concepts

Persistance de la donnée

En programmation, la gestion de la persistance des données et parfois des états d'un programme réfère au mécanisme responsable de la sauvegarde et de la restauration des données. Ces mécanismes font en sorte qu'un programme puisse se terminer sans que ses données et son état d'exécution ne soient perdus.

Ces informations de reprise peuvent être enregistrées sur disque, éventuellement sur un serveur distant (un serveur de bases de données relationnelles, par exemple).

Base de données

- **Ensemble structuré d'informations** (*d'une entreprise ou organisation*), **mémorisé** sur une machine (*serveur*).

- **Données stockées** et **organisées** sous forme de fichiers ou ensemble de fichiers.

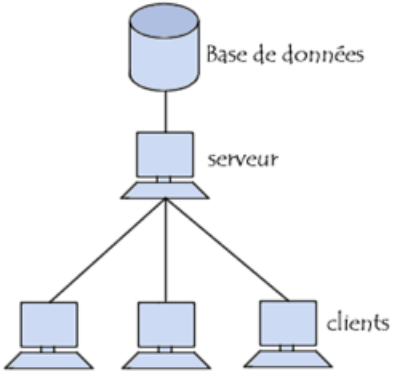
- Une BD sert à **créer**, **enregistrer**, **récupérer** et **manipuler** des **données communes**.

SGBD

- **Système de Gestion de Base de Données** (ou DBMS: Data Base Management System)

- **Ensemble cohérant** de **services** (*logiciels*) permettant aux utilisateurs **d'accéder**, **mettre à jour** ou **administrer** une DB

- Fonctionne sur le modèle **client/serveur** (requêtes/traitements)



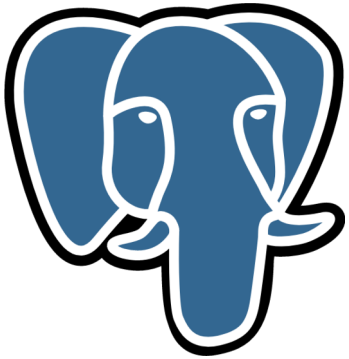
SGBD

Pourquoi utiliser un SGBD ? Quels objectifs ?

- Indépendance physique
- Indépendance logique
- Accès / partage des données
- Administration centralisée
- Non redondance des données
- Cohérence des données
- Sécurité des données
- Résistance aux pannes

SGBD

- Installation de postgreSQL



2.

Base de données : modélisation

Modélisation

- **Pourquoi modéliser ?**
  - Avoir une représentation graphique de la structure
  - Connaître les propriétés attendues d’une données
  - Connaître les relations entre les données

- **Comment modéliser ?**
  - Effectuer un design conceptuel
  - Insérer des cardinalités
  - Effectuer un modèle logique

Modélisation : savoir extraire l'information

Lorsque vous allez recevoir un cahier des charges, il faudra tout d'abord réussir à en extraire les informations qui nécessitent un stockage persistant. Ces information seront regroupées dans un documents appelé le "**dictionnaire de données**".

Celui-ci se présente généralement sous forme d'un tableau comme suit :

Nom donnée	Type donnée	Référence	Commentaire
nom_client	TEXT	Client	Contient le nom des clients

L'objectif est d'obtenir le contenu de la structure de nos **tables** pour notre futur base de données.

Modélisation : Les types de données

Dans une base de données il existe plusieurs **types** de données utilisables en fonction de la nature de l'information que l'on souhaite stocker :

Nombre entier :

Type	Nombre d'octets	Minimum	Maximum
TINYINT	1	-128	127
SMALLINT	2	-32768	32767
MEDIUMINT	3	-8388608	8388607
INT	4	-2147483648	2147483647
BIGINT	8	-9223372036854775808	9223372036854775807

Chaîne de caractères :

Type	Longueur maximale	Mémoire occupée
TINYTEXT	2^8 octets	Longueur de la chaîne + 1 octet
TEXT	2^16 octets	Longueur de la chaîne + 2 octets
MEDIUMTEXT	2^24 octets	Longueur de la chaîne + 3 octets
LONGTEXT	2^32 octets	Longueur de la chaîne + 4 octets

Tous les types sont trouvables ici : https://www.postgresql.org/docs/9.5/datatype.html

Modélisation : Atelier

Atelier extraction de données :

Dans le cahier des charges qui vous ai fourni. Extraire un maximum de données qui pour vous doivent être stockées dans une base. Le format de sorti attendu est un dictionnaire de données.

Modélisation : la clef primaire

Lorsque vous allez stocker une information dans votre base de données, il est intéressant de mettre en place une information **unique** et **simple** qui permettra de retrouver cette donnée dans la base aisément. Cette information est appelé la **“clef primaire”.**

En général celle-ci est au format numérique et se trouve à l’intérieur d’une colonne nommée “**id**”.

id	prenom	nom
1	Chandler	Bing
2	Phoebe	Buffay
3	Monica	Geller
4	Ross	Geller
5	Chandler	Bing

Modélisation : les relations

Une relation a été créé entre deux tables, ce qui veut dire que leurs données sont désormais liées.

Il existe 3 types de relations :

- un à un (one-to-one)
- un à plusieurs (one-to-many) ou plusieurs à un (many-to-one)
- plusieurs à plusieurs (many-to-many)

Chacun de ces types engendre une conséquence différente sur le modèle de données.

Modélisation : les relations

Afin d'éviter les doublons de données il est possible de mettre en place des **relations** entre nos différentes **tables/entités**.

Par exemple :

id	nom	modele	annee_sortie
1	Jean Manchzeck	Kawasaki 750	2015
2	Edouard Bracame	Kawasaki 750	2015

Pourrait devenir :

id	nom
1	Jean Manchzeck
2	Edouard Bracame

0,1
0,1

id	modele	annee_sortie
1	Kawasaki 750	2015

Modélisation : les relations

One-to-One : Une relation one-to-one implique la création d'une **clé étrangère** dans l'une des deux tables. Cette clé représente la référence de la seconde table. Si on reprend l'exemple précédent :

id	nom
1	Jean Manchzeck
2	Edouard Bracame

0,1

0,1

id	modele	annee_sortie	conducteur
1	Kawasaki 750	2015	1
2	Kawasaki 750	2015	2

Modélisation : les relations

One-to-Many : Une relation one-to-many implique également la création d'une **clé étrangère** dans l'une des deux tables. Par contre dans ce cas nous n'avons pas le choix de la table qui portera la référence. Si on reprend l'exemple précédent :

id	nom
1	Jean Manchzeck
2	Edouard Bracame

0,1

n

id	modele	annee_sortie	conducteur
1	Kawasaki 750	2015	1
2	Kawasaki 750	2015	2

Modélisation : les relations

Many-to-Many : Une relation many-to-many engendrera la création d'une table de correspondance. Si on reprend l'exemple précédent :

id	nom
1	Jean Manchzeck
2	Edouard Bracame

conducteur	moto
1	1
2	2

id	modele	annee_sortie
1	Kawasaki 750	2015
2	Kawasaki 750	2015

Modélisation : les règles à respecter

- **Normalisation des tables**
  - Ne contient pas d’espace, d’accents ni de caractères spéciaux
  - Tout doit être écrit en lowercase (minuscule)
  - Les espaces sont remplacés par des underscores : “_”
- **Les relations**
  - En UML l’astuce est de lire les relations avec en les liants avec le verbe “possède” afin de ne pas se tromper dans l’ordre de celles-ci.
- **Un identifiant**
  - Un champs d’identification unique est obligatoire, la clef primaire (souvent nommé “id”).

Modélisation du CRM

- Table client
- Table commandes
- Table utilisateurs

3.

Base de données : Le SQL

SQL: Les requêtes structurelle

Il existe plusieurs instructions possibles sur les **objets** de votre base :

- CREATE **t n** pour créer un objet de type **t** et de nom **n**
- ALTER **t n** pour modifier le schéma d'un objet de type **t** et de nom **n**
- DROP **t n** pour supprimer un objet de type **t** et de nom **n**

Exemple :

- **Création d'une base**
  - CREATE DATABASE <nom_colonne>;
- **Création d'une table**
  - CREATE TABLE <nom_table>(<nom_colonne> <type_colonne> <contraintes>, ...);
- **Modification d'une table**
  - ALTER TABLE <nom_table> ALTER COLUMN <nom_colonne> TYPE <type_colonne> <contraintes>;

SQL: Les requêtes (CRUD)

Il existe plusieurs instructions possibles sur les **données** de votre base :

- INSERT pour ajouter des lignes à une table
- UPDATE pour modifier des lignes d'une table
- DELETE pour supprimer des lignes d'une table
- SELECT pour extraire des données à partir de tables existantes

Exemple :

- **Lire des données**
  - SELECT <nom_colonne> FROM <nom_table>
- **Ajouter des données**
  - INSERT INTO <nom_table> VALUES (‘valeur1’, ‘valeur2’, ...)
- **Modifier des données**
  - UPDATE <nom_table> SET <nom_colonne> = ‘valeur’
- **Supprimer des données**
  - DELETE FROM <nom_table> WHERE <condition>

SQL: La clause WHERE

La clause **WHERE** dans une requête SQL permet d'extraire les lignes d'une base de données qui respectent une condition. Cela permet d'obtenir uniquement les informations désirées.

Exemple :

- SELECT <nom_colonne> FROM <nom_table> WHERE id=1;
- UPDATE <nom_table> SET <nom_colonne> = 'valeur' WHERE name="Toto";

Il est possible d'ajouter plusieurs conditions sur une requête via les opérateurs **AND** et **OR**.

Exemple :

- SELECT <nom_colonne> FROM <nom_table> WHERE id=1 AND name="Toto";
- SELECT <nom_colonne> FROM <nom_table> WHERE id=1 OR name="Toto";

SQL: Les jointures

Les jointures en SQL permettent d’associer plusieurs tables dans une même requête. Cela permet d’exploiter la puissance des bases de données relationnelles pour obtenir des résultats qui combinent les données de plusieurs tables de manière efficace.

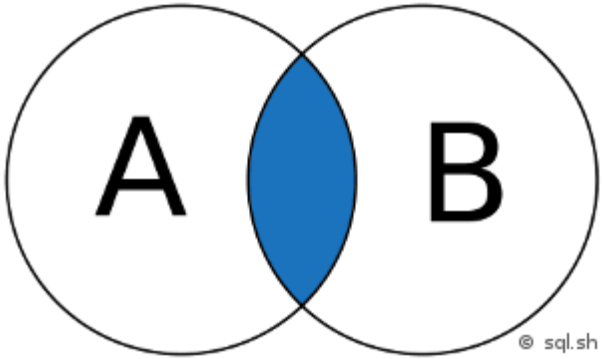
Il existe plusieurs types de jointures :

- INNER JOIN : jointure interne pour retourner les enregistrements quand la condition est vrai dans les 2 tables. C’est l’une des jointures les plus communes.
- CROSS JOIN : jointure croisée permettant de faire le produit cartésien de 2 tables. En d’autres mots, permet de joindre chaque lignes d’une table avec chaque lignes d’une seconde table. Attention, le nombre de résultats est en général très élevé.
- LEFT JOIN (ou LEFT OUTER JOIN) : jointure externe pour retourner tous les enregistrements de la table de gauche (LEFT = gauche) même si la condition n’est pas vérifié dans l’autre table.
- RIGHT JOIN (ou RIGHT OUTER JOIN) : jointure externe pour retourner tous les enregistrements de la table de droite (RIGHT = droite) même si la condition

SQL: Les jointures INNER JOIN

Voici la syntaxe de l’INNER JOIN :

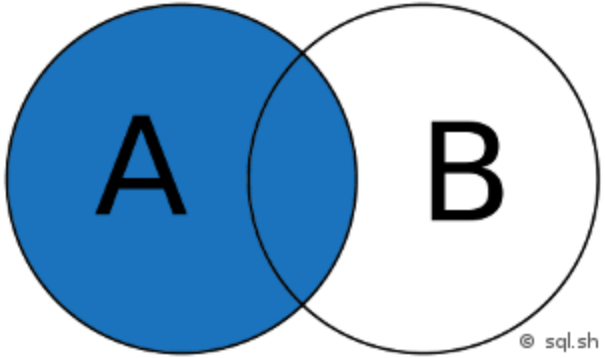
SELECT * FROM table1 INNER JOIN table2 ON table1.id = table2.fk_id;



SQL: Les jointures LEFT JOIN

Voici la syntaxe du LEFT JOIN :

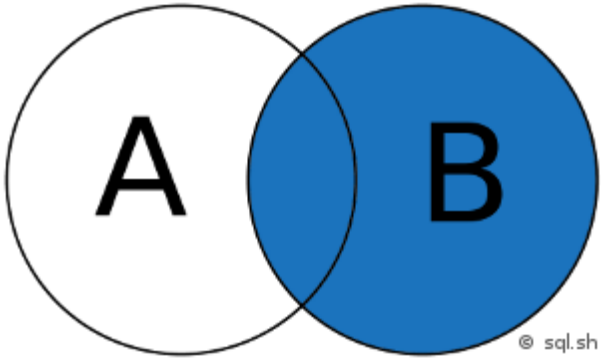
```
SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.fk_id;
```



SQL: Les jointures RIGHT JOIN

Voici la syntaxe du RIGHT JOIN :

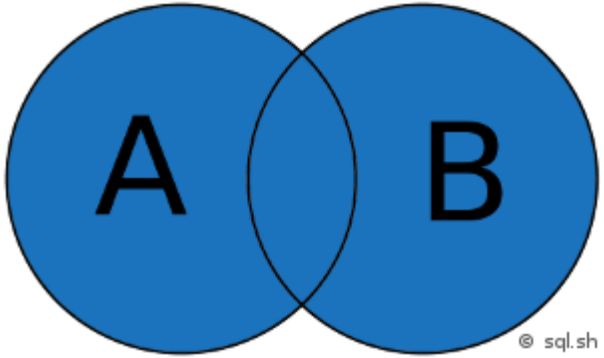
```
SELECT * FROM table1 RIGHT JOIN table2 ON table1.id = table2.fk_id;
```



SQL: Les jointures FULL JOIN

Voici la syntaxe du FULL JOIN :

```
SELECT * FROM table1 FULL JOIN table2 ON table1.id = table2.fk_id;
```



SQL: Les jointures CROSS JOIN

Voici la syntaxe du CROSS JOIN :

```
SELECT * FROM table1 CROSS JOIN table2;
```

Attention : effectuer le produit cartésien d’une table A qui contient 30 résultats avec une table B de 40 résultats va produire 1200 résultats (30 x 40 = 1200). En général la commande CROSS JOIN est combinée avec la commande WHERE pour filtrer les résultats qui respectent certaines conditions.

Les outils Devops

Git, Jenkins, Docker, Ansible

Nicolas Haziza
nicolas.haziza@gmail.com

1.

DevOps : Git

Git : C’est quoi, à quoi ça sert ?

- Logiciel de gestion des versions des sources
- Fonctionne via un système de client (local) et serveur (distant)

- Permet le partage et la sauvegarde de chaque version du code
- Permet à un grand nombre de collaborateurs de travailler sur un même projet, en offrant une bonne gestion des modifications.
- Permet de synchroniser un projet entre plusieurs collaborateurs, en ayant l’assurance que tous les fichiers soient à jour.
- Permet d’avoir un historique précis de tous les changements et modifications d’un projet. Cela permet de clarifier les questions récurrentes: “Où est la dernière version du fichier X?” et “Qu’est-ce qui a été changé entre les révisions 41 et 42”?.

Git : Terminologie

- **Repository** (Repo) : c’est le dépôt, le dossier de travail contenant les fichiers d’un projet
- **Commit** : action d’enregistrer vos modifications dans l’historique du projet. Il est accompagné d’un commit message.
- **Push** : permet d’envoyer les commits locaux vers le serveur (remote)
- **Pull** : récupérer les commits depuis le serveur en local
- **Checkout** : permet de voyager dans le temps, en retournant vers un commit précis
- **Stash** : Permet de mettre de côté vos modifications, sans commiter
- **Merge** : action de fusionner deux versions d’un projet
- **Rebase** : action de rembobiner pour modifier des commits précédents
- **Branch** : version alternative du projet

Git : Interface

Git : Fonctionnement

Pour démarrer un projet git il faut initialiser un **repository (dépôt)**. Pour cela vous avez deux approches :

- Importer un répertoire déjà existant dans git
- Cloner un repository git existant

Git pense les données à la manière d'un flux d'instantanés ou "snapshots". C'est à dire :

A chaque fois qu'on va valider ou enregistrer l'état d'un projet dans Git, il va prendre un instantané du contenu de l'espace de travail à ce moment et va enregistrer une référence à cet instantané pour qu'on puisse y accéder par la suite.

Chaque instantané est stocké dans une base de donnée locale, c'est-à-dire une base de donnée située sur notre propre machine.

Git : Les états des fichiers

Afin de suivre l’état d’un projet git marque d’un état chaque fichier du projet.

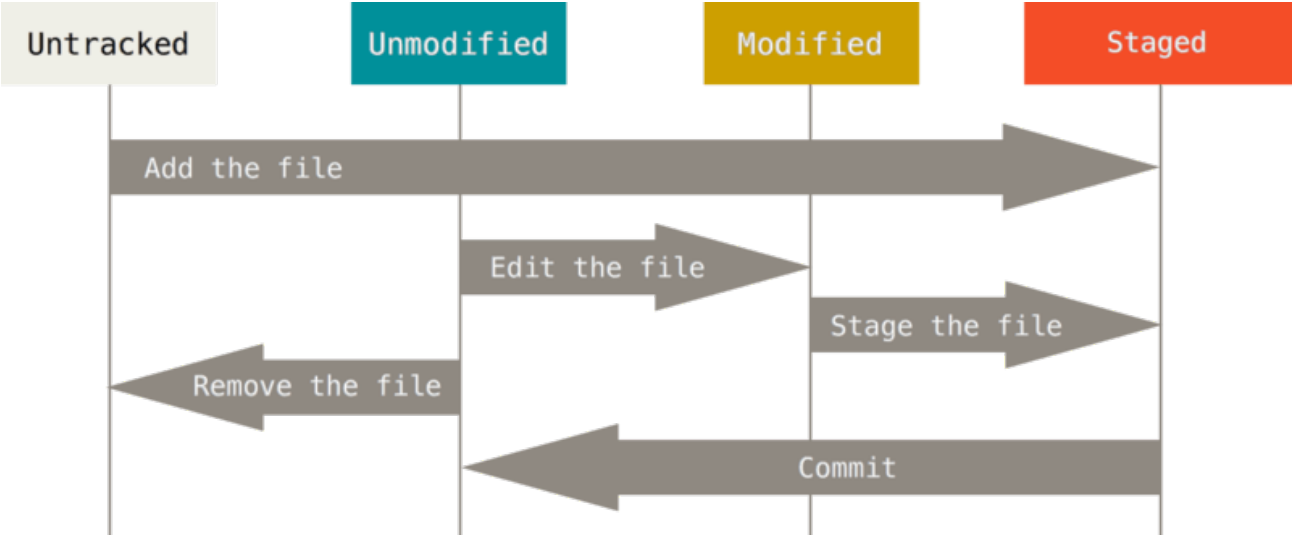
Lorsqu’on démarre un dépôt Git en important un répertoire déjà existant depuis notre machine, les fichiers sont au départ tous non suivis. On va donc déjà devoir demander à Git de les indexer et de les valider (les enregistrer en base). Lorsqu’on clone un dépôt Git déjà existant, c’est différent puisqu’on copie tout l’historique du projet et donc les fichiers sont tous déjà suivis par défaut.

Ensuite, chaque fichier suivi peut avoir l’un de ces trois états :

- **Modifié (“modified”) ;**
- **Indexé (“staged”) ;**
- **Validé (“committed”).**

Git : Les zones de travail

Les états de fichiers sont liés à des zones de travail dans Git. En fonction de son état, un fichier va pouvoir apparaître dans telle ou telle zone de travail. Tout projet Git est composé de trois sections : le répertoire de travail (working tree), la zone d'index (staging area) et le répertoire Git (repository).



Git : Premier pas

- Création d’un dépôt
- Récupération du dépôt en local

Exercice 1 : Créer un répertoire sandwich. Créer un fichier à l’intérieur ‘burger.txt’ qui contient la liste des ingrédients d’un burger (un par ligne), par exemple :
Steak
Salade
Tomate
Cornichon
Fromage

Git : Premier pas

Exercice 2 :

- Vérifier l’état de votre dépôt local : git status
- Préparer burget.txt pour le commit : git add
- Commitez les modifications : git commit -m “message de commit”
- Vérifier l’état du dépôt local
- Vérifier la liste des changements prêt à l’envoi : git log

Git : Premier pas

Exercice 3 :

- Créez quelques autres sandwichs (hot_dog.txt, jambon_beurre.txt, etc.), modifiez également le contenu de burger.txt. **Attention** : chaque modification doit être placée dans un commit dédié. Effectuez au moins 5 modifications en reprenant les étapes de l’exercice 2.
- Testez git log --graph --pretty=short

Git : Premier pas

Exercice 4 :

- Réalisez une modification dans une de vos recettes. **Attention** ne commitez pas
- Vérifier les modification via un git status puis git diff
- Réalisez un git reset <nom_du_fichier>
- Vérifier les modification via un git status puis git diff
- Réalisez un git checkout <nom_du_fichier>
- Réalisez un git status

Git : Premier pas

Exercice 5 :

- Réalisez un push : git push
- Allez constater la mise à jour sur le repository

Git : Les conflicts

Les systèmes de contrôle de version sont, par essence, destinés à la gestion des contributions entre plusieurs auteurs distribués (généralement, des développeurs). Parfois, plusieurs développeurs peuvent essayer de modifier le même contenu. Si le développeur A essaie de modifier le même code que le développeur B, un conflit peut survenir.

Il faudra alors choisir **manuellement** le travail du développeur A ou celui du développeur B. Une autre solution est possible en réalisation la fusion de code manuellement également.

.

Git : Second pas

Refaire les étapes des précédents exercices mais cette fois en **binome** ou **trinome.**

Votre projet final ne doit contenir aucun conflit.

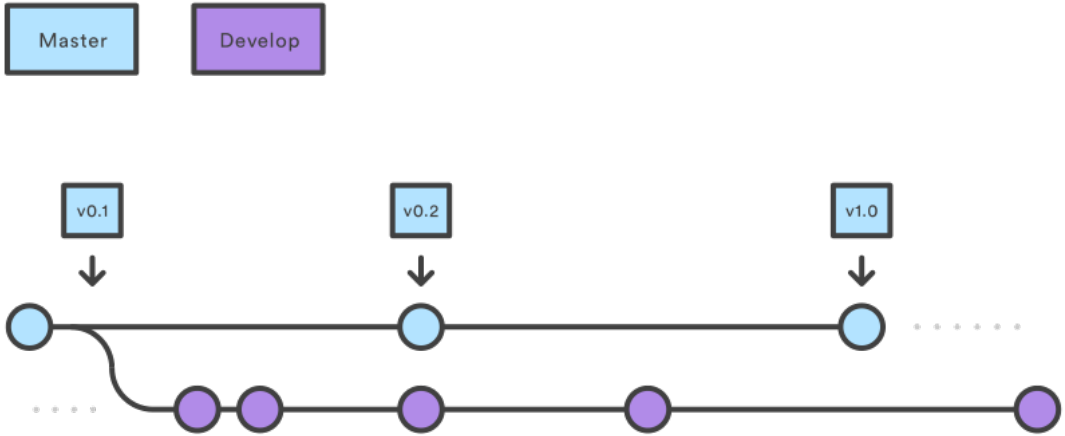
.

Git : Le Gitflow

- Le Gitflow est un workflow de travail. Il décrit les processus d’utilisation de git dans une équipe (créé par Vincent Driessen).
- Le Gitflow représente un framework particulièrement efficace et adapté aux projets avec un cycle de livraison planifié et régulier.
- Ce worklow n’ajoute aucun concept ni aucune commande en dehors de ce qui est exigé pour le workflow d’organisation des branches

Gitflow : Develop & Master

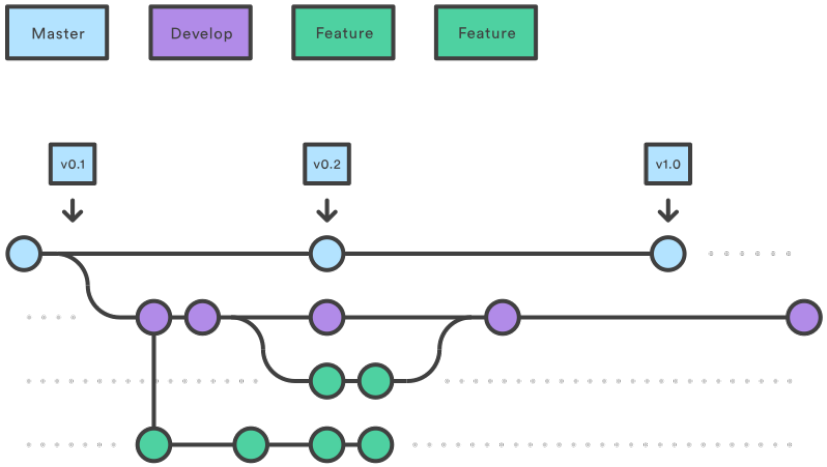
- Le Gitflow utilise deux branches principales au lieu d’une seule.
- La branche master stocke l’historique officiel des versions
- La branche develop sert de branche d’intégration pour les fonctionnalités



- Reprendre le projet des exercices précédent et créer une branche develop depuis master et pusher

Gitflow : Branche de fonctionnalité

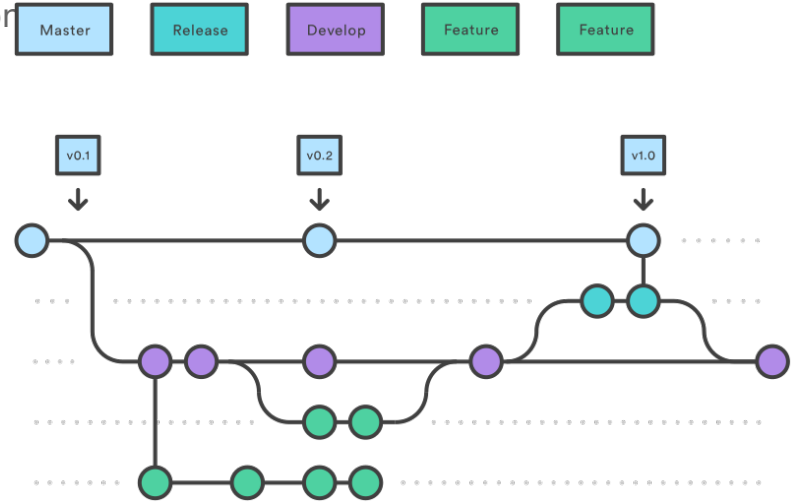
- Le Gitflow utilise une branche par fonctionnalité à développer
- Ces branches sont créees à partir de la branche develop
- Ces branches seront ensuite pushées en vue d’une intégration sur develop



- Créer une branche feature/xxx sur le projet, ajouter un sandwich et pushez sur le repository

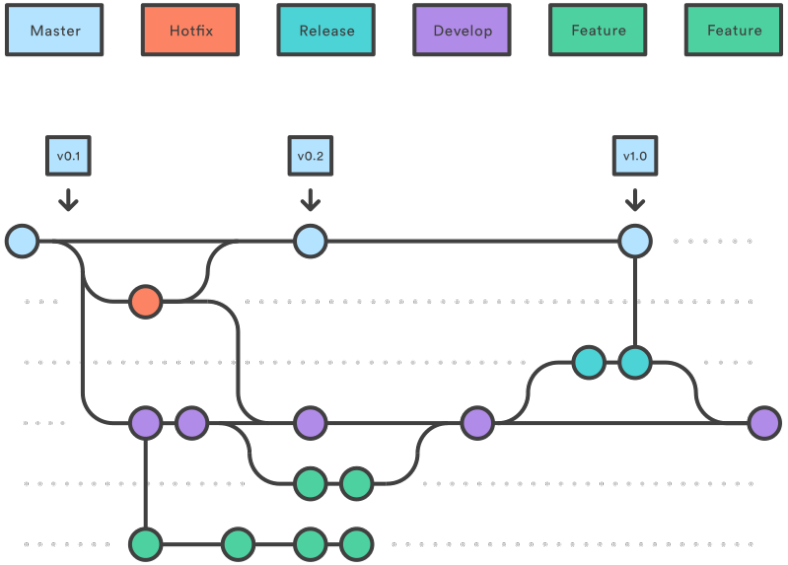
Gitflow : Branche de livraison

- Lorsque vous souhaitez préparer une livraison d'une version, vous allez créer une branche de livraison, souvent nommée release, depuis develop
- La création de cette branche débute un cycle de livraison :
  - Aucune fonctionnalité supplémentaire de pourra être ajoutée à cette branche
  - Dédiée à la corrections de bugs, documentation
- Une fois la branche prête elle sera mergée dans master et taguée d'un numéro de version



Gitflow : Branche de hotfix

- Les branches de maintenances (hotfix) sont utilisées pour appliqué
- Très similaire aux branches de release mais se base sur master (c’est la seule branche qui doit être créée depuis master après develop)
- Une fois la branche prête elle sera mergée dans master et taguée d’un numéro de version (correctif)



Gitflow : Un dernier entraînement

Gitflow : Le résumé

- Une branche de développement (develop) est créée à partir de master.
- Une branche de version (release) est créée à partir de develop.
- Les branches de fonctionnalité (feature) sont créées à partir de develop.
- Lorsqu'une branche de fonctionnalité (feature) est terminée, elle est mergée dans la branche de développement (develop).
- Lorsque la branche de version (release) est terminée, elle est mergée dans les branches de développement (develop) et principale (master).
- Si un problème est identifié dans la branche principale (master), une branche de maintenance (hotfix) est créée à partir de master.
- Une fois la branche de maintenance (hotfix) terminée, elle est mergée dans les branches de développement (develop) et principale (master).

2.

DevOps : Ansible

Ansible : présentation

Ansible est un moteur d'automatisation informatique open source, qui peut éliminer les corvées de votre vie professionnelle, et améliorer également considérablement l'évolutivité, la cohérence et la fiabilité de votre environnement informatique.

Vous pouvez utiliser Ansible pour automatiser trois types de tâches :

- Provisioning : configurer les différents serveurs dont vous avez besoin dans votre infrastructure
- Configuration : modifier la configuration d'une application, d'un système d'exploitation ou d'un périphérique, démarrer et arrêter les services, installer ou mettre à jour des applications, mettre en œuvre une politique de sécurité, ou effectuer une grande variété d'autres tâches de configuration.
- Déploiement d'applications : adopter une démarche DevOps en automatisant le déploiement d'applications développées en interne sur vos environnements de production.

En gros, c’est l’outil de scripting idéal pour un devOps.

3.

DevOps : Docker

Docker : présentation

Docker permet de créer des environnements (appelées containers) de manière à isoler des applications.

Il repose sur le kernel **Linux** et sur une fonctionnalité : les **containers**.

Le principe est d'exécuter du code dans un environnement isolé.

Docker a également d'un autre composant : **cgroups.** Ce composant a pour objectif de gérer les ressources (utilisation RAM, CPU, etc.).

Docker vs Machine virtuel (VM)

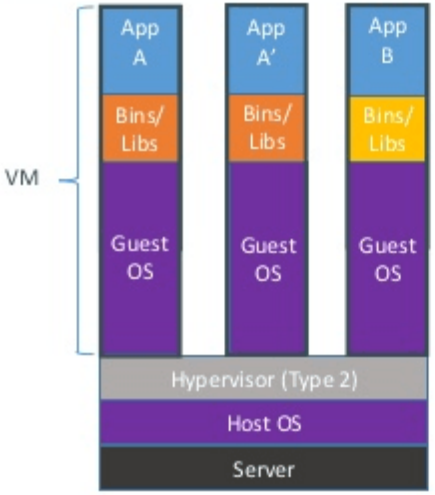
Une machine virtuelle isole tout un système (son OS), et dispose de ses propres ressources.

Dans le cas de **Docker**, le kernel va partager les ressources du système hôte et interagir avec les containers. Techniquement, **Docker n'est pas une VM**, mais en terme d'utilisation, **Docker** peut-être apparenté à une **VM**.

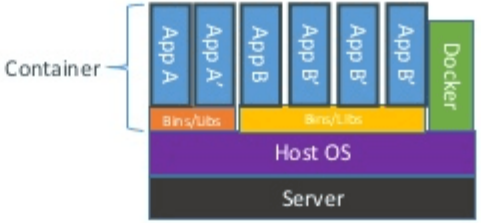
Lancer un environnement, et isoler les composants de ce container avec les composants de mon hôte, voilà ce que **Docker** sait faire ! Il le fait d'ailleurs très bien, et reste une alternative bien plus performante que les **VMs** (à utilisation équivalente).

Docker vs Machine virtuel (VM)

Containers vs. VMs



Containers are isolated, but share OS and, where appropriate, bins/libraries



Docker : cas d’utilisation

Une entreprise lyonnaise que j’appellerai à tout hasard **Wanadev**, souhaite développer un projet Java comme ils savent le faire. Leur équipe est composée de deux personnes.

- Manu est un vieux de la vieille. Lui, Debian 7, c'est parfait. De plus, il est ISO avec les serveurs de production.
- Baptiste lui est un vrai hippie. Il dispose de la dernière distribution exotique de debian. Lui, **JAVA**, c'est la toute dernière version ou rien.

Cependant il va falloir réaliser des tests unitaires, ce qui va poser problèmes aux deux développeurs...

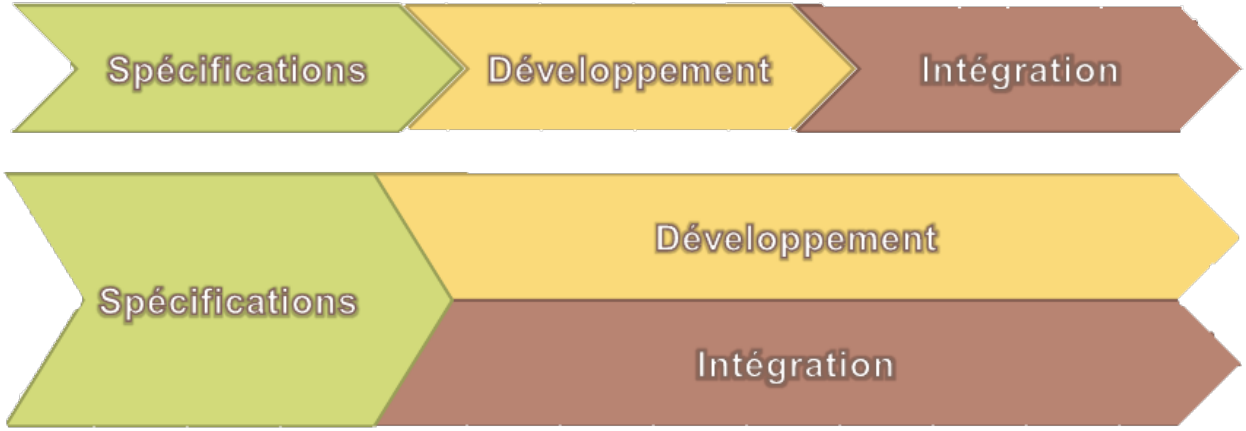
En effet, ça pose problème que deux développeurs ne travaillent pas sur les mêmes environnements. Grâce à Docker, on peut faire en sorte que Manu et Baptiste travaillent sur les mêmes versions Linux sans craindre des problèmes de compatibilité entre leurs codes respectifs.
Dans ce cas, le plus simple est de mettre en place un **Dockerfile,** document "chef d'orchestre", qui permettra à Manu et à Baptiste de monter une image similaire. Ce **Dockerfile** sera calqué sur les éléments présents en production. Du coup, Manu et Baptiste en plus de travailler sur un environnement identique, seront sur un environnement similaire à celui de la production !

4.

DevOps : Jenkins

Jenkins : c’est quoi ?

- Serveur d’intégration continue (CI)
- Utilisé pour automatiser des tâches comme :
  - La compilation du projet
  - L’exécution des tests
  - Le déploiement



Introduction à la programmation
Les fondamentaux

Xavier Tabuteau
formation.informatique.86@gmail.com

Les fondamentaux

- **Les éléments fondamentaux :**
  - Les variables
  - Les structures conditionnelles
  - Les structures répétitives
  - Les méthodes (fonctions)
  - Les tableaux (ou array)
  - Les classes
  - L’héritage
  - Les interfaces
  - Les énumérations
  - Les expressions lambda
  - Les collections
  - Les exceptions

1.

Introduction

Introduction

- Développer ne consiste pas à écrire des formules mathématiques (sauf si vous travailler sur des logiciels destinées aux sciences évidemment). On parle plutôt **d'algorithme**.
- Même si les langages sont différents, l'objectif principal d'un langage de programmation reste celui d'instruire la machine pour qu'elle produise des outputs conformes aux objectifs de l'application.
- On résume souvent par :
  - F(X) = Y, où :
    - X représente l'input
    - Y représente l'ouput
    - F() représente la fonction qui permet de transformer X en Y.

Introduction

- La programmation permet de résoudre un problème de manière automatisée grâce à l'application d'un algorithme :

Programme = Algorithme + Données

- Un algorithme est une suite d'instructions qui sont évaluées par le processeur sur lequel tourne le programme.
- Les instructions utilisées dans le programme représente le **code source**.
- Pour que le programme puisse être exécuté, il faut utiliser un langage que la machine peut comprendre : un **langage de programmation**.

Introduction

- Langages procéduraux / langages objets
  - Il existe des langages procéduraux tel que le langage C.
  - Il existe des langages fonctionnant à base d’objets tel quel le langage Java.

Introduction

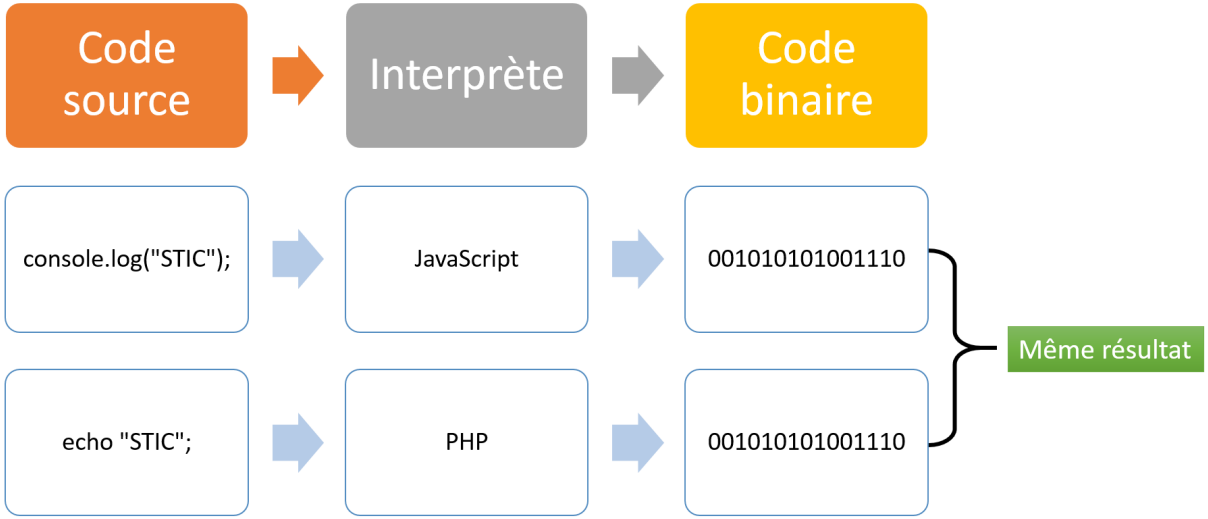
- Langage bas niveau vs langage de haut niveau
  - Un langage de bas niveau est un langage qui est considéré comme plus proche du langage machine (binaire) plutôt que du langage humain. Il est en général plus difficile à apprendre et à utiliser mais offre plus de possibilité d'interactions avec le hardware de la machine.
  - Un langage de haut niveau est le contraire, il se rapproche plus du langage humain et est par conséquent plus facile à appréhender. Cependant les interactions se voient limitées aux fonctionnalités que le langage met à disposition.

Introduction

- Compilation vs Interprétation
  - La compilation d’un programme consiste à transformer toutes les instructions en langage machine avant que le programme puisse être exécuté. Par conséquent il sera nécessaire de refaire la compilation après chaque modification du code source.
  - Si un langage n’est pas compilé, il est nécessaire d’utiliser un interprète qui traduit les instructions en temps réel (on run time). Dans ce cas le code source est lu à chaque exécution et par conséquent les changement apportés au code seront pris en compte directement. La contrainte réside dans le fait que le machine faisant tourner le programme doit disposer de l’interprète de celui-ci.

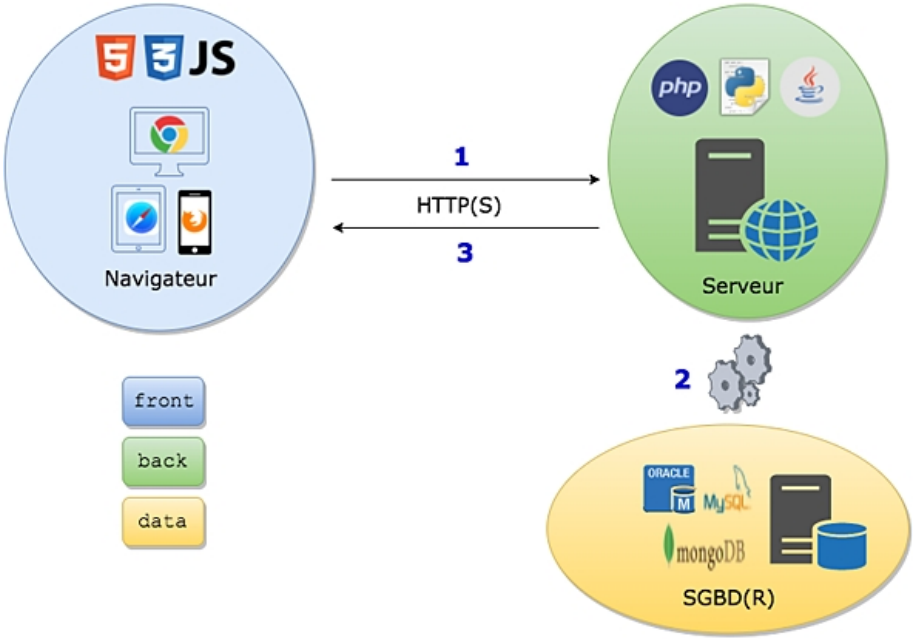
Introduction

- Alors c’est pas des 0 et des 1 le code ?



Introduction

- **Anatomie d'une application interactive**



Introduction

- **Composition de Java :**

Le langage Java

- Ecriture des programmes Java
- Orienté objet
- Compilation du code Java en byte code (langage machine portable en fichier .javac)

Le Machine Virtuelle Java

- Interprète et exécute le byte code
- La machine virtuelle Java (JVM) est disponible pour chaque système d'exploitation
- Un fichier en byte code peut être exécuté sur n'importe quelle JVM indépendamment de l'OS

La plateforme Java

- Ensemble de classes prédéfinies
- API (Application Programming Interface) disponible pour les développeurs

Introduction

- **Gestion de la mémoire**

- L’interpréteur Java (JVM) sait quels emplacements mémoires il a alloué.
- Il sait déterminer quand un objet alloué n’est plus référencé par un autre objet ou une variable.
- Ramasse-miette (« Garbage Collector ») : détecte et détruit ces objets non référencés (libération automatique de la mémoire).

Introduction

- **Java est un langage objet**

- Tout est classe et objet !
- En Java, tout ce qui est produit est sous forme de classes.
- Les fonctionnalités de base de Java sont disponible sous forme de classes.
- C’est au développeur d’identifier ce qui doit être créé par rapport au problème à résoudre !

L’identification des classes est issue d’un processus d’analyse.

Il est possible d’utiliser une méthodologie pour identifier les classes.

Par exemple : UML (Unified Modeling language)

Introduction

- **Java est un langage objet**

Exemple :

- Un garage entretient des voitures
- Un garage est référencé par sont N° de Siret, nom, adresse, propriétaire, chiffre d’affaire et nombre de salariés
- Les voitures possèdent des caractéristiques comme la marque, un numéro de série
- Les voitures possèdent toutes un moteur et un châssis
- Le châssis a notamment un numéro de série
- Le moteur a notamment une référence et une puissance
- Les voitures roulent, démarrent, freinent

Introduction

- **Java est un langage objet**

Exemple :

Le diagramme de classes UML correspondant aurait :

- Une classe Garage, avec ses propriétés (adresse, etc...)
- Une classe Voiture, avec ses propriétés (marque, etc...)
- Une classe Chassis, avec ses propriétés (poids , etc...)
- Une classe Moteur, avec ses propriétés (puissance , etc...)
- Une relation entre Garage et Voiture (entretien)
- Une relation d’agrégation entre Voiture et Chassis et Moteur, Voiture étant l’agrégat (composée)

Introduction

- **Java est un langage objet**

Exemple :

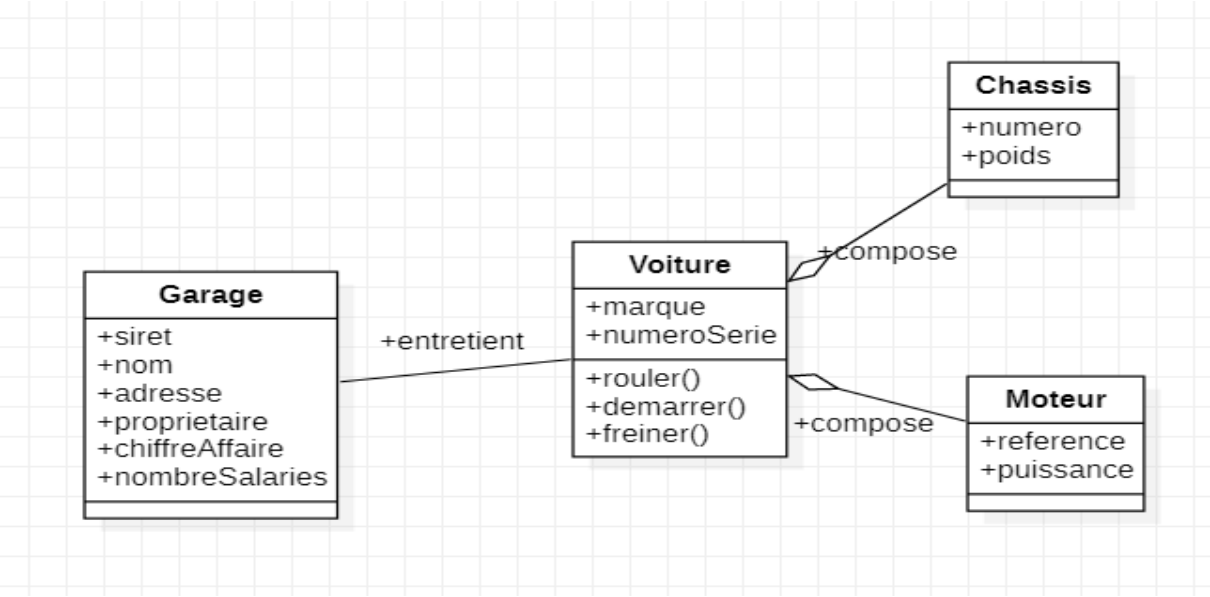
Faire le diagramme de classe avec StartUML : https://staruml.io/download

- Classe Garage avec ses attributs
- Classe Voiture, avec ses attributs et méthodes
- Classe Chassis, avec ses attributs
- Classe Moteur, avec ses attributs
- Relation entre Garage et Voiture (entretien)
- Relation d’agrégation entre Voiture et Chassis et entre Voiture et Moteur
  (Voiture comporte un châssis et un moteur)

Introduction

- **Java est un langage objet**

Exemple :



2.

Installations

Installation

Pour programmer en langage Java nous avons besoin d'installer une JVM (Java Virtual Machine), un JDK (Java Development Kit) ainsi qu'un IDE (Integrated Development Environment). Il existe plusieurs IDE tel que Eclipse, Visual Studio Code, Intellij, Apache Netbeans.

Nous installerons le JDK v17 LTS et v18 d'Oracle.
Si vous voulez utiliser une version maintenue dans la durée, utilisez la LTS (Long Term Support). Si vous voulez la dernière version, vous choisirez la 18.

Pour l'IDE nous installerons Apache Netbeans parce qu'il est gratuit et support les dernières versions du JDK de Java.

Installation

- **Installation de la JVM**

Lien de téléchargement : https://www.java.com/fr/download/manual.jsp

Exécuter le fichier téléchargé, puis laisser les options par défaut et laisser faire jusqu’au bout.

Installation

- **Installation du JDK v17 LTS et v18 d'Oracle**

Lien de téléchargement :
https://www.oracle.com/java/technologies/downloads/#jdk18-windows

Exécuter le fichier téléchargé, puis laisser les options par défaut et laisser faire jusqu'au bout. Vous pouvez éventuellement changer le dossier d'installation du JDK. Pensez a bien repérer l'emplacement de l'installation du JDK.

Remarque : il est possible d'installer plusieurs version de JDK sur une même machine.

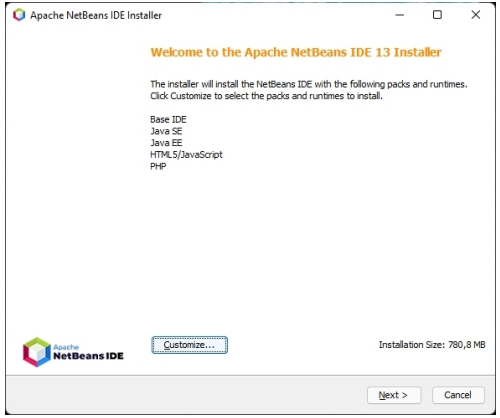
Installation

- **Installation d'Apache Netbeans**

Lien de téléchargement :
https://dlcdn.apache.org/netbeans/netbeans-installers/14/Apache-NetBeans-14-bin-windows-x64.exe

Après le téléchargement, exécuter le logiciel et suivre les étapes suivantes. Si une erreur intervient, indiquant qu'il ne trouve pas l'emplacement du JDK, il faut lancer l'installation via une fenêtre d'invite de commande en mode administrateur et saisir :

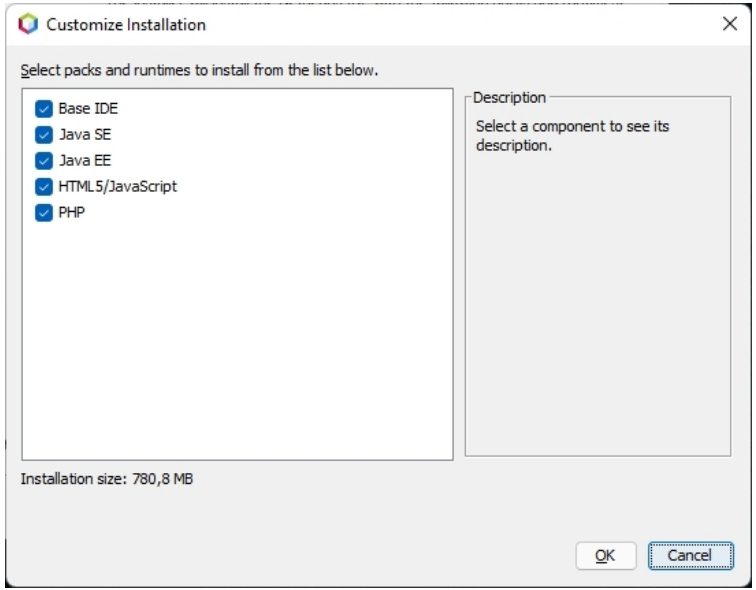
nomFichier.exe --javahome "emplacementJava"



Installation

- **Installation d'Apache Netbeans**

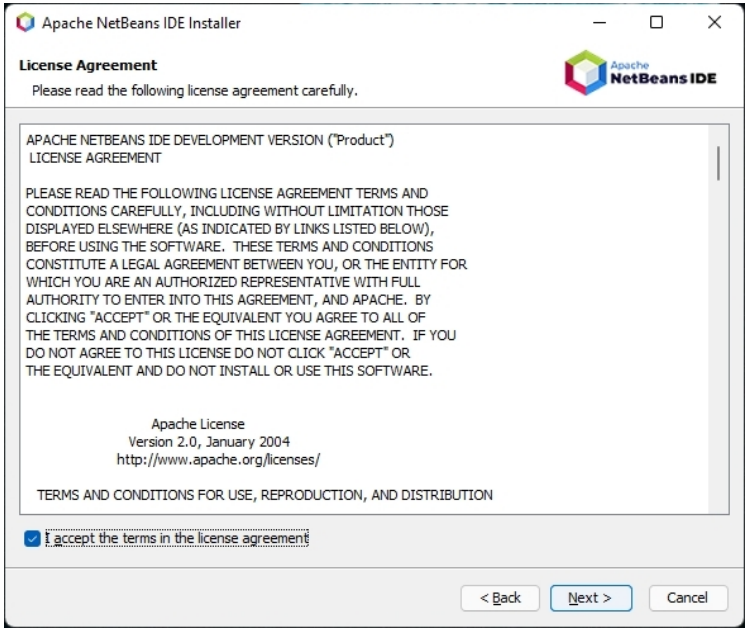
Cliquer sur le bouton « Cutomize », vérifier que tout est coché et valider.



Installation

- **Installation d'Apache Netbeans**

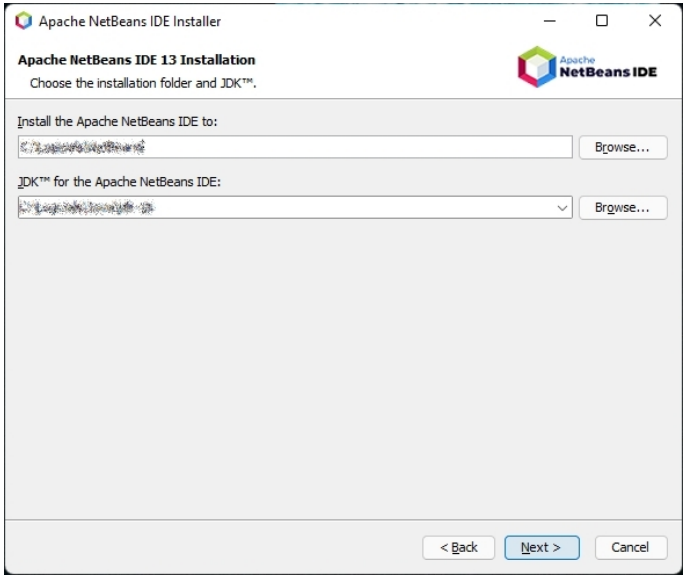
Valider les conditions d'utilisation puis cliquer sur le bouton « Next ».



Installation

- **Installation d'Apache Netbeans**

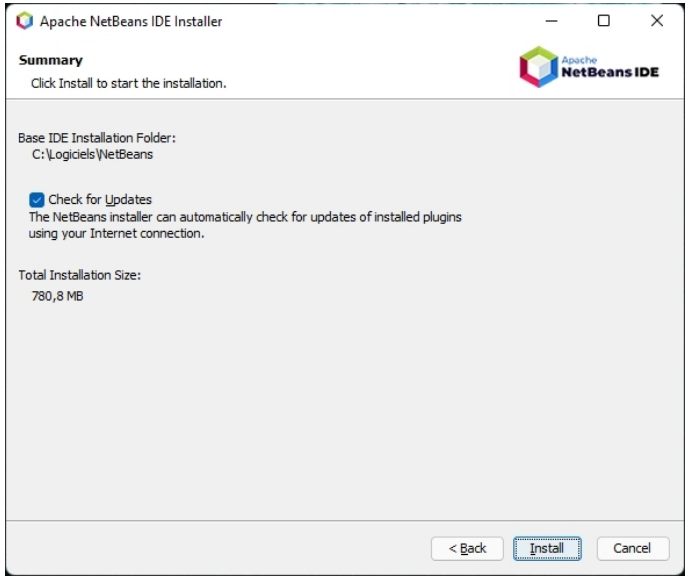
Choisir l'emplacement de l'installation de Netbeans et indiquer l'emplacement du JDK à utiliser par défaut au démarrage de Netbeans.



Installation

- **Installation d'Apache Netbeans**

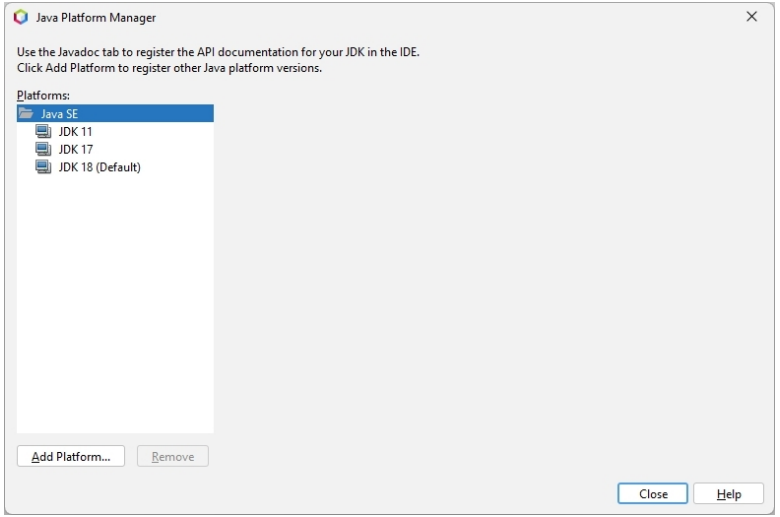
Cliquer sur le bouton « Install » afin de démarrer l'installation de Netbeans.
Puis « Finish » une fois celle-ci terminée.



Installation

- **Paramétrage d'Apache Netbeans**

Cliquer sur le menu « Tools » puis « Java Platforms »



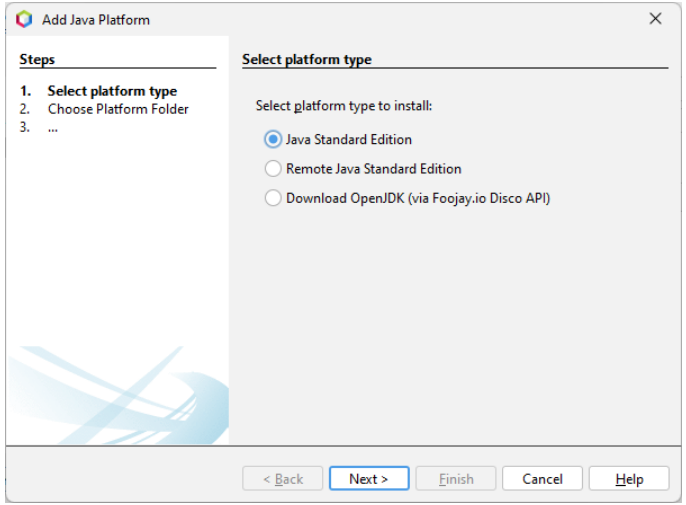
Vérifier les versions du JDK qui apparaissent. Pour ajouter un JDK, cliquer sur « Add Platform » et choisir l'emplacement du JDK à ajouter ou à télécharger.

Installation

- **Paramétrage d'Apache Netbeans**

Laisser sur « Java Standard Edition » sélectionnée pour ajouter un JDK déjà installé sur votre machine et cliquer sur le bouton « Next » (Nous utiliserons cette option).

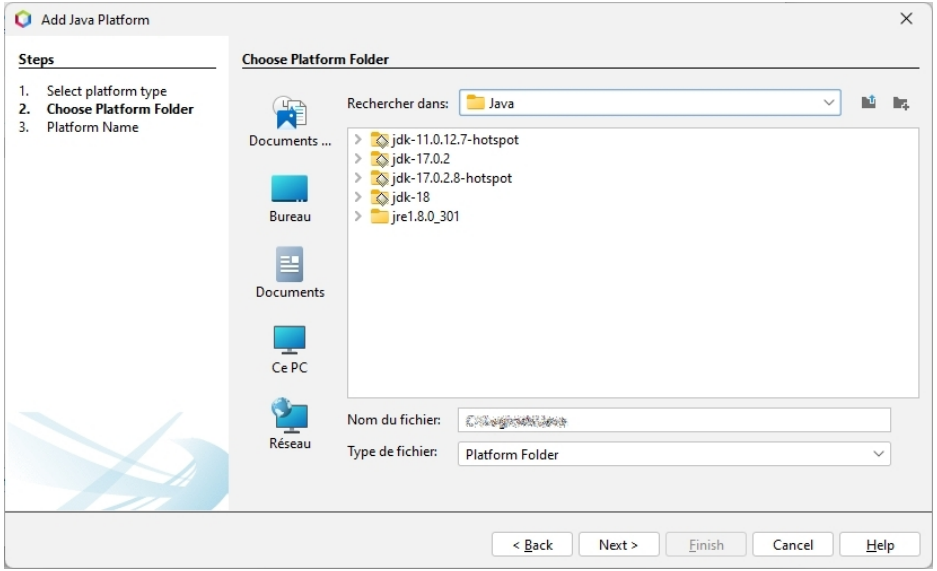
Choisir « Download OpenJDK » pour installer un JDK parmi une liste fournie par le logiciel.



Installation

- **Paramétrage d'Apache Netbeans**

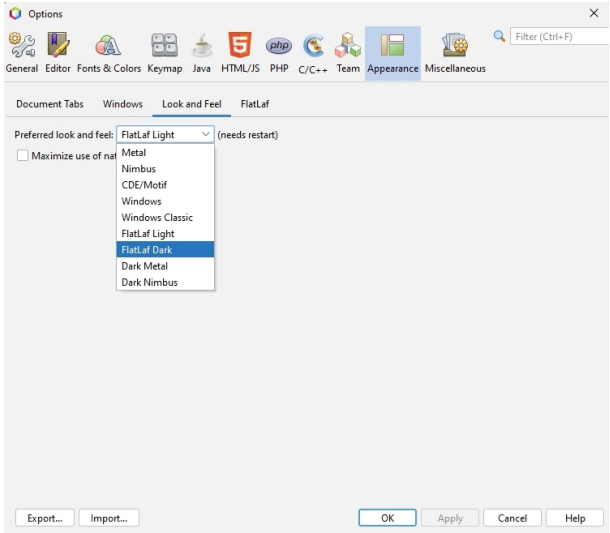
Indiquer l'emplacement du JDK déjà installé et cliquer sur « Next » puis « Finish » une fois l'ajout terminé.



Installation

- **Paramétrage d'Apache Netbeans**

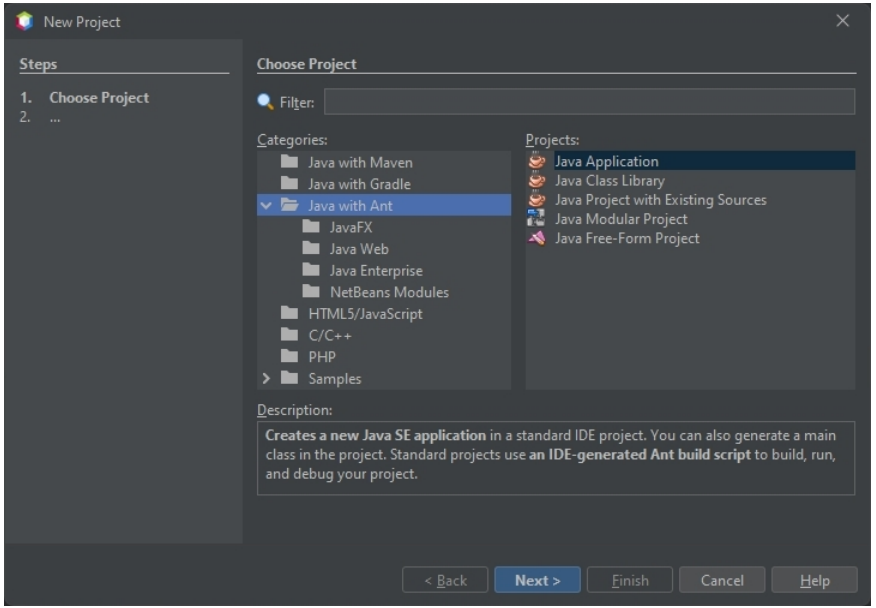
Pour ceux qui préfèrent un mode sombre pour programmer, cliquer sur « Tools » puis Options. Cliquer sur « Appearance » et « Look and Feel » puis choisir dans la liste le dark mode souhaité et valider avec « Apply » (cliquer sur le lien donné par le logiciel quand celui-ci demandera de redémarrer).



Installation

- **Utilisation d'Apache Netbeans**

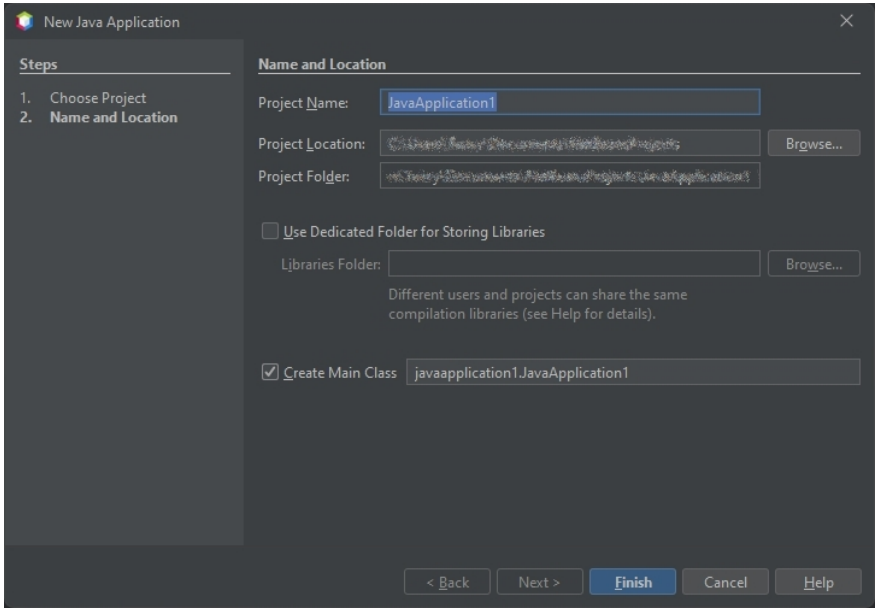
Pour commencer un projet, choisir le menu « File » puis « New Project », choisir « Java with Ant » et « Java Application ».



Installation

- **Utilisation d'Apache Netbeans**

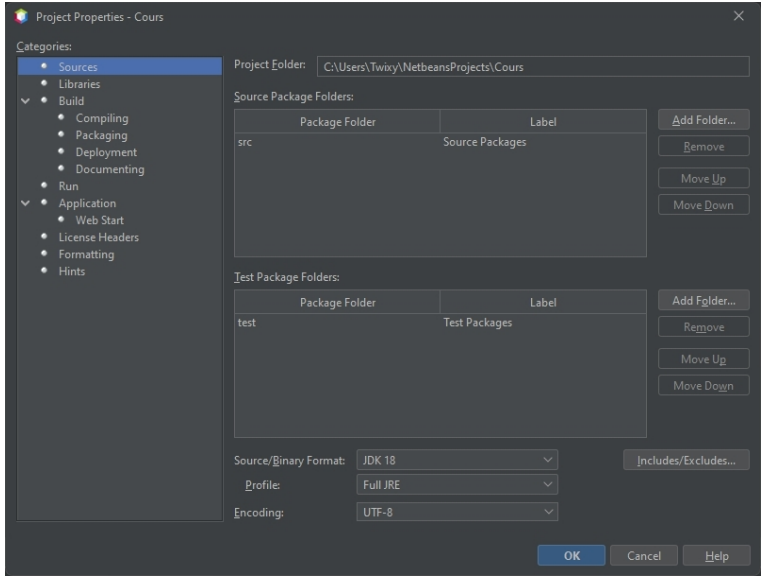
Choisir un nom de projet, un emplacement pour le projet et pour le dossier contenant le projet et cliquer sur le bouton « Finish ».



Installation

- **Changement de JDK dans un projet Apache Netbeans**

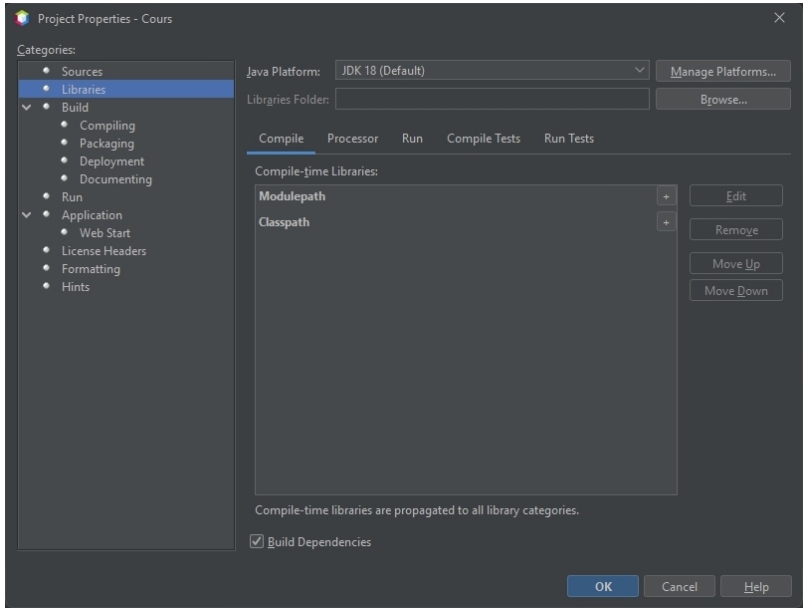
Cliquer sur « File » puis « Project Properties » (ou clic droit sur le nom du projet puis « Properties »). Dans la partie « Sources », vous pouvez modifier l'option « source/format binaire » (version JDK) que Netbeans va utiliser.



Installations

- **Changement de JDK dans un projet Apache Netbeans**

Dans la partie « Libraries » de la même fenêtre, modifier la « Java plate-forme » que le projet utilisera pour construire les classes.



3.

Les variables

Les variables

Une variable permet d'identifier une valeur avec un nom. Elle peut s'écrire avec les caractères de l'alphabet en minuscule, majuscule, les chiffres, et les caractères « _ » et « $ ». Elle ne peut pas commencer par un chiffre, ni avoir de caractères accentués ou spéciaux. Par convention, on écrit les variables en camel case.

Remarque : par convention, les constantes s'écrient en majuscule uniquement.

Utilité des variables :

- La déclaration d'une variable sert à créer une référence dans un emplacement en mémoire.
- L'affectation d'une variable sert à lui associer une valeur.

Remarque : Les types de base ne sont pas des objets, ils n'ont aucune méthode. Pour chaque type de base, il existe une classe correspondante riche en fonctionnalités.

Les variables

- Les types de variables :
  - Les **chaînes de caractères** (string) : elles sont utilisées pour représenter du texte.
  - Les **chiffres** (nombre entier, à virgule flottante, etc.) : ils sont utilisés surtout avec des opérateurs mathématiques.
  - Les **valeurs booléennes (en anglais : booleans)** : elles sont des valeurs dichotomiques (soit vrai, soit faux).
  - Les **tableaux** (array) : ils sont utilisés pour créer des listes avec des indices qui permettent de récupérer la valeur associée.
  - Les **objets** : ils sont des conteneurs qui peuvent inclure souvent tout type de données, y compris de sous-objets, des variables (propriétés / attributs), ou des fonctions (méthodes).

Les variables

- **Les types de base :**

Type	Taille	Description	Intervalle
char	2 octets	Une unité de code, suffisant à représenter un grand nombre de point de code, et même un caractère Unicode (UTF-16) 'b' '\u250C'	'\u0000' à '\uFFFF'
byte	1 octet	Un nombre entier de 8 bits (soit un octet) signé	-128 à 127
short	2 octets	Un nombre entier de 16 bits signé	-32 768 à 32 767
int	4 octets	Un nombre entier de 32 bits signé	−2 147 483 648 et +2 147 483 647
long	8 octets	Un nombre entier de 64 bits signé	−9 223 372 036 854 775 808 et +9 223 3

Les variables

- **Les types de base :**

Type	Taille	Description	Intervalle
float	4 octets	Un nombre à virgule flottante de 32 bits signé (simple précision)	de 2^{-149} (Float.MIN_VALUE) à $2^{128} - 2^{104}$ (Float.MAX_VALUE)
	0.0F		
	−∞ (Float.NEGATIVE_INFINITY)		
	+∞ (Float.POSITIVE_INFINITY)		
	pas un nombre (Float.NaN)		
Double	8 octets	Un nombre à virgule flottante de 64 bits signé (double précision)	de 2^{-1074} (Double.MIN_VALUE) à $2^{1024} - 2^{971}$ (Double.MAX_VALUE)
	0.0D		

Les variables

- **« Boxing » et « Unboxing » :**

Boxing : Conversion automatique du type primitif en son équivalent objet.

Exemple :

```
int valeurPrimitive = 100;
Integer valeurObjet = valeurPrimitive;
```

Unboxing : Conversion automatique de l’objet en son type primitif.

Exemple :

```
Integer valeurNumerique = 100;
int valeur = valeurNumerique
```

Les variables

- **Conversions de type :**

Il est possible de convertir un type de variable en un autre.

Exemple de conversion de type entre chaîne de caractères et valeur numérique :

```
Double monDouble = 3,14159;
String pi = Double.toString(monDouble);


String valeurNumerique = "100";
Integer valeur = Integer.valueOf(valeurNumerique);
```

Remarque : il est possible de forcer la conversion temporaire d’une variable, on appel cela le casting.

Syntaxe :

```
float valFloat = (float)valInt;
```

Les variables

- **Les opérateurs :**

Les opérateurs permettent de manipuler ou comparer des valeurs, notamment des variables. Parmi les opérateurs utilisés fréquemment dans tous les langages de programmation vous trouverez :

- L'opérateur d'**affecation**
- L'opérateur de **négation**
- Les opérateurs **arithmétiques**
- Les opérateurs **binaire (comparaison)**
- Les opérateurs **logiques** (AND et OR).

En Java il existe d'autres opérateurs très utilisés :

- Les opérateurs **unitaires**
- Les opérateurs **avec assignation**

Les variables

- **Opérateur d'affectation :**

L'affectation d'une variable avec le signe « = »
Syntaxe :

variable = valeur;

Remarque : Pour que l'affectation soit correcte il faut que les 2 opérandes soient de même type.

Exemple :

Integer variable1 = 100;
Integer variable2 = variable1;

Affectation multiple :

« type » var1, var2, …, varN = … = var2 = var1;

Les variables

- **Opérateurs arithméthiques :**

Les opérateurs arithméthiques de base sont : +, -, *, / et % (modulo = reste de la division).
Remarque : l'opérateur "+" permet aussi de concatener des chaînes de caractères.

- **Opérateurs unitaires (++ et --) :**

Les opérateurs unitaires permettent de raccourcir les expressions d'incrémentation et de décrémentation ainsi que faire des post ou pré affectations selon la position de l'opérateur unitaire par rapport à la variable concernée.

- **Opérateur de négation :**

! → Négation du résultat d'une évaluation.

Les variables

- **Opérateurs binaires (comparaison) :**

Egalité	→	==
Différent	→	!=
Inférieur	→	<
Supérieur	→	>
Inférieur ou égale	→	<=
Supérieur ou égale	→	>=

- **Opérateur logique :**

&&	→	ET logique.
||	→	OU logique.

Les variables

- **Opérateurs avec assignation :**

Cela permet de raccourcir une expression lorsque la variable ce trouve des 2 cotés de l’expression. Cela fonctionne avec tout les opérateurs de base.

Exemple :

i = i + 2; → i += 2;

4.

Les structures conditionnelles

Les structures conditionnelles

- **Les conditions :**

Les structures de contrôle permettent d'exécuter seulement certaines instructions d'un programme selon la vérification d'une ou plusieurs conditions. Il existe 3 structures conditionnelles. Ces structures sont imbriquables entre elles et les autres types de structures.

- if / else
- switch / case
- (? :)

Les structures conditionnelles

- **if / else :**

Cette structure exécute un bloc d'instructions si la condition au niveau du if est vérifiée. Le else est optionnel et permet d'exécuter un autre bloc d'instructions si la condition du if correspondant est fausse. Toute évaluation qui retourne un booléen peut être utilisée avec le if / else.

Syntaxe d'un if / else :

```
if (variable == valeur) {
        instructions si la condition est vraie;
}
else {
        instrcutions si la condition est fausse;
}
```

Les structures conditionnelles

- **switch / case :**

Le switch apporte de la clarté au code par rapport au « if / else » dans le cas où un traitement différent doit être appliqué selon les différentes valeurs d'une variable. Plutôt que d'imbriquer des « if / else » on préférera utilisée le switch / case. Cette structure peut également être utilisé pour tester des chaînes de caractères.

Important : Il faut stipuler la sortie du bloc switch dès que l'on a exécuter le traitement voulu. Pour cela nous utilisons l'instruction break.
Si aucun cas ne correspond à la valeur testée, alors c'est celui par défaut (default) qui est exécuté.

Les structures conditionnelles

- **switch / case :**

Syntaxe d'un switch / case :

```
switch (variable) {
        case "A":   instructions;
                        ...
                        break;
        case "B":   instructions;
                        ...
                        break;
...
        default:    instructions;
                        ...
                        break;
}
```

Les structures conditionnelles

- **switch / case :**

A partir du JDK 14, la structure switch a changé et n’a plus besoin de break. Un case peut accepter plusieurs valeurs et le « : » devient « -> ». De plus, s’il y a plusieurs instructions pour un « case », ces instructions devront être placées entre accolades comme n’importe quel autre bloc d’instructions.

Syntaxe :

```
switch (valeur) {
        case "A", "B", ...          -> instruction;

        case "C", ... ->            {
                                        instructions;
                                        ...;
                                        }
        ...
        default ->   instruction;
}
```

Les structures conditionnelles

- **switch / case :**

Ce nouveau switch / case permet aussi d'être utilisé sous forme d'expression. Cela permet par exemple, d'initialiser une variable avec une valeur qui dépend de plusieurs conditions. S'il faut plusieurs lignes on utilisera les accolades pour délimiter le bloc d'instructions et, pour retourner la valeur on doit employer « yield » (comme un return).
Remarque : dans ce cas d'utilisation attention au ; à la fin du switch.

Syntaxe :

```
<type> variable = switch (valeur) {
            case "A", "B", ...          -> valeur_de_retour;
            case "C", ... ->            {
                                                instructions;
                                                ...;
                                                yield valeur_de_retour;
                                                }
            ...
            default ->   instruction;
};
```

Les structures conditionnelles

- **(? :) :**

Cette structure conditionnelle est appelée ternaire. Elle permet de remplacer une structure conditionnelle if / else dont le résultat peut être affecté directement à une variable. Le type de variable doit correspondre avec la valeur retournée.

Syntaxe de la structure :

« type » resultat = (a==b) ? « valeur retournée si vrai » : « valeur retournée si faux »;

5.

Les structures répétitives

Les structures répétitives

- **Les boucles :**

Ces structures sont appelées boucles. Elles sont la base d’un concept très utile en programmation : l'**itération.** Cela permet d’exécuter plusieurs fois des instructions. Il existe 4 structures répétitives. Ces structures sont imbriquables entre elles et les autres structures.

- for
- do … while
- while
- for (:)

Les structures répétitives

- **for :**

Ces structures existe dans beaucoup de langages. Elle fonctionne ainsi :

1 - L’initialisation est exécutée, une seul fois.
2 - Le test est évalué, et s’il est faux on quitte la boucle.
3 - Le bloc de commandes du for est exécuté.
4 - L’incrémentation est effectuée.
5 - Le test est évalué, et s’il est faux on quitte la boucle, sinon on revient à l’étape 3.

Syntaxe d’une boucle for :

```
for (initialisation variable; test de sortie;  incrémentation) {
                traitements a faire en boucle si test est vrai;
}
```

Les structures répétitives

- **do … while :**

La particularité de cette boucle, c’est que le bloc d’instruction est exécuté une fois minimum. Le while situé à la fin de la boucle permet d’évalué la condition de sortie de la boucle.

Structure d’un do … while :

```
Initialisation valeurTest;

do {
        instructions;
        incrémentation de la valeurTest;
}
while (valeurTest < valeur);
```

Les structures répétitives

- **While :**

La particularité de cette boucle, c’est qu’elle peut ne pas être exécutée. Le while permet d’évaluer la condition de sortie de la boucle.

Structure d’un while :

```
Initialisation de valeurTest;

while (valeurTest > valeur) {
        instructions;
        décrémentation de valeurTest;
}
```

Les structures répétitives

- **For(:) :**

Elle est appelée boucle for « intelligente ». La particularité de cette boucle, c'est qu'elle s'utilise avec des tableaux ou des collections uniquement, et qu'il n'y a pas à gérer la sortie de la boucle. A chaque itération, l'élément du tableau / collection est chargé dans la variable. La progression est automatique jusqu'à la fin du tableau / collection.

Structure d'un for (:) :

```
for (<type> variable : tableau ou collection) {
        instructions;
}
```

Les structures répétitives

- **Les instructions break et continue :**

continue : permet d’arrêter les instructions du bloc de la boucle et de recommencer la boucle avec la valeur suivante.

break : permet de sortir définitivement de la boucle en cours.

Ces deux instructions sont utilisable dans tous les types de boucles.

6.

Les méthodes

Les méthodes

Les méthodes sont des fonctions situées dans une classe. Or Java ne fonctionne qu’avec des classes, donc avec des méthodes. Elles représentent une sorte de programme dans le programme, car elles sont la première forme d'organisation du code. On utilise des méthodes pour regrouper des instructions et les appeler sur demande : chaque fois qu'on a besoin de ces instructions, il suffira d'appeler la méthode au lieu de répéter toutes les instructions. Pour accomplir ce rôle, le cycle de vie d'une méthode se divise en deux phases :

1. Une phase unique dans laquelle la méthode est définie.
2. On définit à ce stade toutes les instructions qui doivent être groupées pour obtenir le résultat souhaité.

3. Une phase d’appel de cette méthode qui peut être répéter autant de fois que désiré.
4. On demande à la méthode de mener à bien toutes les instructions dont elle se compose à un moment donnée dans la logique de notre programme.

Les méthodes

Une méthode peut optionnellement retourner une valeur.

Si elle ne retourne pas de valeur, elle doit être du type « void ».
Si elle retourne une valeur, la dernière instruction dans la méthode devra être « return » suivi de la variable ou valeur à retourner qui doit correspondre au type de la méthode.

Elle peut aussi avoir des paramètres (variables) de façon optionnelle.

S'il y a des paramètres alors l'appel de la méthode doit contenir le même nombre et type de paramètres.

Exemple de méthode Java très utilisée :
System.out.println(x); → x représente ce qu'il faut afficher dans la console Java.

Les méthodes

Syntaxe de la définition d’une méthode :

```
« type » nomMethode(type variable1, …) {
        instructions;
        …
        return valeurOuVariable;
}
```

Syntaxe de l’appel d’une méthode :

```
resultat = nomMethode(variable1, …);
```

Remarque : resultat doit être du même type que la méthode.

7.

Les tableaux

Les tableaux

Un tableau de base est une suite ordonnée ou non d'élément du même type. Le type du tableaux peut être ce qu'on veut, même une classe personnalisée. Il est possible de les passer en paramètre d'une méthode.

La particularité d'un tableau, c'est qu'il faut le déclarer avant utilisation, et son allocation mémoire est faite à ce moment là. Il est possible de déclarer et alimenter un tableau sans préciser sa taille au moment de son initialisation.

Pour récupérer la taille du tableau on utilisera la propriété length.

Le nombre d'éléments chargés dans un tableaux peut être différent de la taille de celui-ci.
On peut accéder à un élément d'un tableau via son index. Le premier élément d'un tableau à toujours l'index 0.

L'accès à un index du tableau pour lequel il n'y a pas d'objet chargé retourne la valeur null.
Une tentative d'accès en dehors des bornes du tableau lève une exception (erreur).

Les tableaux

Il est possible de créer des tableaux à plusieurs dimensions. Ce n’est pas forcément très pratique et lisible selon la situation.

Syntaxe d’un tableau :

Visibilité type[] nom = new type[nbrElements]; // 1 dimension
Visibilité type[][] nom = new type[nbrElements1][nbrElements2]; // 2 dimensions
Ou
Visibilité type nom[] = new type[nbrElements]; // 1 dimension
Visibilité type nom[][] = new type[nbrElements1][nbrElements2]; // 2 dimensions

Pour initialiser directement un tableau avec des éléments, on peut faire :

Visibilité type[] nom = {elem1, elem2, elem3, ...};
Ou
Visibilité type nom[] = {elem1, elem2, elem3, ...};

8.

Les classes

Les classes

Une classe est le modèle (un moule) qui va recevoir essentiellement les attributs (variables) et méthodes (fonctions) de l’objet. On peut créer autant d’objet qu’on veut à partir d’une classe. Ils seront indépendant les uns des autres.

Dans Java il est possible de définir plusieurs classes dans un même fichiers suffixé par .java mais, parmi ces classes, une seule sera définie comme « public » et c’est elle qui donnera le nom au fichier.

Une classe a un nom commençant par une majuscule.

Les classes

Exemple :

Client.java

```
1  package gestion;
2  public class Client {
3      private String nomclient;
4      private Integer statut;
5      public Boolean creerClient(String nomclient, Integer statut) {
6          this.nomclient=nomclient;
7          this.statut=statut;
8          return true;
9      }
10     public Boolean changerStatut(Integer statut) {
11         if (this.statut==0) {
12             return false;
13         }
14         this.statut=statut;
15         return true;
16     }
17 }
18 class Adresse{
19     public String adresse;
20 }
21
```

Les classes

- **La visiblité :**

On trouve 3 mots-clés pour la visibilité des attributs et méthodes mais 4 visibilités différentes.
public : la portée est totale, même en dehors du package.
private : opposé de public, inaccessible depuis l'extérieur d'une classe.
protected : accès depuis toutes les classes du même package ou des classes qui héritent de celle-ci.
« friendly » : En cas d'absence de mot clé, l'accès est par défaut depuis toutes les classes du même package (Il est appelé friendly mais ne s'écrit pas).

Une classe ne peut avoir que la visibilité public ou « friendly »

```
2 public class Client {
3     private String nomclient;
4     private Integer statut;
```

```
23        Client monclient = new Client();
24        monclient.nomclient="Sarl";
```

Erreur de compilation !!!

Les classes

- **Les constructeurs :**

Le constructeur est une méthode particulière permettant de créer un objet. En cas de passage de paramètres dans le constructeur, il permet aussi de l'initialiser avec ceux-ci.
Un constructeur est une méthode spéciale qui porte obligatoirement le nom de la classe dans laquelle il est défini. Il n'a pas type de retour.

Il est appelé automatiquement lorsqu'on demande la création d'un objet de la classe de ce constructeur (utilisation du mot clé « new »).

Il peut y avoir plusieurs constructeurs dans une classe. Ils se différencient par le nombre et type de paramètres passés. Comme une surcharge de méthode classique.

Il n'est pas obligatoire, mais très utile.

Il peut être appelé depuis un autre constructeur de façon explicite avec le mot-clé « this ». Comme avec des méthodes surchargées classique.

Les classes

- **Le mot-clé this :**

Le mot-clé « this » fait référence à l’objet en cours.

```
2  public class Client {
3      private String nomclient;
4      private Integer statut;
5      private Adresse adresse;
6      public Boolean creerClient(String nomclient, Integer statut, String adresse) {
7          this.nomclient=nomclient;
8          this.statut=statut;
9          this.adresse = new Adresse();
10         this.adresse.setAdresse(adresse);
11         return true;
12     }
13     public Boolean changerStatut(Integer statut) {
14         if (this.statut==0) {
15             return false;
16         }
17         this.statut=statut;
18         return true;
19     }
20     public String getAdresse() {
21         return this.adresse.getAdresse();
22     }
23 }
24 class Adresse{
25     private String adresse;
26     public String getAdresse() {
27         return adresse;
28     }
29     public void setAdresse(String adresse) {
30         this.adresse = adresse;
31     }
```

Déclaration de la propriété de type Adresse

Stockage de l'adresse dans l'objet de type Adresse, nommé adresse

Accesseur, ou getter, permettant l'accès au contenu de l'adresse via la propriété de type Adresse

Propriété adresse de la classe Adresse, avec ses deux accesseurs.

Les classes

- **L'encapsulation :**

L'encapsulation en Java est un terme qui recouvre une chose simple : Protéger les données d'un objet. Cela permet de créer un couplage faible au lieu d'un couplage fort entre objets et classes. Le couplage faible est recommandé car cela permet d'avoir une meilleur maintenabilité du code. Une classe peut contenir des données de type public et private. Si on déclare toutes les propriétés en public, alors l'encapsulation est inutile mais nous auront un couplage fort.

```
1  package nonencapsule;
2  public class Client{
3      public void crediterCompte(double credit) {
4          CompteBancaire cb = new CompteBancaire();
5          cb.crediter(100.12);
6          cb.valeur=200;
7      }
8  }
9  class CompteBancaire {
10     public double valeur;
11     public double crediter(double credit) {
12         this.valeur+=credit;
13         return this.valeur;
14     }
15 }
16
```

Accessible directement !

Non protégée !

Les classes

- **L'encapsulation :**

Les accesseurs (ou Getters et Setters en anglais) sont des méthodes qui permettent de ne pas accéder directement à la variable de la classe qui elle, est privée. C'est donc un couplage faible.

```
1  package nonencapsule;
2  public class Client{
3      public void crediterCompte(double credit) {
4          CompteBancaire cb = new CompteBancai
5          cb.crediter(100.12);
6      }
7  }
8  class CompteBancaire {
9      private double valeur;
10     public double crediter(double credit) {
11         this.valeur+=credit;
12         return this.valeur;
13     }
14     public double getValeur() {
15         return valeur;
16     }
17 /*  public void setValeur(double valeur) {
18         this.valeur = valeur;
19     }*/
20 }
21
```

Accesseur pour accéder à la valeur → **Encapsulation**

Les classes

- **JavaBean :**

C’est une représentation unique d’une entité fonctionnelle utilisable à divers endroits du programme.

Exemple :

- Un client
- Un article
- Une commande

Acronymes similaires pour dire JavaBean :

- POJO (Plain Old Java Object)
- DTO (Data Transfer Object)
- DAO (Data Access Object)

Les classes

- **JavaBean :**

Un JavaBean est une classe qui doit respecter les conventions suivantes :

- Implémenter l’interface Serializable.
- Proposer un constructeur sans paramètre.
- Disposer d’accesseur publiques pour les attributs private (getters et setters)
- Classe non déclarer « final ».

Les classes

- **JavaBean :**

Exemple d’utilisation :

- Une classe ArticleService réalisera des opérations sur un article (création, modification, suppression, …).
- Les méthodes de cette classe travaillerons sur le JavaBean Article.

Les classes

- **Surcharge :**

La surcharge en Java c'est la capacité d'une classe à accepter d'avoir des méthodes avec le même nom. Il est possible d'utilisé le mot clé **this** pour appeler une méthode surchargée dans une autre.

Contrainte : différenciation sur le nombre et/ou la nature des types qui sont déclarés pour cette méthode.

Attention : le type de retour ne peut servir de discriminant !

Exemple de surcharge dans les méthodes du JDK :

La méthode valueOf de la classe String

Les classes

- **Les mots clés static et final :**

static : lie un attribut ou une méthode à la classe. Cela à pour utilité de pouvoir définir des propriétés au niveau global de la classe. Donc ces attributs ou méthodes sont des objets uniques. Modifier un attribut « static » d'une classe, modifie cet attributs pour tout les objets instanciés de cette classe. Par convention, on y accède via le nom de la classe au lieu du nom de l'objet.

final : Ce mot clé s'utilise sur une variable, une méthode ou une classe. Après initialisation sur une variable celle-ci ne peut plus être changée ! Cela provoquerait une erreur de compilation. En résumé, ce mot clé permet de déclarer une constante.

Remarque : associé « static » et « final » est possible. Cela permet de créer une constante unique à la classe.

Le mot clé final appliqué sur une méthode, permet l'interdiction de la redéfinition de cette méthode dans le cas de l'héritage.
Le mot clé final appliqué sur une classe, permet l'interdiction de l'héritage depuis celle-ci.

Sera vu plus tard ;)

9.

L’héritage

L’héritage

C’est une notion indissociable du langage Java et de la Programmation Orienté Objet (POO). La question à se poser quand on parle d’héritage est :

Est-ce que ma classe est une sorte d’autre classe plus générique ?

Exemple : Un Smartphone est une sorte de téléphone.

Cela induit donc une notion de parenté. La classe la plus générique sera appelée classe mère et la classe qui en héritera sera appelée classe fille. Les types génériques en Java héritent tous d’une classe nommée Object. Seuls les types de bases n’héritent pas de la classe Object. Par conséquent, la création d’une classe qui n’hérite de rien explicitement, hérite quand même de la classe Object. L’instruction « extends Object » n’est donc pas nécessaire car implicite.

Syntaxe de l’héritage :

```
class Smartphone extends Telephone {
…
}
```

L’héritage

Avec l’héritage, on peut redéfinir les méthodes héritées. L’annotation « @Override » n’est pas obligatoire mais c’est une bonne pratique pour indiquer :

- Aux développeurs qu’il s’agit d’une méthode redéfinie.
- Au compilateur qu’il s’agit d’une méthode redéfinie pour qu’il vérifie son exactitude.

Le mot clé **super** : permet d’accéder aux attributs et méthodes de la classe mère si la visibilité de celle-ci le permet.

Le mot clé **this** : permet de référencer les attributs et méthodes de l’objet en cours et de l’objet dont il hérite si la visibilité de celui-ci le permet.

L’héritage

Il est possible de bloquer l’héritage en Java via le mot clé final sur la classe. En effet, cela provoque un blocage complet. Aucune classe ne peut hériter d’une classe définie avec final. On peut aussi limiter le blocage au niveau d’une méthode ou même simplement d’un attribut (devient une constante).

Si une méthode est définie dans la classe mère, et qu’elle n’est pas redéfinie dans la classe fille, alors cette méthode est accessible depuis un objet de la classe fille si la visibilité le permet.

L'héritage

- **L'abstraction :**

Dans une classe, il est possible de déclarer des méthodes sans code associé.
→ parce qu'il n'est pas possible de fournir un code pour ce niveau d'abstraction.
→ parce qu'on veut mettre en œuvre le polymorphisme.

Exemple :
une classe Voiture avec une méthode rouler()
une classe Camion avec une méthode rouler()
une classe Vehicule avec une méthode abstraite rouler(), c'est à dire sans code car on ne sait pas dire comment le véhicule roule. C'est trop abstrait à ce niveau.

Remarque : si on déclare une méthode abstraite, alors la classe aussi doit être abstraite.

Syntaxe de classe abstraite :

```
abstract class NomClasse {
...
}
```

L'héritage

- **Le polymorphisme :**

Le polymorphisme dans java veut simplement dire qu'une classe peut prendre plusieurs formes et c'est d'autant plus vrai avec les classes qui hérite d'une classe supérieure.

```
25 public class Garage{
26     public static void main(String[] args) {
27         Vehicule v1 = new Voiture(true);
28         Vehicule v2 = new Camion(false);
29         v1.rouler();
30         v2.rouler();
31     }
32 }
```

Création de 2 objets de type Vehicule :
1 compatible avec Voiture et
1 compatible avec Camion

Appel de la méthode rouler() possible car rouler() a été définie dans Vehicule

A l'exécution, c'est la méthode définie sur l'objet réel qui est appelée ! ➔ **polymorphisme**

10.

Les interfaces

Les interfaces

Une interface peut représenter un point d'échange entre un fournisseur et un ou plusieurs clients (contrat).
L'interface comporte uniquement les déclarations des méthodes présentes (il n'y a pas de code interne aux méthodes) et ne peut pas comporter des variables. Par contre, on pourra définir des constantes ou des énumérations pour créer des constantes dans l'interface.
La classe qui implémente l'interface, définit les méthodes de celle-ci.
Par convention, on nomme l'interface avec un « I » en préfixe.
L'utilisation d'une interface permet de créer une dépendance faible entre l'interface et la classe qui l'implémente. Une classe qui appelle une autre classe, crée une dépendance forte à cette classe.

Syntaxe d'une interface :

```
interface INomInterface {
        type nomMethode1(paramètres);
        type nomMethode2();
        ...
}
```

Les interfaces

Exemple avec métaphore :

Plusieurs fabricants décident de proposer une façade de machine à laver standard.
L'objectif est de retrouver les mêmes fonctionnalités au même endroit sur n'importe qu'elle machine à laver.
Les fabricants définissent l'interface de la façade, qu'il peuvent donner à un sous-traitant pour la fabrication (plan, modèle).

Le sous-traitant va savoir quoi faire pour construire les façades, sans connaître les fonctionnalités propre à chaque fabriquant (démarrage, arrêt, programmes, ouverture, …)

Chaque fabricant pourra utiliser la façade pour construire ses modèle de machine à laver.

→ La façade est l'interface (contrat).
→ la machine à laver propre à un fabricant est une implémentation de cette interface.

Les interfaces

Une interface qui n’a qu’une seule méthode abstraite est appelée une interface fonctionnelle. Java fournit une annotation @FunctionalInterface, qui est utilisée pour déclarer une interface comme interface fonctionnelle.
En Java, la classe qui implémentera une interface devra utiliser la syntaxe suivante :

```
class MaClasse implements IMonInterface {
            ...
}
```

Une classe qui implémente une interface doit implémenter toutes les méthodes de celle-ci. Sinon, il y a une erreur de compilation ou il faut déclarer la classe abstraite.
Une même classe peut hériter d’une autre classe et implémenter une ou plusieurs interfaces.
Mais une classe ne peut pas hériter de plusieurs classes. L’héritage multiple n’est pas autorisé en Java.
Remarque : une interface peut héritée d’une autre interface en utilisant la même syntaxe que celle utilisée pour les classes.

Les interfaces

Depuis la version 8 de Java, il est possible d'ajouter des méthodes statiques dans une interface et des méthodes « default ».

Les méthodes « default » et « static » ont pour particularité d'être définie dans l'interface et pourront être utilisées directement (ou redéfinie) par la classe qui implémente l'interface.

Lors d'une création d'une interface dérivée de l'interface contenant une méthode « default » on peut :

→ Ne rien mentionner : l'interface hérite de la méthode par défaut.
→ Redéfinir la méthode
→ Redéclarer la méthode : la méthode devient abstraite.

Les méthodes « static » sont définies dans l'interface comme les méthodes « default » mais elles ne peuvent pas être redéclarées lors de la création d'une interface dérivée. Par conséquent, elles ne peuvent pas être abstraites. En effet, il n'est pas possible en Java de créer des méthodes statiques abstraites.

Les interfaces

Remarque : Les classes abstraites et les interfaces se ressemblent. La principale différence est que les interfaces ne peuvent pas avoir d'état (variables d'instances) ni de constructeur. Elles ne peuvent avoir que des constantes.

11.

Les énumérations

Les énumérations

Une énumération est un ensemble fini de valeurs. La particularité de l'énumération c'est qu'on ne peut utiliser que les valeurs définies au sein de celle-ci. C'est une sorte de liste de constantes, d'où le fait décrire le valeurs en majuscule dans l'énumération. Une énumération utilise le type « enum » de Java.

Remarque : Pour que le type « enum » soit reconnu il faut utiliser Java 8 minimum.

Syntaxe d'une énumération :

```
enum Nom {
        VAL1,
        VAL2,
        …,
        VALn
}
```

```
Appels :    Nom.VAL1;                    // retourne VAL1.
            Nom.VAL1.ordinal();          // retourne l'index de VAL1
```

12.

Les expressions lambda

Les expressions lambda

Les expressions lambda sont introduites dans Java 8 et sont utilisées dans la programmation fonctionnelle. Une expression lambda est donc une fonction qui peut être créée sans appartenir à aucune classe. Une expression lambda peut être transmise comme s'il s'agissait d'un objet et exécutée à la demande.
Une expression lambda est traitée comme une fonction, donc le compilateur ne crée pas de fichier « .class ».
L'expression lambda fournit l'implémentation d'une interface fonctionnelle.

Syntaxe :

(liste d'arguments) → { instructions; }

La liste d'arguments accepte 0 à n paramètre.

Les expressions lambda

Le type d’un paramètre est facultatif. Le compilateur peut déduire le type de la valeur du paramètre.

Pas besoin d’utiliser des accolades dans le corps d’une expression si le corps contient une seule instruction.

Le mot clé « return » est facultatif si le corps a une seule expression.

13.

Les collections

Les collections

Les tableaux Java vu précédemment permettent de stocker des « collections » de données. Il existe en Java une façon plus simple de gérer ces collections. « Java Collections » désigne un ensemble d'interface et de classes permettant de stocker, trier et traiter les données (classes utiles avec de multiples méthodes).

La plupart des collections se trouvent dans le package java.util

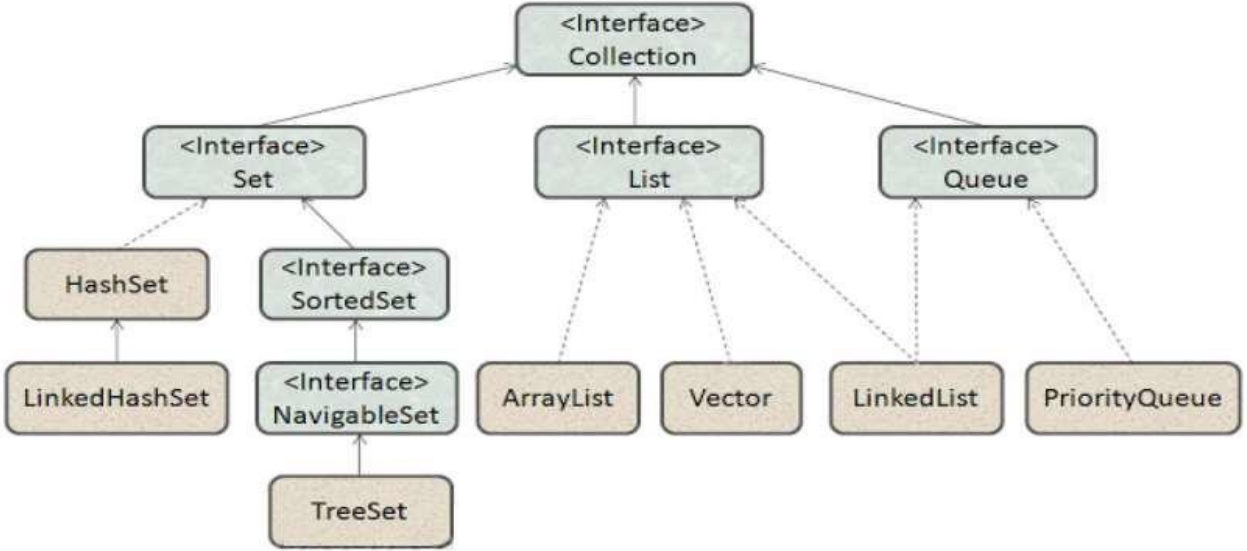
Une collection est un « objet » qui collecte des éléments dans une liste.
Une collection représente un ensemble de données de même type.

La composition des collections de Java est une structure hiérarchique établie à partir de deux grands ensembles.

- Collection (java.util.Collection)
- Map (java.util.Map)

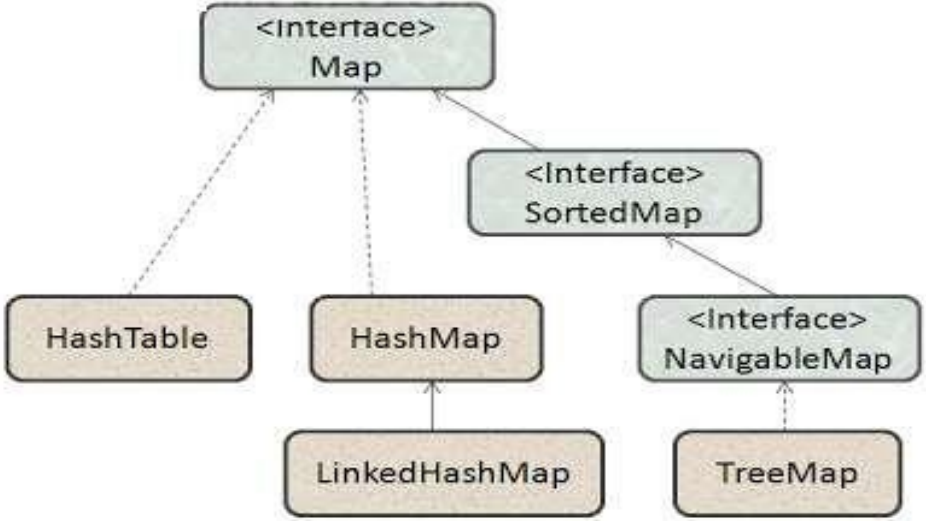
Les collections

- **Java.util.Collection :**



Les collections

- **Java.util.Map :**



Les collections

- **List :**

La classe ArrayList implémente l'interface List, qui elle-même hérite de l'interface Collection. Elle dispose donc, de l'ensemble des méthodes de l'interface List et de l'interface Collection. Une ArrayList peut-être vue et traitée comme une interface List.

On peut écrire :

```
List listeDeChaines = new ArrayList(); // polymorphisme !
```

Il n'y a pas encore le type de donnée à stocker de précisé dans cette liste.
On le précise en utilisant les « Generics » de Java :

```
List<String> listeDeChaines = new ArrayList<>();
```

L'appel d'une méthode d'une liste se fait comme suit :

```
maListe.nomDeLaMethode(parametres)
```

Les collections

- **List :**

Liste de méthodes utiles pour gérer les ArrayList :

add(x, val)	→ charge la valeur val dans la liste à la position x. (si x n'est pas préciser, charge la valeur au premier index disponible).
addAll(liste2)	→ charge la liste2 dans une liste.
get(x)	→ accède à l'élément de la liste ayant l'index x.
set(x, val)	→ modifie l'élément à la position x de la liste par la valeur val.
indexOf(val)	→ obtient l'index du premier élément ayant pour valeur val.
lastIndexOf(val)	→ obtient l'index du dernier élément ayant pour valeur val.
contains(val)	→ vérifie si la valeur val est contenu dans la liste (retourne vrai ou faux).
remove(x)	→ supprime de la liste l'élément situé à l'index x.
remove(val)	→ supprime de la liste le premier élément égale à val.
clear()	→ supprime tous les éléments de la liste.
size()	→ retourne la taille de la liste.

Les collections

- **Set :**

Le type Set en Java est une liste ayant pour particularité de contenir que des éléments uniques). Si on converti une liste de type List en Set les doublons de la List seront supprimés dans le Set. Les collections sont convertible d'un type à l'autre.

Attention : pas de garantie de l'ordre d'insertion !

Les collections

- **Map :**

L'interface Map permet de stocker un ensemble de paires « clé / valeur » dans une table de hachage. Une Map ne peut pas contenir des éléments dupliqués, chaque clé à une unique valeur. Map est implémenté avec les classes suivantes :

HashMap → pas de garantie de l'ordre d'insertion.
LinkedHashMap → garantie l'ordre d'insertion.
TreeMap → stockage des éléments triés selon leur valeur.

L'interface SortedMap hérite de Map et implémente les méthodes pour ordonner les éléments dans l'ordre croissant et décroissant.

Remarque : TreeMap est une implémentation de SortedMap.

Les collections

- **Map :**

Liste de méthodes utiles pour gérer les Map :

Entry → représente une entrée du Map, c’est à dire un ensemble clé / valeur.
EntrySet() → retourne un objet Set contenant les paires clé / valeurs du Map.
put(x, y) → ajout d’un ensemble clé / valeur.
GetKey() → obtient la clé de l’entrée en cours.
GetValue() → obtient la valeur de l’entrée en cours.
setValue(val) → charge la valeur de l’entrée en cours avec la valeur val.

Les collections

- **Le tri d’une collections**

Le tri d’une collection se fait par la méthode sort().

Exemple : Collections.sort(liste);

Attention : si le type d’objet utilisé n’implémente pas l’interface Comparable, ou si on souhaite trier selon un ordre différent, alors il faut fournir une implémentation spécifique de l’interface Comparator !

Les collections

- **L’itération de collection :**

Il existe plusieurs possibilités d’itérer sur une liste en Java.

- Utilisation de l’interface Iterator
- Utiliser la boucle « intelligente » for (:)
- Utiliser les boucles classiques (for (;;), while, do...while)
- Utiliser la méthode forEach() avec une expression lambda

14.

Les exceptions

Les exceptions

Les exceptions c’est la gestions des erreurs.
Dans un langage non objet on va traiter les erreurs de cette façon :

```
retour = fonction1(parametre x);
si (retour == erreur) {
                traiter l’erreur;
}
```

En langage objet, non traitons l’erreur via une classe nommée Exception.

- Une exception est un objet instancié de la classe Exception.
- Il faut informer le système de ce qui déclenche la levée d’une exception.
- Il faut savoir récupérer et traiter une exception.

Toutes les exceptions levées dans une méthode et qui n’ont pas été traitées localement, doivent être ajoutées à la signature de la méthode via le mot-clé « throws » qui permet de déléguer la responsabilité des erreurs à la méthode appelante.

Les exceptions

- **try / catch / finally :**

La récupération se fait en encapsulant le code risquant de monter une exception avec les instructions try / catch / finally.
Syntaxe d'un try catch :

```
try {
        instructions pouvant provoquer une exception;
}
catch (Exception1 e) {
        instructions si une exception est survenue.
}
catch (Exception2 e) {
        instructions si une exception est survenue.
}
...
finally {
        instruction à exécuter qu'une exception soit survenue ou non.
}
```

Remarque : le finally est très utile pour libérer des ressources (ex : mémoires ou fichiers).

Les exceptions

- **try / catch avec instanceof :**

Une alternative à l'utilisation de plusieurs blocs « catch » consiste à utiliser instanceof :

Syntaxe avec instanceof :

```
try {
        instructions pouvant provoquer une exception;
}
catch (Exception e) {
        if (e instanceof Exception1)
                instructions si Exception1 est survenue.
        if (e instanceof Exception2)
                instructions si Exception2 est survenue.
}
...
finally {
        instruction à exécuter qu'une exception soit survenue ou non.
}
```

Les exceptions

- **Les exceptions avec ressource :**

Le traitement d'exception avec ressource permet de libérer automatiquement la ressource utilisée à la fin du « try / catch ». Il n'est pas nécessaire d'avoir un bloc finally pour libérer la ressource. La ressource s'indique à la ligne du try comme une création d'objet classique. Cette objet disparaîtra à la fin du « try / catch ».

Syntaxe d'un try avec ressources :

```
try (<type> obj = new UneClasse(<type> param, ...) {
                instructions pouvant provoquer une exception;
}
catch (Exception1 e) {
                        instructions si Exception1 est survenue.
}
...
```

Les exceptions

- **Levée d’execption :**

La signature d’une méthode doit inclure la déclaration des diverses exceptions levées dans le code qui ne sont pas traitées localement, et ceci se réalise avec le l’instruction « throws ».

Syntaxe d’une levée d’exceptions :

```
<Visibilité> <Type> nomMethode(<Type> param) throws Exception1, Exception2 {
        instructions;
        if (condition 1) {
                throw new Exception1("Message de l’exception 1");
        }
        if (condition 2) {
                throw new Exception2("Message de l’exception 2");
        }
        ...
}
```

Les exceptions

- **Les exceptions personnalisées :**

Pour des besoins applicatifs spécifiques, il est possible de créer soit même sa gestion d’exceptions en Java.

Les classes d’exceptions personnalisées doivent hériter de la classe Exception, ou d’une classe qui hérite de cette classe.

La gestion de ces exceptions est identique aux autres.

MODULE JAVA

Java - JEE

Marc GILBERT
marc.gilbert@wijin.tech

1.

Introduction

Introduction

- Java SE : Standard Edition → packages de base (java.lang, java.io, java.math, java.util, java.sql, …)

- Java EE : Entreprise Edition → extension de la plateforme standard

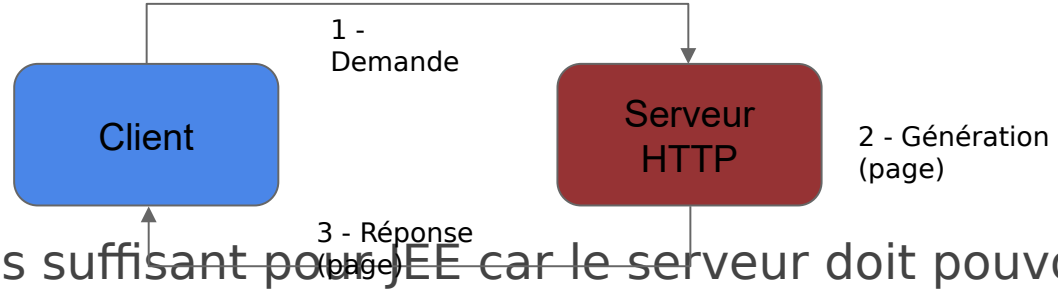
  - **Objectif :** Faciliter le développement d'application web déployées et exécutées sur un **serveur d'application**

Introduction

- Java EE : Exécutable avec une JRE mais nécessite en complément des librairies
- Depuis 2018, on parle de Jakarta EE
  - Oracle a confié la maintenance à Apache

Introduction

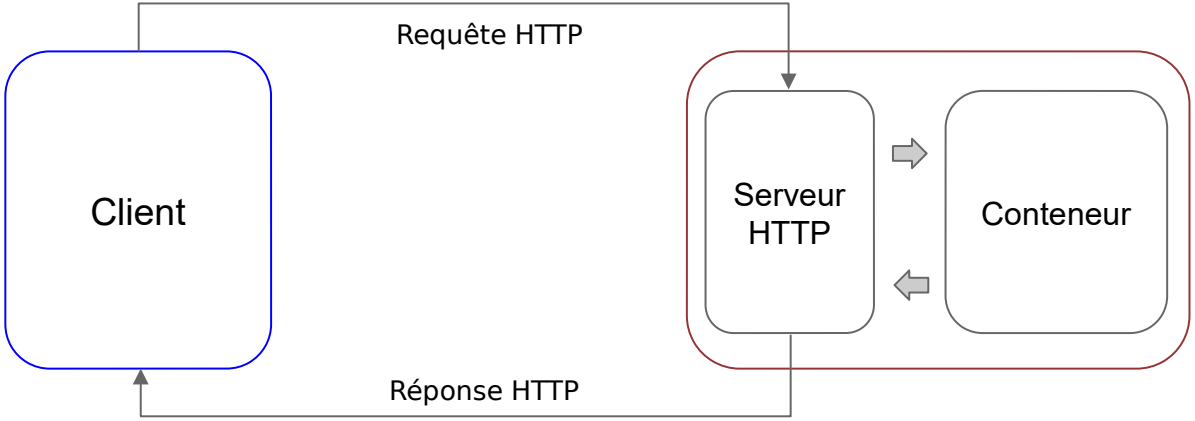
- Echanges HTTP : Entre un client et un serveur
  - Client → Navigateur
  - Serveur → Serveur HTTP



- Pas suffisant pour JEE car le serveur doit pouvoir effectuer d'autres tâches en dehors de l'affichage de pages !

Introduction

- Echanges HTTP pour une application WEB : Entre un client et un serveur
  - Client → Navigateur
  - Serveur → **Serveur d'application**



2.

Le modèle MVC

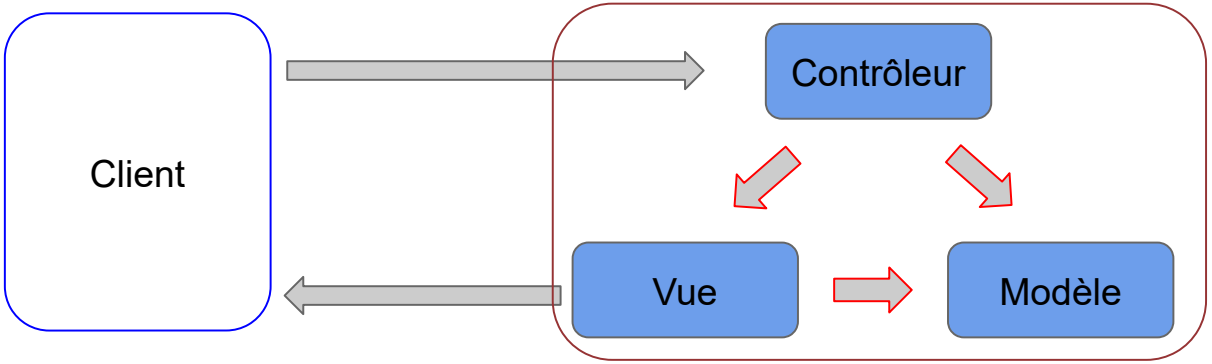
Le modèle MVC

- MVC = Modèle - Vue - Contrôleur (Model - View - Controller)

- Design Pattern : découper l'application en couches pour séparer les responsabilités
  - Traitement, stockage et mise à jour des données : **couche "Modèle"**
  - Interaction utilisateur et présentation : **couche "Vue"**
  - Contrôles des actions et des données : **couche "Contrôleur"**

Le modèle MVC

- MVC = Modèle - Vue - Contrôleur (Model - View - Controller)



Le modèle MVC

- MVC = Modèle - Vue - Contrôleur (Model - View - Controller)

- Exemple de mise en place du MVC sans framework :

  - **“Modèle”** : Traitements et données en **Java** (“beans” et “DAO”)

  - **“Vue”** : Pages **JSP** (Java Server Pages)

  - **“Contrôleur”** : **Servlets** Java

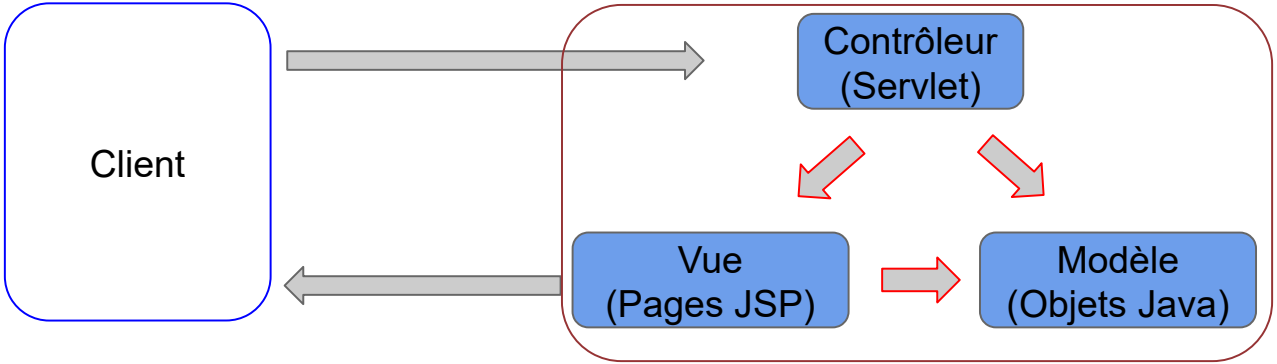
Le modèle MVC

- MVC = Modèle - Vue - Contrôleur (Model - View - Controller)

- Servlet : objet permettant d'intercepter les requêtes faites par un client et de gérer la réponse
  - Méthodes pour scruter les requêtes HTTP
  - Fait appel aux traitements de la couche “Modèle”
  - Demande à la couche “Vue” de retourner le résultat au client

Le modèle MVC

- Exemple de mise en place du MVC sans framework :



3.

Servlet Java

Servlet Java

- Echanges HTTP : verbes GET et POST
- GET : pour récupérer une ressource via son URI
  - Possibilité de positionner des paramètres
  - Lorsque que le serveur reçoit une requête HTTP GET, il retourne la ressource demandée

Servlet Java

- Echanges HTTP : verbes GET et POST
- POST : pour soumettre des données aux serveurs
  - Pour modifier la ressource (création, modification, suppression)

Servlet Java

- Mécanisme sur le serveur d’application
  - Réception de la requête HTTP par le serveur HTTP
  - Transmission au conteneur de servlets
    - Transformation de la requête : objet HttpServletRequest
      - Contient le requête et donne accès aux informations de celle-ci (header et body)
    - Initialisation de la réponse : objet HttpServletResponse
      - Permet de personnaliser les informations de la réponse (header et body)

Servlet Java

- Servlet ?
  - Une servlet = une classe Java
    - Doit permettre le traitement de requêtes et la personnalisation de réponses
    - En synthèse : doit permettre de recevoir une requête HTTP envoyée par un client, et de retourner une réponse HTTP à ce client

Servlet Java

- Servlet
  - Package **javax.servlet**
    - Interface mère Servlet
  - Package **javax.servlet.http**
    - Classe HttpServlet → Classe abstraite
    - Faire hériter nos classes “Servlet” de cette classe !

- Servlet
  - Redéfinition de méthodes dans nos classes Servlet :
    - doGet() → gestion des requêtes HTTP GET
    - doPost() → gestion des requêtes HTTP POST
    - Utilisation de la bonne méthode ? rôle de la méthode service()
      - Exécution automatique
      - Lecture de l'objet HttpServletRequest et distribution de la requête HTTP à la bonne méthode doXXX() en fonction du type

- Servlet
  - Déclaration des Servlets
    - Pour que l’application les connaisse et les expose !
    - Nécessite un lien entre la **servlet** et une **URL**
    - Permet de **diriger** la requête HTTP vers la bonne Servlet
    - Déclaration dans un fichier : **web.xml**

Servlet Java

- Servlet
  - Déclaration des Servlets
    - Fichier **web.xml** positionné dans un répertoire **WEB-INF** sous src/main/webapp

Servlet Java

- Servlet

  - Définition d’une Servlet

```
<servlet>
        <servlet-name>UserServlet</servlet-name>
        <servlet-class>fr.m2i.crm.servlet.UserServlet</servlet-
class>
        </servlet>
```

- Servlet

  - Définition d’une Servlet - options

```
<description>Description de la Servlet</description>
<init-param>
    <param-name>auteur</param-name>
    <param-value>Marc</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
```

- Servlet
  - Mapping d’une Servlet

```
<servlet-mapping>
    <servlet-name>UserServlet</servlet-name>
    <url-pattern>/userServlet</url-pattern>
</servlet-mapping>
```

- Servlet
  - Cycle de vie d’une Servlet
    - Lors de la première sollicitation d’une Servlet ou lors du démarrage de l'application web, le conteneur de Servlets crée une instance de cette Servlet
    - Elle est conservée en mémoire pendant toute la durée de vie de l'application
    - La même instance de la Servlet est utilisée pour chaque requête entrante dont l’URL correspond au pattern d'URL défini pour la Servlet

4.

Servlet et Vue

Servlet et Vue

- Mise en place de la couche vue sans framework : JSP
  - JSP ?
    - Page qui contient des **balises** HTML et des **balises** JSP
    - Extension de fichier “.jsp”
    - Exécution **côté serveur** !

Servlet et Vue

- Mise en place de la couche vue sans framework : JSP
  - JSP : technologie de la plateforme Java EE
  - Combine les technologies HTML, XML, Servlet et JavaBeans pour créer des vues dynamiques

Servlet et Vue

- Mise en place de la couche vue sans framework : JSP

  - **Pourquoi** ?

    - Simplifier la partie présentation (trop lourd pour la Servlet)

    - Séparation claire entre le code de **contrôle** et la **présentation**, telle que recommandé par le pattern MVC

Servlet et Vue

- Cycle de vie d’une JSP
  - Une page JSP est traduite et compilée en Servlet (classe Java héritant de HttpServlet)
  - La Servlet ainsi générée est utilisée durant l'existence de l'application
  - Si la JSP est modifiée, la Servlet est générée et compilée de nouveau

Servlet et Vue

- Cycle de vie d’une JSP
  - **Astuce** : on peut trouver le code généré et compilé par le serveur d’application → avec Tomcat, répertoire “work”

Servlet et Vue

- Association Servlet / JSP
  - Servlets Java développées : **contrôleurs**
  - Servlets générées à partir de JSP : **vues**
  - **Association dans les Servlets Java développées pour indiquer la JSP en charge de la présentation**

```
this.getServletContext().getRequestDispatcher(
"/ma_jsp.jsp").forward( request, response );
```

5.

Echange de données

Echange de données

- Données issues du serveur : attributs
  - Modification de la requête **par le serveur** pour y ajouter des attributs (méthode **setAttribute()** sur la requête)
  - Récupération possible dans la JSP
- Données issues du clients : paramètres
  - Récupération de paramètres **fournis par le client** (méthode **getParameter()** sur la requête)

6.

Technologie JSP

JSP

- Différentes manières de coder dans une JSP :
  - Code Java → **à éviter**
    - Balises pour la déclaration : <%! %> (déclaration de variables et méthodes)
    - Balises commentaires : <%-- --%>
    - Balises Scriptlet : <% %> (code Java)
    - Balise d'expression : <%= %> (affichage dans la présentation)
  - EL ("Expression Language") → **à préférer**
    - ${ expression }
    - Syntaxe simple

JSP

- Utilisation des EL :
  - Manipulation des beans Java :
    - ${monBean.attribut}
    - “monBean” = nom du bean Java
    - “attribut” = attribut (propriété) du bean Java
    - → équivalent au code Java : monBean.getAttribut()

JSP

- Utilisation des EL :
  - Manipulation des beans Java :
    - ${monBean.attribut}
    - Protection contre les valeurs “null” → pas d’affichage si “monBean” est “null” ou si la valeur de “attribut” est “null” (pas d’erreur)

- Utilisation des EL :
  - Manipulation des collections Java :
    - Exemple d'une liste Java :

```
List<String> customers = new ArrayList<>();
customers.add(customer1);
```

    - En "EL", accès aux éléments de la liste : ${customers.get(i)}
      - "i" représentant la position de l'élément (à partir de 0)

7.

JSTL

JSTL

- JSTL : JSP Standard Tag Library
  - Ensemble de tags utilitaires
    - Afficher une expression et sécuriser les formulaires contre les failles XSS (affichage avec balise “c:out”)
    - Gérer dynamiquement les différents liens et URL (redirection avec “balise c:url”)
    - Import dans les pages (balise “c:import”)
    - Condition à l'affichage du résultat de la validation (balise “c:if”)
    - Parcours d’une collection (balise “c:foreach”)
    - …

JSTL

- JSTL : JSP Standard Tag Library
  - Pour utilisation :
    - 1. Ajout de la librairie **jstl**
      - Exemple sur projet Maven :

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

    - 2. Déclaration **taglib** dans les fichiers JSP :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

JSTL

- JSTL : JSP Standard Tag Library
  - Documentation : https://javaee.github.io/jstl-api/

8.

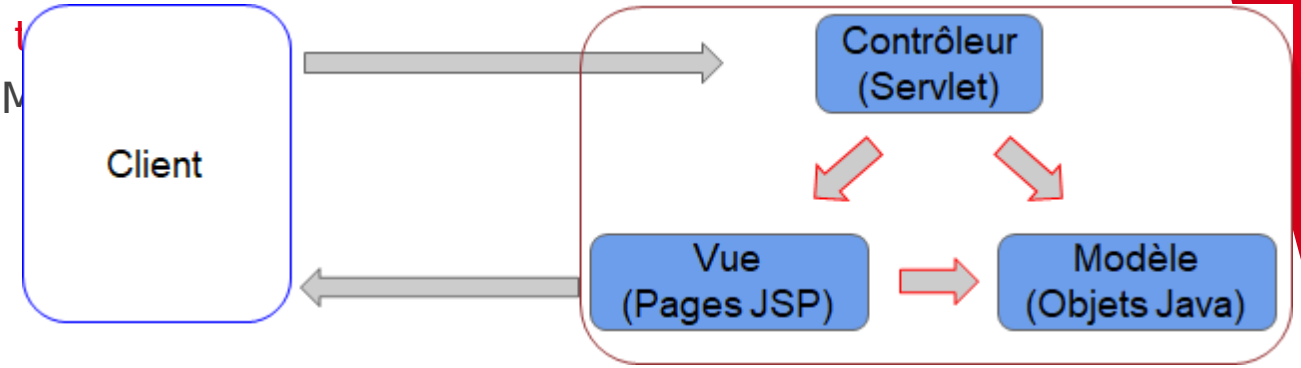
Traitement de formulaire

Traitement de formulaire

- Rappel sur modèle MVC

  - La Servlet est un contrôleur chargé de diriger les requêtes vers les traitements correspondants

  - Ces t
  » du M



Traitement de formulaire

- Eléments nécessaires pour le traitement d'un formulaire

- Un objet (bean Java) pour stocker les données validées du formulaire

- Un objet « Métier » pour récupérer et valider les données du formulaire à fait partie de la couche « **Modèle** » du MVC
  - Chargé également d'indiquer les erreurs

  - Une Servlet capable de rediriger vers la vue formulaire en cas d'erreur pour afficher celle(s)-ci

9.

Gestion de session

Gestion de session

- Pourquoi ?
  - HTTP = protocole sans état (“stateless”)
  - Par défaut, **pas de lien** entre les requêtes d’un même client !
  - La session a pour but de mémoriser des informations pour un client donné (informations récupérables sur les requêtes faites par ce client)

Gestion de session

- Session en JEE
  - **Session** = espace mémoire alloué pour un client
  - **Fermeture** d'une session :
    - Au bout d'un certain temps d'inactivité
    - Sur déconnexion
    - …
  - Objet Java **HttpSession** récupérable depuis la requête

Gestion de session

- Dans la Servlet (ou bean Java ayant accès la requête) :
  - Récupération de la session :

    HttpSession session = request.getSession();

  - Récupération d'un attribut dans la session :

    session.getAttribute("nomAttribut");

  - Positionnement d'un attribut dans la session :

    session.setAttribute("nomAttribut", valeur);

Gestion de session

- Dans la JSP :
  - Récupération de la session dans une expression « EL » :

    sessionScope.nomAttribut

10.

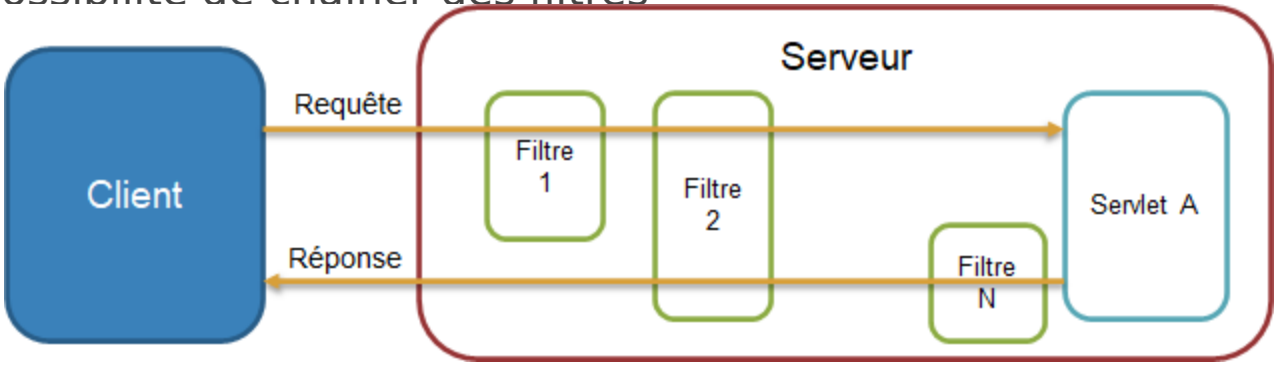
Gestion des filtres et des listeners

Filtre

▶Filtre ? :

▶Permettre de prétraiter ou post-traiter les données d'une requête

▶Possibilité de chaîner des filtres



Filtre

- Mise en œuvre :

- Déclaration dans fichier **web.xml**

  - Définition

```
<filter>
    <filter-name>LoadingFilter</filter-name>

<filter-class>fr.m2i.crm.filter.LoadingFilter</filter-class>
    </filter>
```

Filtre

- Mise en œuvre :

- Déclaration dans fichier **web.xml**

  - Mapping (URL)

```
<filter-mapping>
    <filter-name>LoadingFilter</filter-name
    <url-pattern>/*</url-patt
</filter-mapping>
```

Le filtre va s'appliquer sur les requêtes effectuées pour le pattern d'URL indiqué :

/* à toutes les URL
/createCustomer à l'URL createCustomer

Filtre

- Mise en œuvre :

- Classe Java
  - Implémentation de *javax.servlet.Filter*
  - Méthode init() : exécutée à l'initialisation du filtre par le serveur d'application
  - Méthode doFilter() : exécutée à chaque fois qu'une requête arrive pour l'URL du filtre
  - Méthode destroy() : exécutée lors de la destruction de l'objet par le serveur d'application

Filtre

- Mise en œuvre :

- Classe Java
  - Méthode doFilter() : la requête est interceptée par le listener !
    - Il faut donc propager cette requête en fin de méthode (chaînage)
      - chain.doFilter(request, res);

« chain » est de type FilterChain, est fourni en paramètre de la méthode doFilter()

L'appel de doFilter sur l'objet « chain » déclenche le transfert vers le suivant dans la chaîne (filtre suivant ou Servlet cible)

Listener

- Listener ?
- Objet notifié lors d’une modification de son environnement
- Design Pattern « Observateur »
- Objectif en JEE : avertir l’application d’un changement
- Différentes interfaces de listener
    - Changement d’état du ServletContext
    - Changement d’état des attributs stockés dans le ServletContext
    - Entrée/Sortie d’une requête
    - Changement d’état des attributs stockés dans un HttpServletRequest
    - ….

Listener

- Mise en œuvre

- Déclaration dans fichier **web.xml**

- Définition

```
<listener>
        <listener-class>fr.m2i.crm.config.MonListener</listener-class>
</listener>
```

Listener

- Mise en œuvre
- Classe Java à exemple pour listener sur changement d'état du ServletContext
  - Implémentation de *javax.servlet.ServletContextListener*
  - Méthode contextInitialized() : exécutée au lancement de l'application, avant le chargement des Servlets et des filtres à on y positionne ce qu'on souhaite faire
  - Méthode contextDestroyed() : exécutée à la destruction du contexte d'exécution

11.

Projet fil rouge CRM

Mise en place de JEE avec Servlet et JSP

Projet Fil Rouge

- Etape 1 :
  - Mise en place des premières Servlets pour traitement des requêtes HTTP GET, avec traitement de validation des données
  - Mise en place des premières Vues (pages JSP)
    - Création d'un client (formulaire)
    - Affichage d'un client
    - Création d'une commande (formulaire)
    - Affichage d'une commande
  - Récupération des paramètres de la requête issue du formulaire dans la Servlet pour contrôle et affichage d'un

Projet Fil Rouge

- Etape 1 :
  - Limites de cette étape 1 :
    - Comportement figé
    - Duplication de code
    - Pas d’affichage conditionnel

Projet Fil Rouge

- Etape 2 :
  - Utilisation de la librairie JSTL
  - Factorisation de code (formulaire de création d'un client)
  - Génération dynamique des pages Vues (pages JSP) avec utilisation des balises JSTL
    - Ajout d'un menu pour accéder aux formulaires de création
    - Affichage conditionnel des résultats en fonction des valeurs des attributs présent dans la requête

Projet Fil Rouge

- Etape 2 :
  - Limites de cette étape 2 :
    - L’accès direct aux pages JSP est possible (pas de masquage de la technologie utilisée)
    - Pas de normalisation HTTP (seules les requêtes GET sont traitées)
    - Les traitements de validation ne sont pas en dehors des Servlets

Projet Fil Rouge

- Etape 3 :
  - Masquage des JSP : positionnement dans le répertoire WEB-INF pour qu'elles ne soient pas accessible directement par un utilisateur
    - Modification des servlets pour que la méthode doGet gère l'affichage de la vue
  - Envoi des formulaires avec le verbe http POST
    - Modification des fichiers JSP
    - Modification des servlets pour implémenter une méthode doPost()

Projet Fil Rouge

- Etape 3 :
  - Extraction des traitements de validation dans des classes métiers dédiées
    - Appel depuis les servlets
    - Gestion des erreurs
  - Traitement de la redirection depuis les Servlets
    - En fonction du résultat du traitement, redirection vers le formulaire (en cas d'erreur) ou vers la page d'affichage
    - Modification des pages JSP pour gérer l'affichage des erreurs et des valeurs saisies sur les formulaires ré-affichés

Projet Fil Rouge

- Etape 3 :
  - Limites de cette étape 3 :
    - Pas de conservation et donc pas d’accès aux clients et commandes créées
    - Pas de gestion de session (pour conserver des informations d’une requête à l‘autre)

Projet Fil Rouge

- Etape 4 :
  - Gestion de la session
    - Enregistrement dans la session des clients et commandes créées
    - Affichage de la liste des clients et des commandes présents en session --> nouvelles Servlets et vues JSP
    - Suppression d’un client ou d’une commande de la session à nouvelles Servlets

Projet Fil Rouge

- Etape 4 :
  - Limites de cette étape 4 :
    - Pas de gestion spécifique des exceptions de validation
    - Pas de lien avec la base de données
      - Pas de chargement des données
      - Pas de persistances des ajouts/suppressions

Projet Fil Rouge

- Etape 5 :
  - Spécialisation des exceptions de validation de formulaires POST
    - Modification des classes métiers
  - Lien avec la couche d'accès aux données
    - Persistance des données
    - Stockage de la DaoFactory au niveau du contexte applicatif (ServletContext) avec un listener
  - Chargement des données en session au démarrage de l'application (Filtre)

MODULE TEST

Tester son code de façon unitaire

Marc GILBERT
marc.gilbert@wijin.tech

1.

Les tests unitaires

A quoi servent les tests unitaires ?

- Lors de la réalisation d'un projet informatique, les **tests unitaires (« TU »)** vont permettre de garantir la qualité du résultat
  - Valider unitairement les fonctionnalités d'un produit
  - Organisation à mettre en place
  - Ne remplace pas une recette complète !

A quoi servent les tests unitaires ?

- **Règle d'or** : Plus tôt une anomalie est détectée, moins son impact sur la charge du projet est important

Périmètre des tests unitaires

- Réalisés par les développeurs
- 2 contraintes
  - L'environnement de développement
  - La compréhension fonctionnelle du sujet traité
- Un TU = un test d'une sous-partie du logiciel (limité au seul périmètre de la fonctionnalité développée)
- Les TU ont leurs limites

Principe des tests unitaires

- Principe de base
  - Codage d’une fonctionnalité / correction du code si nécessaire
  - Exécution avec un échantillon de données
  - Observation du résultat
  - → Itération jusqu’à obtenir le résultat attendu

2.

Organisation des TU

Organisation des tests unitaires

- Organisations possibles :
  - Le développeur livré à lui-même, seul ou en équipe
  - Travail en équipe

Organisation des tests unitaires

- Le développeur livré à lui-même
  - Hiérarchiser son travail
    - Suivre l’ordre logique de construction de la solution
    - Commencer par tester les bouts de code mutualisés et réutilisés en plusieurs endroits du logiciel (par exemple, l’accès aux données)
    - Après fiabilisation, passer aux couches logicielles appelantes (services, middleware) puis au front (présentation)
    - Peut nécessiter des « bouchons » pour tester certaines parties

Organisation des tests unitaires

- Le développeur livré à lui-même
  - Mesurer le temps passé
    - Temps de développement / Temps de test unitaire
    - Travail itératif : « Je développe, je teste, je corrige, je teste, … »

Organisation des tests unitaires

- Le développeur livré à lui-même
  - Mesurer le temps passé
    - Le test doit rester réfléchi et ne pas devenir un automatisme (prise de recul nécessaire)
    - Ratio développement/TU :
      - Au moins **20%**
      - Mais pas plus de **33%**
      - Par exemple, pour 1 journée de développement, compter 0,5 jour de test unitaire

Organisation des tests unitaires

- Travail en équipe
  - Sensibiliser l'équipe
    - Rôle du Chef de Projet (CP) ou du Scrum Master
    - Le CP ou l'équipe Agile est garant du résultat, du respect du planning et de la charge
    - Le CP ou l'équipe Agile doit connaître les **forces** et **faiblesses** de l'équipe
    - **Communication indispensable**

Organisation des tests unitaires

- Travail en équipe
  - S’assurer que les besoins en test sont couverts
    - Outils et prérequis nécessaires
    - Jeux de données

Organisation des tests unitaires

- Travail en équipe
  - Anticiper les **dépendances** de tâches
    - Ne pas provoquer de **blocages** entre développeurs
    - Exemple :
      - Création et suppression d'une même donnée → le développeur en charge de la suppression devra avoir la donnée initialement créée pour faire son test

Organisation des tests unitaires

- Méthodes de tests unitaires
  - **Les tests croisés**
    - Principe : un développeur A réalise les TU d'une fonction développée par un développeur B
    - Avantages :
      - Apporte un regard extérieur sur la fonction (pas « juge et partie »)
      - Favorise les échanges entre développeurs et donc permet de diffuser la connaissance de l'ensemble de l'applicatif
      - Renforce la cohésion d'équipe

Organisation des tests unitaires

- Méthodes de tests unitaires
  - **Test First**
    - Principe : écrire le test unitaire avant de coder
    - Avantages :
      - **Comprendre** ce qu'on doit développer
      - Ne développer que ce qui est **nécessaire** au passage du test

Organisation des tests unitaires

- Méthodes de tests unitaires
  - **Test Driven Development (TDD)**
    - Principe : évolution du Test First, avec une ligne de code de production par ligne de test en échec
      - **Loi n°1** : Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant
      - **Loi n°2** : Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation
      - **Loi n°3** : Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite
    - Avantages : comme Test First + **concision** et **qualité** du code

Organisation des tests unitaires

- Méthodes de tests unitaires
  - **Extrem Programming (développement en binôme)**
    - Principe : 2 développeurs pour 1 poste de développement
    - Avantages :
      - 4 yeux pendant le développement → réduit le risque d'anomalie
      - Favorise les échanges entre développeurs et donc permet de diffuser la connaissance de l'ensemble de l'applicatif
      - Renforce la cohésion d'équipe

3.

Risques liés à la gestion des TU

Risques liés à la gestion des TU

- Impact sur la charge
  - Définir le seuil de qualité
  - Ne pas négliger les tests à cause des coûts : une anomalie détectée au plus tôt coutera moins cher à corriger
  - **Ne pas faire** de sur-qualité (ne pas oublier le budget à respecter)

Risques liés à la gestion des TU

- Impact sur l'équipe
  - Une bonne stratégie de test conduit à une meilleure **cohésion**
  - **Ne pas démotiver** l'équipe avec une stratégie trop contraignante (trop de documentation à produire, …)
  - Une trop grande exigence en terme de nombre de tests ou de détail peut être perçu comme un manque de confiance

4.

Les tests unitaires automatisés

Limites des TU manuels

- Nécessitent une intervention humaine pour l’exécution
- Non reproductible rapidement
- Nécessite une analyse humaine pour l’interprétation du résultat

Intérêt des TU automatisés

- Pouvoir rejouer régulièrement un ensemble de tests unitaires pour maintenir la qualité pendant la construction
  - Intégration continue
  - Non régression
- Détecter et corriger au plus tôt les anomalies de régression
- Avoir un référentiel des tests unitaires « as code »

Inconvénients/difficultés des TU automatisés

- N’assurent pas le fonctionnement entre les différents composants (on reste au niveau unitaire)
- Doivent être pertinent
- Nécessitent de maintenir à jour les jeux de données
- Ont un coût
  - Mise en œuvre
  - Maintenance

Principe des TU automatisés

- Un test unitaire automatisé se déroule en **quatre** étapes :
  - **setup** : initialiser des objets ou des ressources pour exécuter le test
  - **call** : exécuter le code à tester
  - **verify** : vérifier les données issues des traitements
  - **teardown** : permettre de libérer les ressources ou revenir à l'état initial

Principe des TU automatisés

- Un test unitaire automatisé doit impérativement se faire de façon isolée, donc sans dépendre d'autres tests ni requérir les dépendances utilisées par la fonctionnalité en cours de test

- Un test unitaire automatisé doit obligatoirement être répétable. Cela permet de **capitaliser** les tests unitaires non seulement pour les **tests unitaires** mais aussi pour les tests de **non-régression**

- Idéalement, chaque cas de tests doit être autonome et ne

Principe des TU automatisés

- Quelques **recommandations** :
  - Le nom d’un test unitaire automatisé doit permettre de facilement fournir une indication sur le but du test
  - Le test unitaire doit être vu comme un complément à la documentation, permettant de comprendre le **fonctionnement** de l’application
  - Le nom du test et sa mise en forme doivent être clairs

Principe des TU automatisés

- Quelques **recommandations** :
  - Il est préférable de n'avoir qu’une seule assertion par test car un test ne devrait avoir qu'une seule raison d'échouer
  - Le code d’un test unitaire automatisé doit être maintenu au même titre que le code qu'il teste
    - La même attention doit être portée dans son écriture (respect des normes, commentaires, refactoring, …)

Principe des TU automatisés

- Quelques **recommandations** :
  - Un test unitaire automatisé doit s’exécuter le plus rapidement possible
    - Un nombre important de tests unitaires doit apporter de la qualité au produit mais ne doit pas compromettre par exemple l’intégration continue avec un temps d’exécution trop long

Principe des TU automatisés

- Quelques **recommandations** :
  - Il est nécessaire que les tests unitaires n'aient aucune dépendance
    - Vis-à-vis de systèmes externes (base de données, système de fichier, ...)
    - Vis-à-vis des autres couches logicielles de l'application
  - Utilisation de « bouchon » (mock) pour simuler le comportement des systèmes ou couches applicatives externes à la fonction à tester

5.

Les tests unitaires en Java

TU en Java

- Framework de test unitaire pour Java : **JUnit**
- Mise en place de **JUnit**
  - Site : https://junit.org/junit5/
  - JUnit GA V5.7.0
  - Prérequis : Java 8 minimum

TU en Java

- Architecture JUnit 5
  - 3 modules
    - **JUnit Platform** : API permettant aux outils d'exécuter des tests (IDE ou outils de build)
    - **JUnit Jupiter** : API reposant sur des annotations pour écrire des tests unitaires JUnit 5 et un **TestEngine** pour exécuter les tests
    - **JUnit Vintage** : **TestEngine** pour exécuter des tests JUnit **3** et **4**

TU en Java

- Principe d'écriture de tests avec JUnit
  - Ecriture de classes de tests avec des méthodes annotées (annotations JUnit5)
  - Chaque test est défini dans une méthode comportant l'annotation **@test** dans le bloc de documentation de la méthode
  - Dans une méthode de test, on utilise des assertions pour effectuer les vérifications (par exemple *assertSame()* pour vérifier qu'une valeur constatée correspond à une valeur attendue)

TU en Java

- Principe d’écriture de tests avec JUnit
  - Exemple

```
public class MaClasseDeTest {

        @Test
        void monTest() {
            Assertions.assertTrue(true);
        }
}
```

TU en Java

- Principe d’écriture de tests avec JUnit
  - Il est possible de positionner un **libellé** sur les classes et les méthodes de tests
  - Ce libellé est affiché par les « tests runners »
  - Ce libellé est défini en utilisant l'annotation **@org.junit.jupiter.api.DisplayName**

Annotation

Annotation

```
@DisplayName("Ma classe de test")
public class MaClasseDeTest {

    @Test
    @DisplayName("Mon test")
    void monTest() {
        Assertions.assertTrue(true);
    }
}
```

TU en Java

- Principe d’écriture de tests avec JUnit
  - **Cycle** de vie
    - Préparer l’environnement d’exécution du test avant le test : Before
    - Remettre l’environnement dans son état initial après le test : After

TU en Java

- Principe d'écriture de tests avec JUnit
  - **Cycle** de vie
    - Annotation ***@BeforeEach*** pour configurer l'environnement de test avant l'exécution d'un test
      - Exécutée **avant** chaque méthode de test
    - Annotation ***@AfterEach*** pour remettre à l'état initial l'environnement de test
      - Exécutée **après** chaque exécution d'une méthode de test (en succès ou non)
    - Intérêt : ne pas dégrader les données de tests après chaque test

TU en Java

- Principe d'écriture de tests avec JUnit
  - **Cycle** de vie
    - Annotation ***@BeforeAll*** pour configurer l'environnement de test avant l'exécution du premier test
      - Exécutée **une seule fois** avant le premier test
    - Annotation ***@AfterAll*** pour remettre à l'état initial l'environnement de test
      - Exécutée **une seule fois** après l'exécution du dernier test
    - Exemple : initialiser un objet une seule fois avant tous les tests

TU en Java

- Exercice

Les méthodes annotées par **@BeforeAll** et **@AfterAll** doivent être statiques !

```
public class MaClasseDeTest {

  @BeforeAll
  static void setupAll() {
    System.out.println("setupAll");
  }

  @BeforeEach
  void setup() {
    System.out.println("setup");
  }

  @AfterEach
  void teardown() {
    System.out.println("teardown");
  }

  @AfterAll
  static void teardownAll() {
    System.out.println("teardownAll");
  }

  ...
```

Les méthodes annotées par **@BeforeAll** et **@AfterAll** doivent être statiques !

TU en Java

- Exercice

```
public class MaClasseDeTest {

 ...

  @Test
  void unTest() {
    System.out.println("unTest");
    Assertions.assertTrue(true);
  }

  @Test
  void unAutreTest() {
    System.out.println("unAutreTest");
    Assertions.assertTrue(true);
  }

}
```

TU en Java

- Exercice

Résultat attendu dans la console :

```
setupAll
setup
unTest
teardown
setup
unAutreTest
teardown
teardownAll
```

TU en Java

- Principe d’écriture de tests avec JUnit
  - **Les assertions**
    - Une assertion a pour but de contrôler un résultat
    - Si le contrôle échoue, alors l'assertion lève une exception qui fait échouer le test
    - Exemples d’assertions :
      - assertEquals() / assertNotEquals() → tester l’égalité
      - assertTrue() / assertFalse() → tester une valeur booléenne
      - assertNull() / assertNotNull()

TU en Java

- Principe d’écriture de tests avec JUnit
  - **Gestion des exceptions (tester si une exception est levée)**
    - Une assertion permet de vérifier que l'exécution d’une méthode passée en paramètre lève l'exception précisée : **assertThrows()**

assertThrows(<Exception>.class, () -> {<méthode appelée>;});

    - Si l’exception attendue n’est pas levée lors de l’exécution du test, alors l’assertion lève une exception pour faire échouer le test

TU en Java

- Exercice

```
 @Test
  void verifierException() {
    String valeur = null;
    Assertions.assertThrows(NumberFormatException.class, () -> {
Integer.valueOf(valeur); });
  }
```

Test OK !

TU en Java

- Exercice

```
 @Test
  void verifierException() {
    String valeur = "1";
    Assertions.assertThrows(NumberFormatException.class, () -> {
Integer.valueOf(valeur); });
  }
```

Ce test doit échouer car aucune exception de type **NumberFormatException** n'est levée par l'exécution de la méthode Integer.valueOf(1)

TU en Java

- Principe d’écriture de tests avec JUnit
  - **Désactivation de tests**
    - Pour ne pas que le test soit exécuté (pas terminé, en attente de codage, …)
    - Annotation **@Disabled**
    - Pour désactiver toutes les méthodes d’une classe : annotation **@Disabled** au niveau de la classe

TU en Java

- Exercice

```
@Test
@Disabled("A ne pas executer")
void monTest() {
  fail("Non implémenté");
}
```

Ne sera pas exécutée

TU en Java

- Principe d'écriture de tests avec JUnit
  - **Répétition(s) d'un test**
    - Possibilité de répéter un même test plusieurs fois
    - Annotation **@RepeatedTest(<nombre de répétition>)** sur la méthode de test
    - Les méthodes **@BeforeEach** et **@AfterEach** seront exécutées respectivement avant et après chaque exécution du test répété
    - Possibilité de positionner un nom également
      - Paramètre « value = X » pour le nombre de répétitions
      - Paramètre « name = Y » pour le nom à positionner

TU en Java

- Exercice

```
@DisplayName("test repété")
@RepeatedTest(3)
void testRepete() {
        System.out.println("testRepete");
        Assertions.assertEquals(2, 1 + 1, "1 + 1 doit être égal à 2!");
}
```

3 exécutions du test

TU en Java

- Exercice

```
@DisplayName("test repété 2")
@RepeatedTest(value = 3, name = RepeatedTest.LONG_DISPLAY_NAME)
void testRepete2() {
        System.out.println("testRepete2");
        Assertions.assertEquals(2, 1 + 1, "1 + 1 doit être égal à 2!");
}
```

3 exécutions du test avec le nom long de la méthode affichée

TU en Java

- Principe d’écriture de tests avec JUnit
  - **Test avec paramètre(s)**
    - Possibilité d’exécuter un test plusieurs fois avec des paramètres différents
    - Annotation ***@ParameterizedTest*** sur la méthode de test
    - + source de données !

TU en Java

- Principe d’écriture de tests avec JUnit
  - **Test avec paramètre(s)**
    - Sources de données
      - @ValueSource → tableau de valeurs
      - @EnumSource → énumération
      - @MethodSource → méthode
      - @CsvSource → valeurs au format csv
      - @CsvSourceFile → valeurs dans un fichier csv
      - …

TU en Java

- Exercice

```
@ParameterizedTest
@ValueSource(ints = { 1, 2, 3 })
void testParametreAvecValueSource(int valeur) {
   Assertions.assertEquals(valeur + valeur, valeur * 2);
}
```

1 exécution du test pour chaque valeur positionnée dans l'annotation **@ValueSource**

TU en Java

- Exercice

```
@ParameterizedTest
@CsvSource({ "1, 1", "1, 2", "2, 3" })
void testAdditioner(int a, int b) {
  int attendu = a + b;
  Assertions.assertEquals(attendu, a + b);
}
```

1 exécution du test pour chaque « ligne » de l'entrée csv positionnée dans l'annotation **@CsvSource**

TU en Java

- Principe d'écriture de tests avec JUnit
  - **Ordre des tests**
    - 1ère chose : annoter la classe avec **@TestMethodOrder(OrderAnnotation.class)**
    - Ensuite, on peut spécifier l'ordre d'exécution des méthodes de test avec l'annotation **@Order**

```
@Order(1)
void monTest() …

@Order(2)
Void monAutreTest() …
```

5.

Les tests unitaires avec bouchon

TU avec bouchon

- Un test unitaire ne doit pas dépendre d'un système externe ou d'une autre couche logicielle (dépendances) !
- Objectif = **Simuler** un composant qui n'est pas disponible dans l'environnement de test unitaire automatisé
- Plusieurs frameworks disponibles : par exemple **Mockito**

TU avec bouchon

- Objectifs de **Mockito** = Spécifier le comportement que doit avoir l'objet « **Mock** » (simulé) avec :
  - Les méthodes invoquées → paramètres d'appel et valeurs de retour
  - L'ordre d'invocation de ces méthodes
  - Le nombre d'invocations de ces méthodes

TU avec bouchon

- **Mockito** permet de créer dynamiquement des objets à partir d'interfaces ou de classes et de tester différents cas de figure :
  - La simulation de cas d'erreurs en levant des exceptions
  - La validation des appels de méthodes
  - La validation de l'ordre de ces appels
  - La validation des appels avec un timeout

TU avec bouchon

- Utilisation de **Mockito** :
  - Annotation ***@ExtendWith(MockitoExtension.class)*** sur la classe de test
  - Création d'objets « mockés » avec l'annotation **@Mock**

TU avec bouchon

- Utilisation de **Mockito** :
  - Définition du comportement des objets « mockés » :
    - ***Mockito.when(<méthode>).thenReturn(<résultat attendu>);***
    - Exemple :
      Mockito.when(service.getMessage()).thenReturn(‘’Mon message’’);

TU avec bouchon

- Utilisation de **Mockito** :
  - Définition du comportement des objets « mockés » :
    - Simuler le retour d'une exception :
      Mockito.when(<méthode>).thenThrow(<exception à lever>);
    - Exemple :
      Mockito.when(service.getMessage()).thenThrow(new Exception());

TU avec bouchon

- Utilisation de **Mockito** :
  - Vérification d'interactions :
    - Vérifier que les interactions avec le mock sont bien celles **attendues**
      - Méthode ***Mockito.verify(<objet mocké>).<méthode>;***
    - Vérifier que le nombre d'interactions est **correct**
      - Méthode ***Mockito.verify(<objet mocké>, times(<nombre d'interactions>).<méthode>;***
    - Exemple : Mockito.verify(**service**, times(1)).getMessage();

6.

Couverture du code par les tests unitaires

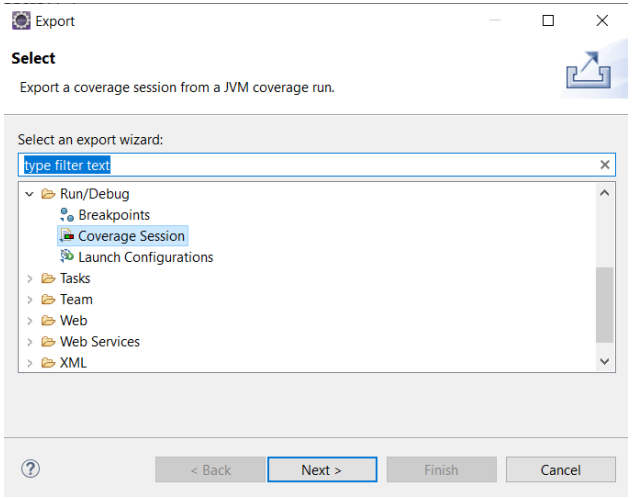
Couverture du code par les TU

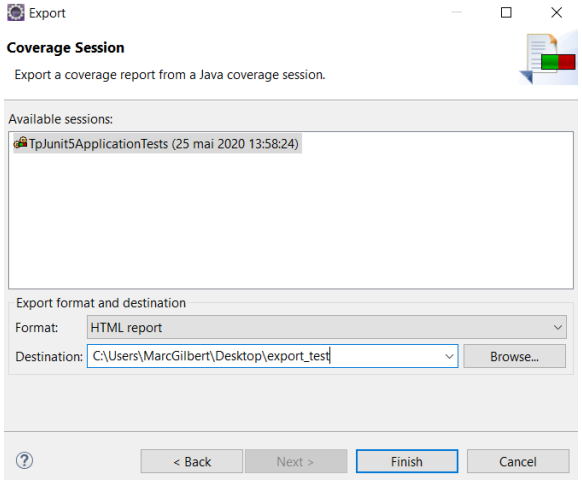
- **Analyse** de la couverture de code
  - Taux de code source couvert par les tests
    - Composant **EclEmma** intégré dans Eclipse (basé sur JaCoCo)
    - Composant **Emma** pour intelliJ (ou composant intégré)

Element	Coverage	Covered Lines	Missed Lines	Total Lines
commons-collections	80.7 %	11092	2646	13738
src	80.7 %	11092	2646	13738
org.apache.commons.collections	77.1 %	3991	1188	5179
org.apache.commons.collections.bag	66.9 %	234	116	350
org.apache.commons.collections.bidimap	91.2 %	964	93	1057
AbstractBidiMapDecorator.java	85.7 %	6	1	7
AbstractBidiMapDecorator	85.7 %	6	1	7
AbstractBidiMapDecorator(BidiMap)	100.0 %	2	0	2
getBidiMap()	100.0 %	1	0	1
getKey(Object)	100.0 %	1	0	1
inverseBidiMap()	0.0 %	0	1	1

Couverture du code par les TU

- **Analyse** de la couverture de code
  - Permet de générer un rapport HTML de couverture de code





7.

Documentation

Documentation JUnit

- **Biblio**
  - https://junit.org/junit5/

MODULE JAVA

Java - Java Server Faces

Marc GILBERT
marc.gilbert@wijin.tecn

1.

Frameworks MVC

Frameworks MVC

- **Objectif :** simplifier la mise en oeuvre du modèle MVC sur son application

- Rappel MVC :

  - Modèle de conception pour organiser le code source d'une application Web

  - Découpage clair des responsabilités
    - Découpage simplifiée du développement des fonctionnalités
    - Meilleure maintenabilité

Frameworks MVC

- **Framework MVC =** surcouche aux Servlets
- Inversion de contrôle :
  - Le framework prend la main sur le contrôle de l’application

Frameworks MVC

- **Avantages**
  - **Simplification** du flot d’exécution
    - Moins de code
    - Prise en compte native de tâches génériques
    - Organisation du code à respecter

Frameworks MVC

- **Différents type de frameworks MVC**
  - Basé sur les **requêtes**
  - Basé sur les **composants**

Frameworks MVC

- **Différents type de frameworks MVC**
  - **Basé sur les requêtes**
    - Le plus simple à comprendre
    - Agit sur le cycle de vie d’une requête (actions)

Frameworks MVC

- **Différents type de frameworks MVC**
  - **Basé sur les requêtes**
    - Prend en entrée une requête et retourne une réponse
      - Flot d'exécution linéaire
      - Orienté "actions"
    - Utilisation d'une **Servlet unique** pour la redirection
      - Délègue les actions et traitements à la couche "Modèle"

Frameworks MVC

- **Différents type de frameworks MVC**
  - **Basé sur les requêtes**
    - Exemples de frameworks :
      - Struts
      - Stripes

Frameworks MVC

- **Différents type de frameworks MVC**
  - **Basé sur les composants**
    - Découpage du code en composants
    - Masquage des chemins des requêtes

Frameworks MVC

- **Différents type de frameworks MVC**
  - **Basé sur les composants**
    - Utilisation d’une **Servlet unique**
    - Développement des actions métiers uniquement
    - Création des vues avec un langage spécifique

Frameworks MVC

- **Différents type de frameworks MVC**
  - **Basé sur les composants**
    - Exemples de frameworks :
      - JSF (Java Server Faces) → adopté par le standard JEE
      - Wicket

Frameworks MVC

- **Différents type de frameworks MVC**
  - **Comparatif**
    - Basé sur les **requêtes**
      - Plus de code à produire
      - Plus de contrôle sur le cycle de vie des requêtes
    - Basé sur les **composants**
      - Moins de code à produire
      - Moins de possibilité de contrôle sur le cycle de vie des requêtes et le rendu

2.

Java Server Faces

Framework JSF

- **Framework MVC basé sur les composants**
  - **Spécification** construite sur l'**API Servlet**
  - Fournit des **composants** sous forme de bibliothèques de balises
    - Utilisables dans les fichiers **JSP** ou d'autres technologies de restitution (par exemple : **Facelet → utilisée par défaut**)
  - Permet de créer des **templates**

Framework JSF

- **Template ?**
  - **Découpage** des pages/vues en plusieurs composants
  - **Assemblage** de ces composants pour obtenir une vue finale

Framework JSF

- **Principes de JSF**
  - **Contrôleur unique** (Servlet)
    - Gère le cycle de vie des requêtes
    - Nommée la “FacesServlet”
    - → Pattern “Front Controller”
  - **Fonctionnalités natives**
    - Récupération des paramètres des requêtes
    - Conversions
    - Validations
    - Génération des réponses
    - …

Framework JSF

- **Principes de JSF**
  - **On a besoin de :**
    - Une **vue** : page JSP ou page “Facelet”
    - Un **modèle** : bean Java
  - **JSF** s’occupe de **lier** la vue avec le modèle
  - La **FacesServlet** gère **le traitement des requêtes** qui arrivent sur l’application
    - Plus besoin de coder les Servlets !!!

Framework JSF

- **Organisation du code sur une application avec JSF**
  - **Vues :**
    - Pages JSP (utilisation **dépréciée**) ou pages “**Facelet**” (XHTML)
  - **Modèles :**
    - Beans **Java**
  - **Contrôleurs :**
    - Servlet unique “**FacesServlet**”
    - Bean Java “**managed-bean**”

Framework JSF

- **Organisation du code sur une application avec JSF**
  - **Configuration**
    - Fichier “**web.xml**”
      - Déclaration de la **Servlet unique** (FacesServlet)

```
<servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

Framework JSF

- **Organisation du code sur une application avec JSF**
  - **Configuration**
    - Fichier “**web.xml**”
      - Mapping des requêtes vers la Servlet unique

```
<servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
```

→ toutes les requêtes concernant des fichiers **xhtml**

3.

Facelets

Facelets

- **Facelet : techno de référence pour les vues avec JSF**
  - **Pages XHTML**
    - Format **XML**
    - Extensions possibles :
      - .jsf
      - **.xhtml**
      - .faces

Facelets

- **Facelet : techno de référence pour les vues avec JSF**
  - **Pages XHTML**
    - **Structure**

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
    ...
</html>
```

Facelets

- **Facelet : techno de référence pour les vues avec JSF**
  - **Pages XHTML**
    - **Utilisation de bibliothèques et de balises**
      - Inclusion via **namespace** (attribut xmlns)

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
                 xmlns:f="http://java.sun.com/jsf/core"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:ui="http://java.sun.com/jsf/facelets">
   ...
</html>
```

Facelets

- **Facelet : techno de référence pour les vues avec JSF**
  - **Pages XHTML**
    - **Exemple de balise pour afficher un libellé** (bibliothèque html de JSF)

```
<h:outputLabel for="lastname">Nom : <span
class="requis">*</span>
</h:outputLabel>
```

Facelets

- Facelet : techno de référence pour les vues avec JSF
  - Utilisation possible des “EL”
    - Utilisation du “#”

```
#{customer.lastname}
```

4.

Processus

Processus

- **Différentes étapes pour le traitement d'une requête**
  - **1. Restauration de la vue**
    - La requête entrante est dirigée vers la Servlet unique (FacesServlet)
    - La **FacesServlet** détermine le nom de la page demandée et vérifie si une vue existe pour cette page
    - La **FaceServlet** cherche les composants utilisés par cette vue
      - Création ou réutilisation si un composant est déjà créé
    - La **FaceServlet** sauvegarde la vue dans un objet de type **FacesContext**

Processus

- **Différentes étapes pour le traitement d'une requête**
  - **2. Application des valeurs contenues dans la requête**
    - Les paramètres de la requête sont **récupérés** et **convertis** par rapport au type
      - Si erreur lors de la conversion, une erreur est stockée dans l'objet de type **FacesContext**

Processus

- **Différentes étapes pour le traitement d'une requête**
  - **3. Validation des données**
    - Exécution des règles de validation mises en oeuvre par les développeurs
      - Si la validation d'un champ échoue, une erreur est stockée dans l'objet de type **FacesContext**

Processus

- **Différentes étapes pour le traitement d'une requête**
  - **4. Mise à jour des valeurs**
    - Positionnement des valeurs dans les beans
      - Valeurs valides au sens **format**

Processus

- **Différentes étapes pour le traitement d'une requête**
  - **5. Appel aux actions**
    - Appel par JSF au code métier (partie "Modèle" du MVC)
      - Classes "beans" (managed beans)
      - Utilisation des données par l'application

Processus

- **Différentes étapes pour le traitement d'une requête**
  - **6. Envoi de la réponse**
    - Affichage de la vue à l'utilisateur
      - Génération de la page HTML

5.

Managed Beans

Managed Bean

- **Bean Java géré par JSF**
  - **Pour faire le lien entre l'IHM et le code métier de l'application**
    - Contient des accesseurs (lien avec les données de la page)
    - Contient des méthodes de validation (pour les formulaires)
    - Contient des méthodes d'action (lien avec le code métier de l'application)

6.

Utilisation d’une implémentation

Implémentation JSF

- **Différentes implémentations existantes**
  - **Eclipse Mojarra**
    - https://projects.eclipse.org/projects/ee4j.mojarra
  - **Apache MyFaces**
    - http://myfaces.apache.org/#/

Implémentation JSF

- **Utilisation de Apache MyFaces**
  - **Import Maven**
    - Implémentation

```
<dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-impl</artifactId>
    <version>2.3.7</version>
</dependency>
```

Implémentation JSF

- **Utilisation de Apache MyFaces**
  - **Import Maven**
    - Utilisation de CDI pour l’injection de dépendances (beans)

```
<dependency>
        <groupId>javax.enterprise</groupId>
        <artifactId>cdi-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>
```

Implémentation JSF

- **Utilisation de Apache MyFaces**
  - **Import Maven**
    - Utilitaire JAXB-API (transformation Java <-> XML)

```
<dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.3.1</version>
    </dependency>
```

7.

Projet fil rouge CRM

Mise en place de JSF

Projet Fil Rouge

- Mise en place du projet
  - pom.xml
- Mise en place de la gestion des clients
  - Formulaire de création d’un nouveau client
  - Page d’affichage de la liste des clients
- Mise en place de la gestion des commandes
  - Formulaire de création d’une nouvelle commande
  - Page d’affichage de la liste des commandes

Projet Fil Rouge

- Mise en place de la gestion des clients
  - Formulaire de création d’un nouveau client
    - Fichier createCustomer.xhtml
    - Bean CustomerBean.java
      - Méthode de création d’un client
      - Méthode de validation personnalisée pour le téléphone
  - Page d’affichage de la liste des clients
    - Fichier listCustomers.xhtml
    - Bean CustomerBean.java
      - Méthode de récupération des clients
      - Méthode de suppression d’un client

Projet Fil Rouge

- Mise en place de la gestion des commandes
  - Formulaire de création d’une nouvelle commande
    - Fichier createOrder.xhtml
    - Bean OrderBean.java
      - Méthode de création d’une commande
      - Méthode(s) de validation personnalisée(s) ?
  - Page d’affichage de la liste des commandes
    - Fichier listOrders.xhtml
    - Bean OrderBean.java
      - Méthode de récupération des commandes
      - Méthode de suppression d’une commande

MODULE JPA

JPA avec Hibernate

Marc GILBERT
marc.gilbert@wijin.tech

1.

Introduction

Introduction JPA

- **JPA :** Java Persistence API
  - Framework de **persistance** en Java
    - ORM : Object-Relational Mapping (exemple : Hibernate)
  - **Principes** de base
    - Définition de la correspondance entre le structure des classes Java et le schéma relationnel de la Base de données
    - Manipulation directe des objets dans le code Java

Introduction JPA

- **JPA :** Java Persistence API
  - Le Framework s’occupe de la **transformation**
    - Plus besoin de requêtes SQL !
    - Langage de requêtage propre mis à disposition

2.

Problématique de la persistance

Problématique de la persistance

- **Limites** de JDBC
  - Nécessite l'utilisation de **requêtes SQL**
  - Représentation **différente** des données
    - Langage SQL pour le requêtage
    - Classes Java pour les entités

Problématique de la persistance

- **Limites** de JDBC
  - Ça marche, **mais…**
    - Beaucoup de code à produire
    - Si un grand nombre de références entre les classes, nécessité de charger beaucoup de choses (problématique des **ressources**)
    - Problème de **cohérence** entre les objets et la Base de données
      - C’est au développeur de gérer la cohérence entre le contenu des objets et la Base de données

Problématique de la persistance

- **De JDBC à JPA**
  - Avec JPA, on définit des **correspondances** entre des classes (POJO : Plain Old Java Object) et des tables
  - JPA gère la **cohérence** entre les objets et les données en Base de données
  - Beaucoup **moins de code** technique à produire !

Problématique de la persistance

- **Fonctionnement de JPA**
  - Manipulation **d'objets** métier Java **uniquement**
  - API proposant des fonctionnalités :
    - Récupération d'objets à partir des données de la Base
      - Langage propre : JPQL
    - Persistance des objets en base
      - Insertion, modification, suppression
      - Avec gestion des transactions

Problématique de la persistance

- **Fonctionnement de JPA**
  - Nécessite une **implémentation**
    - Hibernate, TopLink, EclipseLink, ...
    - **Attention** : éviter d'utiliser les fonctionnalités spécifiques d'une implémentation particulière pour ne pas être dépendant

3.

Modèle de persistance

Modèle de persistance

- **Mapping sur les entités Java**
  - Une **entité** = un objet **métier**
  - Une classe Java est **mappée** sur une table SQL
    - **Correspondance** entre les attributs de la classe et les colonnes de la table
    - Mapping réalisé avec des **annotations** dans la classe

Modèle de persistance

- **Mapping sur les entités Java**
  - Au minimum :
    - **@Entity** : défini comme entité
    - **@Id** : pour l'identifiant de l'entité
    - Un constructeur (pour instancier)
    - Un getter et un setter pour chaque attribu

```java
@Entity
public class Person {

    @Id
    private Long id;            // ID unique en base
    private String firstname;   // Prénom

    /* CTOR */

    public Person() {
            // Default constructor
    }

    /* GETTERS */

    public Long getId() {
            return this.id;
    }

    public String getFirstname() {
            return this.firstname;
    }

    /* SETTERS */

    public void setId(Long id) {
            this.id = id;
    }

    public void setFirstname(String firstname) {
            this.firstname = firstname;
    }
}
```

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - Les annotations **@Entity** et **@Id** **ne suffisent pas** à faire comprendre à l'ORM comment manipuler cette entité
  - Il faut ajouter **@Table** pour décrire plus spécifiquement les détails à propos de la base de données (ex: nom, schéma)

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - **@GeneratedValue** est utilisé conjointement avec @Id pour les valeurs de clé générée :
    - **IDENTITY** : pour spécifier une colonne d'identité de la base de données
    - **AUTO** : pour choisir automatiquement une implémentation basée sur la base de données utilisée
    - **SEQUENCE** : pour utiliser une séquence (si la base de données la supporte)(Voir @SequenceGenerator)
    - **TABLE** : pour spécifier qu'une base de données utilisera une table et une colonne d'identité pour s'assurer son caractère unique (voir @TableGenerator)

Modèle de persistance

- Concepts **ORM** : Notion d'**entité**
  - **Exemple :**

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstname;
```

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - En l'absence d'information précise, Hibernate va assumer certaines configurations par défaut
    - L'utilisation de **@Column** permet de définir les noms de colonnes souhaités, ainsi que des informations à leur propos

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - L'utilisation de **@Column** permet de définir les noms de colonnes souhaités, ainsi que des informations à leur propos :
      - columnDefinition
      - insertable
      - length
      - name
      - nullable
      - precision
      - scale
      - table
      - unique
      - updatable

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
  - Sans les informations de l'annotation, Hibernate a considéré l'attribut « firstname » comme :
    - Étant mappé à une colonne nommée "firstname" (et "first_name")
    - Pouvant avoir une valeur à NULL
    - Ayant une longueur à 255 caractères

```
@Entity
@Table(name = "person")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(
        name = "first_name",
        nullable = false,
        length = 100
    )
    private String firstname;
```

Modèle de persistance

- Concepts **ORM** : mapping sur les **entités**
  - **Annotation de colonnes**
    - En dehors de « simplement décrire » la base de données dans le code, Hibernate possède aussi des annotations permettant d'écrire du code plus simple :
    - Exemple : l'utilisation de **@Enumerated** permet de dire à Hibernate de « mapper » des chaînes de caractères, des nombres en base vers des « **Enums** » Java (plus pratique)
    - Exemple : l'utilisation de **@Temporal** permet de dire à Hibernate de « mapper » des dates SQL (TIMESTAMP, DATETIME, etc.) vers des « **Date** »

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Il existe 4 types de relation entre entités :
    - Relation « one to one » : **relation 1 -> 1**
    - Relation « one to many » : **relation 1 -> n**
    - Relation « many to one » : **relation n -> 1**
    - Relation « many to many » : **relation n -> n**

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - S'ajoutent à ces types de relation « des **configurations** »
    - Notion de **sens** :
      - Relation **unidirectionnelle**
      - Relation **bidirectionnelle**
    - Notion de **cascade** : que faire de B, lié à A,
      - Lors d'une **mise à jour** de A ?
      - Lors d'une **suppression** de A ?

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **1 -> 1** : annotation **@OneToOne**
    - Pour définir une relation **forte**, **bidirectionnelle** entre 2 entités
    - Exemple : « Un navire est gouverné par un capitaine »

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **1 -> N** : annotation **@OneToMany**
    - Pour définir une relation entre 1 entité et une liste d'entités
    - Exemple : « Une personne possède un ou plusieurs téléphones »

```
// Person.java

@OneToMany(mappedBy = "person")
private List<Phone> phones;
```

Modèle de persistance

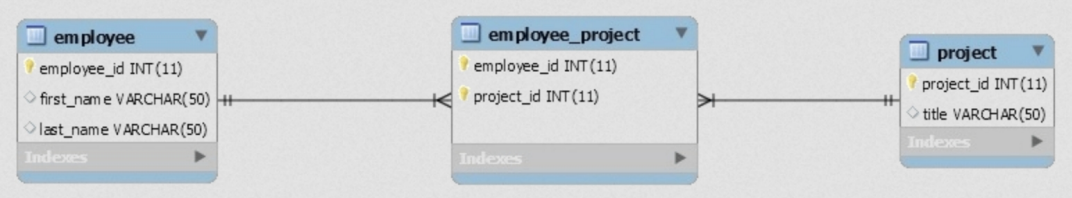
- Concepts **ORM** : cardinalité des **entités**
  - Relation **N -> 1** : annotation **@ManyToOne**
    - Pour définir une relation **contraire** à @OneToMany (plusieurs entités liées à une autre)
    - Exemple : « Un ou plusieurs téléphones peuvent être détenus par une personne

```
// Phone.java

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "person_id")
private Person person;
```

Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **N -> N** : annotation **@ManyToMany**
    - Pour définir des relations entres entités dont on ne peut faire les liens qu'à travers des **tables de jointure**
    - Exemple : « Un ou plusieurs employés travaillent sur un ou plusieurs



Modèle de persistance

- Concepts **ORM** : cardinalité des **entités**
  - Relation **N -> N** : annotation **@ManyToMany**

```
// Employee.java

@ManyToMany(cascade = { CascadeType.ALL })
@JoinTable(
    name = "Employee_Project",
    joinColumns        = { @JoinColumn(name = "employee_id") },
    inverseJoinColumns = { @JoinColumn(name = "project_id") }
)
List<Project> projects;
```

```
// Project.java

@ManyToMany(mappedBy = "projects")
private List<Employee> employees;
```

Modèle de persistance

```
// Phone.java

@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "person_id")
private Person person;
```

- Concepts **ORM** : mode de récupération d'**entités** liées entre elles
  - Il existe **2 modes** de récupération d'entités
    - **LAZY** : interroge la base de données seulement quand la propriété est appelée (exemple: appel à un getter)
    - **EAGER** : interroge la base de données dès que l'objet original est créé

Modèle de persistance

- Concepts **ORM** : types d'effet « **cascade** » entre entités
  - Les relations entre entités **dépendent** souvent de l'existence d'une autre
  - Exemple : relation **Personne** <-> **Adresse**
    - Sans la personne, **l'adresse n'aurait pas de signification métier**
    - En **supprimant** la personne, on souhaiterait que son adresse soit **supprimée** aussi

Modèle de persistance

- Concepts **ORM** : types d'effet « **cascade** » entre entités

```java
@Entity
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String name;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Address> addresses;
}
```

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private int zipCode;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;
}
```

Modèle de persistance

- Bien réfléchir à la **cardinalité** des entités
- Ne pas partir tête baissée pour **écrire les entités**
- Attention à positionner le “fetch type” en LAZY sur les collections d’objets qui ne sont **pas nécessaires** au premier abord
  - Syndrome de « ramener la terre entière »
- Penser aux effets « **cascade** » entre entités
  - Souhaite-t-on garder une carte d’identité en base si l’on a

4.

Manipulation des entités

Manipulation des entités

- Configuration d’une **unité de persistance**
  - **Fichier XML** définissant la liste des classes correspondant à des entités, la connexion à la base de données, des paramètres, …
  - Dans l'application **Java**
    - Récupération d’un **entity manager** pour manipuler les entités
    - Les modifications se font via une **transaction**

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_2.xsd">
                <persistence-unit name="crm" transaction-type="JTA">
                                <class>fr.m2i.crm.model.Customer</class>
                                <class>fr.m2i.crm.model.Order</class>
                                <properties>
                        <!-- database connection -->
                        <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
                        <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/crm" />
                        <property name="javax.persistence.jdbc.user" value="crm" />
                        <property name="javax.persistence.jdbc.password" value="crm" />
                  </properties>
                </persistence-unit>
                </persistence>
```

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <persistence-unit ...>
      - Définit le nom de l'unité de persistance et le type de transaction utilisée

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <class>
      - Définit qu'une classe Java sera une entité dont les instances seront persistantes en base de données

Manipulation des entités

- **Unité de persistance**
  - Fichier **persistence.xml** dans répertoire META-INF (par exemple sous src/main/resources)
    - <properties>
      - Ensemble de propriétés de configuration
      - Par exemple, les paramètres de connexion à la base (URL, driver, utilisateur et mot de passe)

Manipulation des entités

- **Entity Manager**
  - Récupéré à partir de la fabrique de gestionnaire d'entité et via le nom donné à l'unité de persistance

```
private EntityManager em = null;
...
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("monPu");
em = emf.createEntityManager();
...
```

Manipulation des entités

- **Etat d'un objet persistant**
  - Plusieurs états pour l'instance d'une classe entité
    - **Persistant** : l'entité a une correspondance de contenu en BDD
      - **Gérée** : état synchronisé par le gestionnaire d'entité avec le contenu en BDD
      - **Détachée** : état non géré, les modifications ne sont plus synchronisées avec la BDD
    - **Transient** : objet java classique avec existence uniquement en mémoire de la JVM

Manipulation des entités

- **Etat d'un objet persistant**
  - Plusieurs états pour l'instance d'une classe entité
    - **Supprimé** : instance persistante dont on a supprimé le contenu associé en BDD
      - L'objet existe toujours en mémoire de la JVM mais n'a plus de correspondance en base

Manipulation des entités

- **Opérations sur les instances d’entité**
  - Récupération d’une instance d'une entité en précisant sa classe et son identifiant
    - <T> T find(Class<T> entityClass, Object id)
      - Customer customer = em.find(Customer.class, 3) ;

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Modification du contenu de la BDD en mode transactionnel

```
EntityTransaction trans = null;
try {
        trans = em.getTransaction();
                trans.begin();
                        … ici les actions …
                trans.commit();
        } catch(Exception e) {
        if (trans != null) trans.rollback();
        }
```

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Rendre persistant en BDD un objet qui devient géré par le gestionnaire d'entités
    - void persist(Object entity)

```
trans.begin();
Customer customer = new Customer(...);
em.persist(customer);
trans.commit();
```

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Récupérer une copie gérée par le gestionnaire d'entité de l'objet passé en paramètre
    - <T> T merge(T entity)

```
Customer newCustomer = em.merge(customer);
// Modifications effectuées sur l'objet retourné par le merge, pas l'initial
newCustomer.setLastname("Dupont");
trans.commit();
```

Manipulation des entités

- **Opérations sur les instances d’entité**
  - Détacher un objet du gestionnaire d'entité (les modifications sur l’objet ne sont alors plus reportées sur la BDD)
    - void detach(Object entity)

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Détacher tous les objets du gestionnaire d'entité
    - void clear()

Manipulation des entités

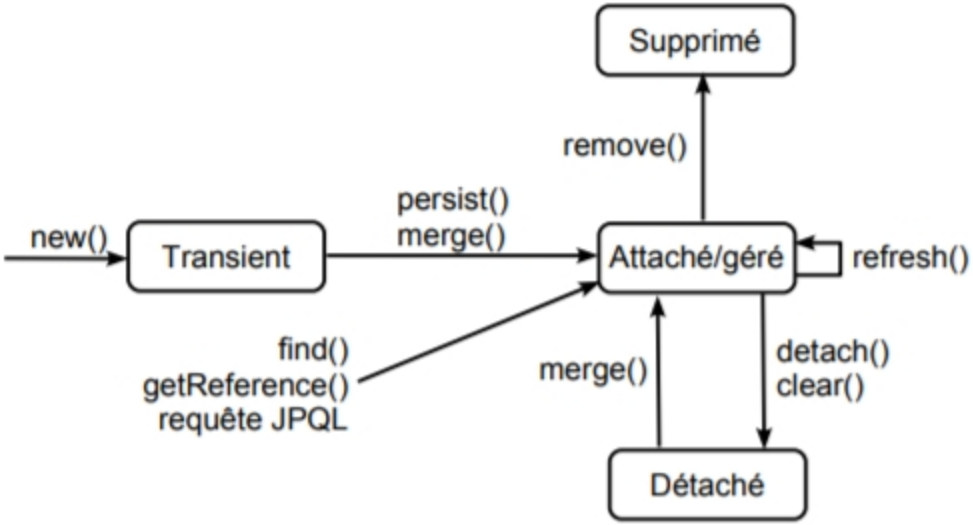
- **Opérations sur les instances d'entité**
  - Supprimer un objet : effacer ses données en base
    - void remove(Object entity)

Manipulation des entités

- **Opérations sur les instances d'entité**
  - Remettre à jour le contenu de l'objet par rapport au contenu en base
    - void refresh(Object entity)

Manipulation des entités

- **Cycle de vie d'un objet persistant**



Manipulation des entités

- **JPQL et Query**
  - **JPQL** n'est pas du SQL
  - Langage **d'interrogation** centré sur les **objets Java**

```
private EntityManager em;
// ...

public List<Person> getAll() {
    Query query = em.createQuery("SELECT p FROM Person p");
    List<Person> results = query.getResultList();
}
```

Manipulation des entités

- **JPQL et Query**
  - **Query** avec paramètres
    - Liaison par **nom de paramètre** (“name parameter binding”)

Requête → SELECT p FROM Person WHERE p.name = :searched

query.setParameter(“searched”, “Harry”);

Manipulation des entités

- **JPQL et Query**
  - **Query** avec paramètres
    - Liaison par **position de paramètre** (“positionnal parameter binding”)

Requête → SELECT p FROM Person WHERE p.name = ?1

query.setParameter(1, “Harry”);

Manipulation des entités

- JPQL et Query
  - Native Query
    - Pour **reprendre** la main sur le **SQL**

```
List<Object[]> persons = session
    .createNativeQuery("SELECT id, name FROM PERSON" )
    .list();

for(Object[] person : persons) {
        Number id = (Number) person[0];
        String name = (String) person[1];
}
```

```
List<Phone> phones = session.createNativeQuery(
        "SELECT id, phone_number, phone_type, person_id " +
        "FROM Phone" )
    .addEntity( Phone.class )
    .list();
```

MODULE WS

Présentation des services Web

1.

Présentation des architectures distribuées

Architectures ?

- Architecture centralisée : « client/serveur »
  - 1 serveur / N clients
- Architecture distribuée : « peer to peer »
  - N serveurs / N clients

Rappel sur l'architecture centralisée

- Client/serveur : distinction stricte entre le rôle de client et le rôle de serveur
  - 1 client effectue une requête pour un service donné sur un serveur et attend une réponse
  - 1 serveur reçoit une demande de service, la traite et retourne une réponse au client

Rappel sur l'architecture centralisée

- Caractéristiques du client :
  - Actif
  - Connecté à un serveur
  - Envoie des requêtes à un serveur
  - Attend et traite les réponses du serveur
  - Interagit avec un utilisateur final (par exemple avec une IHM)

Rappel sur l’architecture centralisée

- Caractéristiques du serveur :
  - Passif
  - A l’écoute des requêtes clients
  - Traite les requêtes et fournit une réponse
  - Pas d’interaction directe avec les utilisateurs finaux

Rappel sur l’architecture centralisée

- Exemples d’architecture centralisée client/serveur :
  - Consultation de pages web (envoi de requêtes HTTP depuis un navigateur à un serveur pour consulter les pages)
  - Gestion des mails (client pour envoyer et recevoir les mail, serveur pour la gestion : SMTP, POP, IMAP)

Rappel sur l'architecture centralisée

- Découpage en couches :
  - **Présentation** : affichage, dialogue avec un utilisateur final
  - **Service** : traitements, règles de gestion et logique applicative
  - **Données** : DAO, persistance des données

Rappel sur l'architecture centralisée

- Découpage en couches :
  - La **répartition** de ces couches entre les rôles de client et de serveur permet de distinguer entre les différents types d'architecture client/serveur
    - 2 tiers
    - 3 tiers
    - N tiers

Architectures distribuées

- Relations d’égal à égal :
  - Pas de connaissance globale du réseau
  - Pas de coordination globale des nœuds
  - Chaque nœud ne connait que les nœuds constituant son voisinage
  - Toutes les données sont accessibles depuis n’importe quel nœud
  - Les nœuds sont volatiles

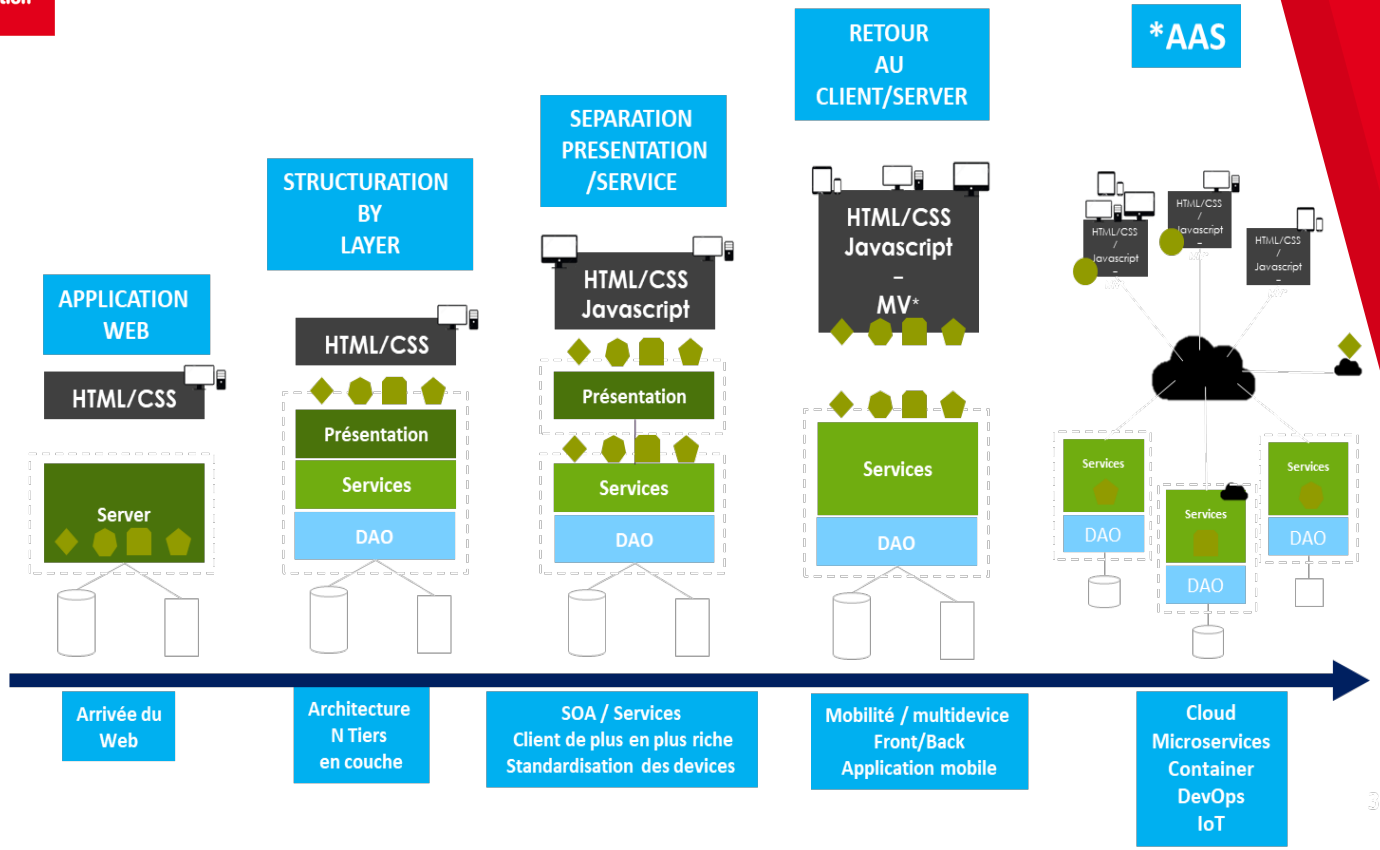
Architectures distribuées

- Avantages :
  - Plus adapté à la montée en charge (scalabilité)
  - Meilleure robustesse en cas de panne (réplication, pas se SPOF : « single point of failure »)

Architectures distribuées

- Inconvénients :
  - Problématiques spécifiques
    - Concurrence
    - Fragmentation des données
    - Gestion de la réplication
    - …

Evolution des architectures au cours du temps



2.

Positionnement des Web services

Les Services Web

- Exemple d’une agence de voyage :
  - Un produit « voyage » = une combinaison de plusieurs produits
    - Réservation des billets de transport
    - Réservation des nuits d’hôtel
    - Réservation des locations de voiture
    - …

Les Services Web

- Exemple d’une agence de voyage :
  - La construction d’un produit « voyage » est le résultat d’informations récupérées auprès de différents fournisseurs :
    - Compagnies aériennes
    - Compagnies ferroviaires
    - Loueurs de voiture
    - Chaînes hôtelières

Les Services Web

- Exemple d’une agence de voyage :
  - Une application de réservation de voyage sollicite d’autres applications réparties pour satisfaire la demande !
  - 2 types de sollicitations :
    - Transformation : adaptation du dialogue en fonction du profil utilisateur
    - Agrégation : appel à des applications proposées par des partenaires ou fournisseurs

Les Services Web

- Exemple d’une agence de voyage :
  - Pour réaliser cela, on s’appuie des Services Web !

Les Services Web

- Un Service Web, c’est quoi ?
  - Fonction distante mise à disposition sur un réseau → accessibilité
  - Infrastructure souple pour des échanges entres des systèmes distribués hétérogènes
  - Localisable à partir de registres
  - Couplage faible

Les Services Web

- Un Service Web, c’est quoi ?
  - Répond à la problématique B2B (SOA -> architecture orientée service)
  - Un service résout un problème donnée
  - Combinaison possible pour résoudre des problèmes complexes

Les Services Web

- Idée générale
  - Un client a un besoin
  - Pour un besoin, plusieurs services et donc fournisseurs peuvent exister (avec ses propres caractéristiques)
  - Le client choisit un fournisseur pour pouvoir utiliser son service (celui qui correspond à son besoin et qui est compatible avec ses exigences (coût, performance, …)

Les Services Web

- Pourquoi peut-on avoir besoin de Services Web ?
  - Besoin d'interopérabilité dans des environnements applicatifs distribués
  - Echanges sur des protocoles standards (HTTP, SMTP, …)
  - Echanges entre des systèmes hétérogènes (environnements différents, langages différents)

Les Services Web

- Les usages
  - Assemblage de composants faiblement couplés
  - Définition indépendantes mais interaction
  - Adapté pour les applications orientées messages
  - Asynchronisme

Les Services Web

- Les acteurs
  - Le client : celui qui invoque le service web
  - Le fournisseur : celui qui fournit et met à disposition le service web
  - L’annuaire : celui qui détient et partage les informations sur les services web

Les Services Web

- Les acteurs
  - Le fournisseur :
    - Serveur d’application (par exemple JEE)
    - Expose un ou plusieurs services (EJB, servlets, enveloppés d’une couche « service »)

Les Services Web

- Les acteurs
  - L’annuaire :
    - Déclaration dans un annuaire = publication

3.

Approches SOAP et REST

Les types de service web

- Services web de type **SOAP** ?

- Services web de type **REST** ?

Les types de service web

- Services web de type **SOAP**
  - SOAP est un protocole
  - SOAP = Simple Object Access Protocol
  - Initialement conçu pour que des applications développées avec différents langages sur différentes plateformes puissent communiquer

Les types de service web

- Services web de type **SOAP**
  - Protocole = règles imposées qui augmentent la complexité et les coûts
  - Mais, standards qui assurent la conformité et sont privilégiés pour certaines applications en entreprise

Les types de service web

- Services web de type **SOAP**
  - Les standards de conformité intégrés incluent la sécurité, l'atomicité, la cohérence, l'isolement et la durabilité (ACID), et un ensemble de propriétés qui permet d'assurer des transactions de base de données fiables

Les types de service web

- Services web de type **SOAP**
  - Les principales spécifications :
    - WS-Security : standardise la manière dont les messages sont sécurisés et transférés via des identifiants uniques appelés jetons
    - WS-ReliableMessaging : standardise la gestion des erreurs entre les messages transférés par le biais d'une infrastructure informatique non fiable

Les types de service web

- Services web de type **SOAP**
  - Les principales spécifications :
    - WS-Adressing : ajoute les informations de routage des paquets en tant que métadonnées dans des en-têtes SOAP, au lieu de les conserver plus en profondeur dans le réseau
    - WSDL (Web Services Description Language) : décrit la fonction d'un service web ainsi que ses limites

Les types de service web

- Services web de type **SOAP**
  - Lorsqu'une requête de données est envoyée à une API SOAP, elle peut être gérée par n'importe quel protocole de couches de l'application : HTTP, SMTP, TCP, ...

Les types de service web

- Services web de type **SOAP**
  - Les messages SOAP doivent être envoyés sous la forme d'un document XML
  - Une fois finalisée, une requête destinée à une API SOAP ne peut pas être mise en cache par un navigateur (pas possible d'y accéder plus tard sans la renvoyer vers l'API)

Les types de service web

- Services web de type **REST**
  - REST **n'est pas un protocole**
  - REST est un ensemble de principes architecturaux adapté aux besoins des services web et applications mobiles légers
  - La mise en place de ces recommandations est laissée à l'appréciation des développeurs

Les types de service web

- Services web de type **REST**
  - L'envoi d'une requête à une API REST se fait généralement par le protocole HTTP
  - À la réception de la requête, les API développées selon les principes REST (appelées API ou services web RESTful) peuvent renvoyer des messages dans différents formats : HTML, XML, texte brut, JSON

Les types de service web

- Services web de type **REST**
  - Le format JSON (JavaScript Object Notation) est le plus utilisé pour les messages : léger, lisible par tous les langages de programmation et les humains
  - Les API RESTful sont plus flexibles et plus faciles à mettre en place

4.

Liens avec la SOA

Architecture orientée Service (SOA)

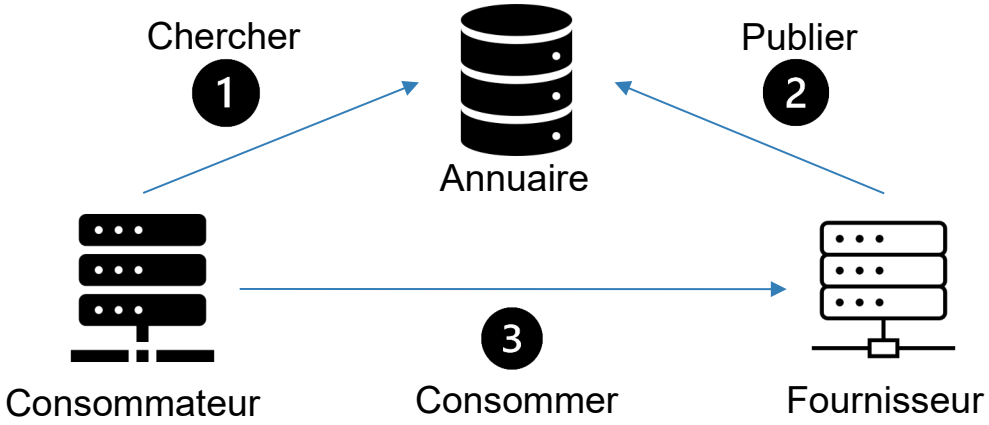
- SOA : Modèle d’interaction applicative qui met en œuvre des services
  - Un client demande un service à un fournisseur

Architecture orientée Service (SOA)

- Objectifs :
  - Réutilisabilité
  - Interopérabilité
  - Réduction du couplage entre les systèmes

Architecture orientée Service (SOA)

- Paradigme SOA
  - **Chercher, publier, consommer**



Architecture orientée Service (SOA)

- Service : composant logiciel distribué exposant des fonctionnalités à forte valeur ajoutée d'un domaine métier
  - Contrat standardisé
  - Couplage lâche
  - Abstraction
  - Réutilisabilité

Architecture orientée Service (SOA)

- Service : composant logiciel distribué exposant des fonctionnalités à forte valeur ajoutée d'un domaine métier
  - Découvrable
  - Autonomie
  - Sans état
  - Composable

Architecture orientée Service (SOA)

- SOA peut s’appuyer sur les web services ! (WSOA)

MODULE WS

WSDL et SOAP

1.

Description du service

Description d’un service

- **Objectif** :
  - Décrire d’un point de vue informatique ce que fait le service

Description d’un service

- La description d’un service doit comporter :
  - La Définition du service
  - Les types de données utilisés notamment dans le cas de types complexes
  - Les opérations utilisables
  - Le protocole utilisé pour le transport
  - L'adresse d'appel

Description d'un service

- Définition en **WSDL** (Web Service Description Language)
  - Métalangage basé sur XML permettant de décrire en détail le service web
  - WSDL est normalisé (W3C)
  - Fichier WSDL disponible sur le serveur pour permettre aux clients de s'informer sur les services web disponibles

Description d'un service

- Ça ressemble à quoi un fichier **WSDL** ?
  - Un bloc de **définition**

```
<definitions name = "HelloService"
  targetNamespace = "http://www.m2i.fr/wsdl/HelloService.wsdl"
  xmlns = "http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns = "http://www.m2i.fr/wsdl/HelloService.wsdl"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
```

Description d'un service

- Ça ressemble à quoi un fichier **WSDL** ?
  - Des types de **données**

```
<message name = "SayHelloRequest">
    <part name = "firstName" type = "xsd:string"/>
  </message>

  <message name = "SayHelloResponse">
    <part name = "greeting" type = "xsd:string"/>
  </message>
```

Description d'un **service**

- Ça ressemble à quoi un fichier **WSDL** ?
  - Des **opérations**

```
<portType name = "Hello_PortType">
    <operation name = "sayHello">
      <input message = "tns:SayHelloRequest"/>
      <output message = "tns:SayHelloResponse"/>
    </operation>
  </portType>
```

Description d'un **service**

- Ça ressemble à quoi un fichier **WSDL** ?
  - Le **binding**

```
<binding name = "Hello_Binding" type = "tns:Hello_PortType">
    <soap:binding style = "rpc"
      transport = "http://schemas.xmlsoap.org/soap/http"/>
    <operation name = "sayHello">
      <soap:operation soapAction = "sayHello"/>
```

Description d'un service

- Ça ressemble à quoi un fichier **WSDL** ?
  - Le **binding**

```
        <input>
    <soap:body
      encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
      namespace = "urn:examples:helloservice"
      use = "encoded"/>
  </input>
```

Description d'un **service**

- Ça ressemble à quoi un fichier **WSDL** ?
  - Le **binding**

```
        <output>
      <soap:body
        encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/"
        namespace = "urn:examples:helloservice"
        use = "encoded"/>
    </output>
  </operation>
</binding>
```

Description d'un **service**

- Ça ressemble à quoi un fichier **WSDL** ?
  - Le **point d'accès** au service

```
<service name = "Hello_Service">
   <documentation>WSDL File for HelloService</documentation>
   <port binding = "tns:Hello_Binding" name = "Hello_Port">
      <soap:address
         location = "http://www.examples.com/SayHello/" />
   </port>
 </service>
```

2.

Protocoles et Bindings

Liaison au protocole (binding)

- Un message SOAP est lié au protocole sur lequel il se déploie
- SOAP se déploie au-dessus de HTTP mais cela ne signifie pas qu'il remplace ou se substitue à quoi que ce soit dans la sémantique du protocole, mais plutôt qu'il en hérite
- La documentation se concentre sur HTTP (protocole le plus utilisé)
- Il est possible d'utiliser SOAP avec d'autres protocoles que HTTP (SMTP, FTP, …)

Liaison au protocole (binding)

- La définition d'une liaison HTTP concerne trois parties :
  - La requête HTTP
  - La réponse HTTP
  - Le cadre d'extension HTTP
- Dans tous les cas, le *media type* "text/xml" doit être utilisé lors de l'encapsulation de messages SOAP dans les échanges HTTP

Liaison au protocole (binding)

- Requête HTTP, grande majorité des liaisons avec la méthode de requête HTTP POST
- En-tête de requête HTTP SOAPAction (SOAPAction HTTP request header) peut être utilisé afin d'indiquer la cible de requête SOAP HTTP
- La valeur que doit renseigner un tel champ représente l'URI de la cible

Liaison au protocole (binding)

- Une valeur de chaîne vide signifie que la cible du message SOAP est fournie par l'URI de la requête HTTP

- L'absence de valeur indique qu'il n'y a pas de cible explicite du message

Liaison au protocole (binding)

- Exemple d’en-tête de requête SOAP HTTP

```
POST /soap/servlet/rpcrouter HTTP/1.1
Host: localhost
Content-Type: text/xml; charset="utf-8"
Content-Length: 345
SOAPAction: "http://electrocommerce.org/">
<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
...
</env:Envelope>
```

Liaison au protocole (binding)

- Réponse HTTP : SOAP suit la sémantique des codes de statut HTTP pour communiquer des informations de statut sur HTTP :
  - Code de statut 2xx = requête du client incluant le composant SOAP a été reçu avec succès, correctement interprété, accepté, …
  - Si une erreur se produit pendant le traitement de la requête, le serveur SOAP HTTP doit renvoyer une réponse HTTP 500 et inclure un élément SOAP **fault** dans le message SOAP de retour

Liaison au protocole (binding)

- Exemple de réponse SOAP :

```
HTTP/1.1 200 OK
Content-Type: text/xml charset="utf-8"
Content-Length: 323
<env:Envelope xmlns:env="
http://www.w3.org/2001/06/soap-envelope">
...
</env:Envelope>
```

3.

Structure d’un message

Structure et propriétés du WSDL

- Utilisation des éléments principaux du XML
  - types : définition des types de messages
  - messages : description des données à transmettre
  - interface : opération abstraite décrivant le communication entre le serveur et le client
  - binding : informations sur le protocole de transport utilisé
  - endpoint : informations sur l’interface de communication (URI)
  - service : point d’accès du service web

Structure et propriétés du WSDL

- Exemple de fichier WSDL
  - https://graphical.weather.gov/xml/SOAP_server/ndfdXMLserver.php?wsdl

4.

SoapUI

SoapUI

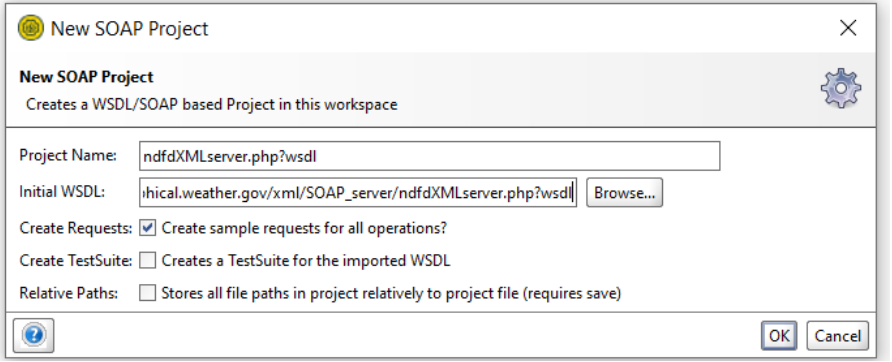
- **SoapUI** est une application open source permettant le test de services web
- https://www.soapui.org/
- https://www.soapui.org/downloads/soapui/

SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Construire un projet SoapUI à partir de la description WSDL d’un service web SOAP
  - Invoquer des opérations d'un service web SOAP
    - https://graphical.weather.gov/xml/SOAP_server/ndfdXMLserver.php?wsdl
    - https://graphical.weather.gov/xml/#use_it

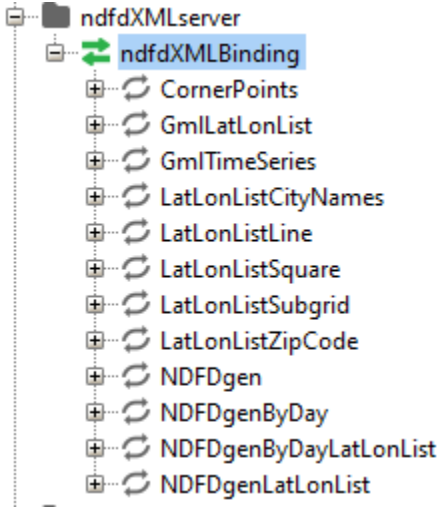
SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Construire un projet SoapUI à partir de la description WSDL d'un service web SOAP
    - File → New SOAP Project



SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Construire un projet SoapUI à partir de la description WSDL d'un service web SOAP
    - Liste des opérations :



SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Invoquer des opérations d'un service web SOAP
    - Sélectionner l'opération ***LatLonListZipCode***
    - Ouvrir l'objet *Request 1*
    - Saisir la valeur « 10001 » dans le paramètre ZipCodeList

```
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.
   <soapenv:Header/>
   <soapenv:Body>
      <ndf:LatLonListZipCode soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <zipCodeList xsi:type="xsd:string">10001</zipCodeList>
      </ndf:LatLonListZipCode>
   </soapenv:Body>
</soapenv:Envelope>
```

SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Invoquer des opérations d'un service web SOAP
    - Exécuter la requête et vérifier le résultat
      - 40.7198,-73.993

SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Invoquer des opérations d'un service web SOAP
    - Sélectionner l'opération ***NDFDgenByDay*** et ouvrir l'objet *Request 1*
    - Saisir les valeurs suivantes :

```
<latitude xsi:type="xsd:decimal">40.7198</latitude>
<longitude xsi:type="xsd:decimal">-73.993</longitude>
<startDate xsi:type="xsd:date">2020-12-08</startDate>
<numDays xsi:type="xsd:integer">1</numDays>
<Unit xsi:type="xsd:string">m</Unit>
<format xsi:type="xsd:string">24 hourly</format>
```

SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Invoquer des opérations d'un service web SOAP
    - Exécuter la requête et vérifier le résultat
      - Prévision météo pour la journée et la ville demandées


```
<temperature type="maximum" units="Celsius" time-layout="k-p24
  <name>Daily Maximum Temperature</name>
  <value>4</value>
</temperature>
<temperature type="minimum" units="Celsius" time-layout="k-p24
  <name>Daily Minimum Temperature</name>
  <value>-2</value>
</temperature>
<probability-of-precipitation type="12 hour" units="percent" t
  <name>12 Hourly Probability of Precipitation</name>
  <value>2</value>
  <value>2</value>
</probability-of-precipitation>
```

SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Invoquer des opérations d'un service web SOAP
    - Sélectionner l'opération ***NDFDgenByDayLatLonList*** et ouvrir l'objet *Request 1*
    - L'objectif est de récupérer les prévisions météo pour la ville de New York (on a déjà les coordonnées) et pour la ville de Beverly Hills (zipcode = 90210)
    - Le paramètre ListLatLon prend une liste de coordonnées
      - Latitude et Longitude séparées par une virgule
      - Chaque couple de coordonnées séparée par un espace

SoapUI

- **Exercice 1 :** inspecter et invoquer un service web
  - Invoquer des opérations d'un service web SOAP
    - Exécuter la requête et vérifier le résultat
      - Quelle température max à Beverly Hills ?

5.

Contenu d’un message

Contenu d'un message SOAP

- Enveloppe (SOAP Envelope) → définit le contenu du message
  - Header (SOAP Header) → optionnel (par exemple pour les autorisations)
  - Body (SOAP Body) → informations sur la requête ou la réponse
- Attachements → optionnels

Contenu d'un message SOAP

- Exemple de message :

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
        <env:Header>
                <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
                 <n:priority>1</n:priority>  </n:alertcontrol>
        </env:Header>
        <env:Body>
                <m:alert xmlns:m="http://example.org/alert">
                 <m:msg>Example message</m:msg>
                </m:alert>
        </env:Body>
</env:Envelope>
```

6.

Code first et Contract First

Code First

- Approche traditionnelle qui consiste à développer le code du service **en premier**
  - Etape 1 : Développement du service
  - Etape 2 : Génération de la documentation du service

Contract First

- Approche différente qui consiste à concevoir le contrat du service **en premier** (avant d'écrire le code)
  - Etape 1 : conception du service
  - Etape 2 : Génération du code à partir de la description du service

Quelle approche choisir ?

- Approche **Contract First**
  - Si l’expérience développeur (DX) compte
  - Pour fournir des API stratégiques
  - Pour assurer une bonne communication

Quelle approche choisir ?

- Approche **Code First**
  - Si besoin d’une livraison rapide
  - Pour développer des API internes

7.

Génération d’artefacts

Génération d'artefacts à partir du WSDL

- Outils disponibles pour générer les artefacts à partir du fichier WSDL (par exemple en Java)

- Artefacts générés :

  - Interface SEI (Service endpoint interface)
  - Classe de service
  - Classe d'exception mappée à partir de la classe wsdl:fault
  - Bean de réponse dérivé de la réponse wsdl:message
  - Valeurs générées de type JAXB (Java Architecture for XML Binding) (classes Java mappées à partir de types de schéma XML)

8.

Les normes d’interopérabilité WS-I

La norme WS-I

- WS-I = Web Services Interoperability
- A l’origine : consortium industriel pour la promotion de l’interopérabilité entre plateformes
  - Rédaction des spécifications des services web WS-*
- Aujourd’hui : norme OASIS (Organization for the Advencement of Structures Information Standards)

Les profils WS-I

- Un profil = un ensemble de contraintes rattachées à la rédaction d'une spécification
  - WS-I Basic Profile
  - WS-I Basic Security Profile
  - Simple Soap Binding Profile

Les profils WS-I

- WS-I Basic Profile
  - Profil de base WS-I
  - Le profil utilise le langage de description des services Web (WSDL) pour activer la description des services sous forme d'ensembles de points de terminaison fonctionnant sur les messages
  - WSI-BP définit un ensemble beaucoup plus restreint de services valides que le schéma WSDL ou SOAP complet
  - WSI-BP restreint généralement la spécification SOAP

Les profils WS-I

- WS-I Basic Security Profile
  - Profil axé sur la sécurité

Les profils WS-I

- Simple SOAP Binding Profile
  - Profil de support pour le profil de base WS-I
  - Définit la manière dont les documents WSDL doivent lier les opérations à un protocole de transport SOAP spécifique

MODULE WS

JAX-WS

Marc Gilbert
marc.gilbert@wijin.tech

JAX-WS

- JAX-WS = API normalisée permettant de créer et de consommer des services Web SOAP en Java
- SOAP est lourd en XML, il est donc préférable de l'utiliser avec des outils / frameworks
- JAX-WS est un framework qui simplifie l'utilisation de SOAP
- JAX-WS fait partie du standard Java

1.

Annotations standards

Annotations JAX-WS

- @WebService
  - Marque une classe Java comme implémentant une interface de service web ou un service web
- @WebMethod
  - Indique une opération du service web

Annotations JAX-WS

- @WebParam
  - Mapping d’un paramètre vers le WSDL
- @WebResult
  - Mapping d’une valeur de retour vers le WSDL

2.

Implémentations : Metro, CXF

Implémentations JAX-WS

- Pour pouvoir utiliser la spécification JAX-WS, il faut une implémentation !

- Différentes implémentations sont disponibles :
  - Metro : projet GlassFish
  - CXF : projet Apache

Implémentation Metro

- **Metro**
  - **com.sun.xml.ws:jaxws-ri** : implémentation JAX-WS proposée par Metro

Implémentation Metro

- **Metro**
  - **Génération automatique du wsdl à l'exécution**
  - Possibilité de générer physiquement le fichier wsdl avec l'utilitaire fourni par Metro : **wsgen**
  - Possibilité de générer les classes Java à partir d'un wsdl : **wsimport**

Implémentation Metro

- **Mise en pratique**
  - Projet Maven avec Java
    - Module « wsdlToJava » : génération du code source à partir d'un WSDL
    - Module « contractFirst » : Implémentation à partir de « wsdlToJava »
    - Module « codeFirst » : création du code source puis génération du WSDL
    - Module « tomcatServer » : déploiement sur un serveur d'application Tomcat

Implémentation Metro

- **Mise en pratique**
  - Projet Maven avec Java
    - Création d’un projet Maven « chapeau » de type « pom »
    - Création d’un module Maven « wsdlToJava »
    - Création d’un module Maven « contractFirst »
    - Création d’un module Maven « codeFirst »
    - Création d’un module Maven « tomcatServer »
    - Création d’un module Maven « soapClient »

Implémentation Metro

- **Mise en pratique – Etape 1**
  - Module « wsdlToJava » : génération du code source à partir d'un WSDL
    - Récupération d'un fichier wsdl et de son xsd
    - Construction du projet avec génération du code à partir du wsdl

Implémentation Metro

- **Mise en pratique – Etape 2**
  - Module « contractFirst » : implémentation à partir du code généré
    - Dépendance « wsdlToJava »
    - Création d’une classe d’implémentation pour le web service

Implémentation Metro

- **Mise en pratique – Etape 2**
  - Module « contractFirst » : implémentation à partir du code généré
    - Création d'une classe Server pour déployer le web service en test
      - Endpoint de publication : http://localhost:9990/CustomerService

Implémentation Metro

- **Mise en pratique – Etape 2**
  - Module « contractFirst » : implémentation à partir du code généré
    - Test avec SoapUI
      - Vérifier la publication du WSDL : http://localhost:9990/CustomerService?wsdl
      - Créer un projet SoapUi et tester le service

SoapUI

- **Exercice SoapUI 2 :** construire une suite de tests simple (Test Suite)
  - New TestSuite
    - Création d’un TestCase avec possibilité d’ajouter des pas

Implémentation Metro

- **Mise en pratique – Etape 3**
  - Module « codeFirst » : création du code source puis génération du WSDL
  - Test avec SoapUI

Implémentation Metro

- **Mise en pratique – Etape 3**
  - Module « codeFirst » : création du code source puis génération du WSDL
    - Créer deux classes « model » : Order et Customer
    - Créer une classe d'implémentation pour les commandes
      - *OrderServiceImpl*
        - 1 méthode de récupération d'une commande par son identifiant → retourne un objet de type Order
        - 1 méthode de construction d'un objet Order pour la réponse

Implémentation Metro

- **Mise en pratique – Etape 3**
  - Module « codeFirst » : création du code source puis génération du WSDL
    - Création d'une classe ServerApp pour déployer le web service en test
      - Endpoint de publication : http://localhost:9980/OrderService

Implémentation Metro

- **Mise en pratique – Etape 3**
  - Module « codeFirst » : création du code source puis génération du WSDL
    - Test avec SoapUI
      - Vérifier la publication du WSDL : http://localhost:9980/OrderService?wsdl
      - Créer un projet SoapUi et tester le service

Implémentation Metro

- **Mise en pratique – Etape 4**
  - Module « tomcatServer » : déploiement sur un serveur d'application Tomcat
  - Test avec SoapUI
  - **Prérequis** : Avoir un serveur Tomcat sur son poste

Implémentation Metro

- **Mise en pratique – Etape 4**
  - Module « tomcatServer » : déploiement sur un serveur d'application Tomcat
    - Ajout de dépendances au fichier pom.xml car Tomcat n'embarque pas nativement d'implémentation pour JAX-WS
    - Dépendance « wsdlToJava »

Implémentation Metro

- **Mise en pratique – Etape 4**
  - Module « tomcatServer » : déploiement sur un serveur d'application Tomcat
    - Création d'une classe d'implémentation pour le web service : *CustomerServiceImpl*
    - Fichier *web.xml* → Descripteur de la servlet web service
    - Fichier *sun-jaxws.xml* → Descripteur du *endpoint* pour notre service

Implémentation Metro

- **Mise en pratique – Etape 4**
  - Module « tomcatServer » : déploiement sur un serveur d'application Tomcat
    - Déploiement du war sur Tomcat
    - Accès au WSDL :

http://localhost:8080/tomcatServer/CustomerWS?wsdl

Implémentation Metro

- **Mise en pratique – Etape 4**
  - Module « tomcatServer » : déploiement sur un serveur d'application Tomcat
    - Test avec SoapUI
      - Créer un projet SoapUI et tester le service

Implémentation Metro

- **Mise en pratique – Etape 5**
  - Module « soapClient » : développement d’un client
  - Test unitaire avec JUnit
  - Test avec SoapUI
    - Utilisation du serveur
    - Utilisation d’un bouchon généré avec SoapUI

Implémentation Metro

- **Mise en pratique – Etape 5**
  - Module « soapClient » : développement d’un client
    - Ajouts au fichier pom.xml
      - Junit
      - wsimport

Implémentation Metro

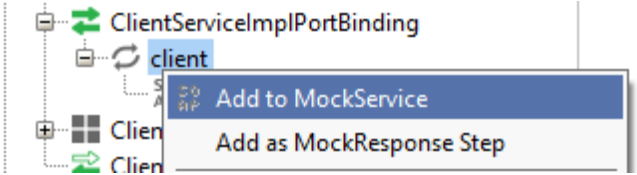
- **Mise en pratique – Etape 5**
  - Module « soapClient » : développement d'un client
    - Vérifier la génération du code
    - Créer une classe pour le client WS : *CustomerServiceClient*
      - Gérer l'URL du WS en attribut
      - Créer une méthode pour appeler le WS

Implémentation Metro

- **Mise en pratique – Etape 5**
  - Module « soapClient » : développement d'un client
    - Créer une classe de test avec Junit pour tester le client WS : *CustomerServiceClientTest*
      - Positionnement de l'URL du WS
      - Méthode de test pour client trouvé
      - Méthode de test pour cas où on ne passe pas de paramètre (fault)

SoapUI

- **Exercice 3 :** Simuler un service web SOAP à partir de sa description WSDL
  - A partir du projet SoapUI *CustomerService*
    - New SOAP MockService
    - Sélectionner le service ***customer***, faire clic droit, puis « Add to MockService »



SoapUI

- **Exercice 3 :** Simuler un service web SOAP à partir de sa description WSDL
  - A partir du projet SoapUI *CustomerService*
    - Possibilité de « bouchonner » la réponse
    - Clic droit sur la réponse créée, puis *Open Request*
      - *Regarder l'URL positionnée dans la requête*
    - Démarrer le bouchon : clic droit sur le *CustomerMockService* et « Start Minimized »

SoapUI

- **Exercice 3 :** Simuler un service web SOAP à partir de sa description WSDL
  - A partir du projet SoapUI *CustomerService*
    - Exécuter la requête et vérifier que la réponse « bouchon » est retournée !

SoapUI

- **Exercice 3 :** Simuler un service web SOAP à partir de sa description WSDL
  - Pour aller plus loin : mock dynamique !

https://www.soapui.org/docs/soap-mocking/creating-dynamic-mockservices/

Implémentation CXF

- **CXF**
  - **org.apache.cxf:cxf-rt-frontend-jaxws** : implémentation JAX-WS proposée par CXF
  - **cxf-rt-transports-http-jetty** : définition des types Java nécessaires pour ne pas utiliser de conteneurs de servlet

Implémentation CXF

- **CXF**
  - **Génération automatique du wsdl à l'exécution**
  - Possibilité de générer physiquement le fichier wsdl avec l'utilitaire fourni par CXF : **java2WS**
  - Possibilité de générer les classes Java à partir d'un wsdl : **wsdl2java**

Implémentation CXF

- **Mise en pratique**
  - Projet Maven avec Java
  - Créer le WS pour récupérer une commande à l'image de ce qui a été fait avec Metro en « code first » (mais cette fois-ci avec CXF)
    - Lancement avec plugin Maven (exec-maven-plugin ; goal = exec:java)
    - Test avec SoapUI

Implémentation CXF

- **Mise en pratique**
  - Génération du WSDL
    - Utilisation de java2WS de CXF
    - Plugin Maven
      - https://cxf.apache.org/docs/maven-java2ws-plugin.html

3.

Sérialisation avec JAXB

JAXB

- **JAXB** = Java Architecture for XML Binding
  - Objectif : faciliter la manipulation d'un document XML en générant un ensemble de classes qui fournissent un niveau d'abstraction
  - JAXB fournit un outil qui analyse un schéma XML et génère à partir de ce dernier un ensemble de classes qui vont encapsuler les traitements de manipulation du document

JAXB

- **JAXB** = Java Architecture for XML Binding
  - JAX-WS utilise JAXB en interne comme couche de liaison pour convertir des objets Java vers et depuis XML

JAXB

- **JAXB** = Java Architecture for XML Binding
  - Les classes générées avec JAX-WS comporte des annotations JAXB
    - @XmlRootElement
    - @XmlElement
    - @XmlSeeAlso
    - @XmlType
    - @XmlAccessorType
    - @XmlSchemaType
    - @XmlEnumValue
    - @XmlEnum
    - @XmlRegistry

4. Asynchronisme

Asynchronisme

- **Problème de HTTP : protocole de « Best effort »**
  - Pas de qualité de service (on ne peut pas avoir la certitude qu'un message est arrivé à destination)
- Architecture **asynchrone** :
  - On reste sur un mécanisme requête/réponse en synchrone
  - Mais la réponse est un acquittement sur la réception de la requête ou simplement l'envoi de la requête

Asynchronisme

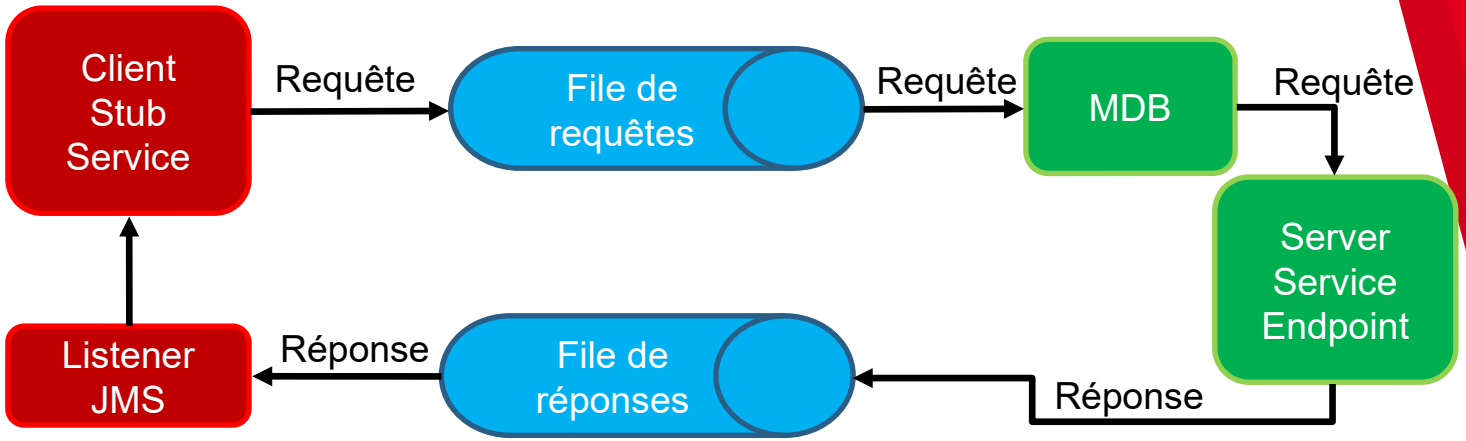
- Architecture **asynchrone** :
  - Scénarios possibles pour la suite :
    - 1/ Le client met à disposition un service web pour être notifié par le serveur de la disponibilité du résultat
      - Le client peut alors invoquer un service de récupération
    - 2/ Le client requête plus tard après un laps de temps, et avec par exemple un identifiant récupéré dans la réponse synchrone du premier appel
  - Peut s’appuyer sur un MOM (par exemple JMS)

5.

Services asynchrones avec JMS

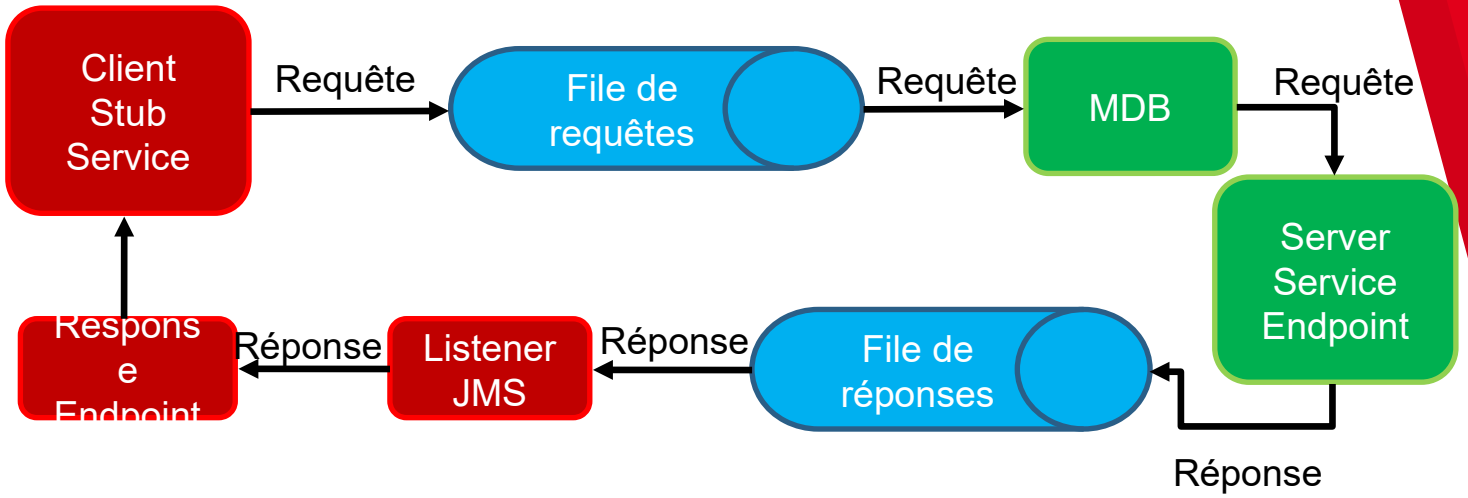
Service synchrone avec JMS

- **Flux d'échange synchrone avec JMS**



Service asynchrone avec JMS

- **Flux d'échange asynchrone avec JMS**



6.

Optimisation des échanges

Optimisation des échanges

- Il peut être intéressant d'optimiser le trafic SOAP au niveau du transport :
  - Des boîtiers accélérateurs peuvent être insérés devant les serveurs afin de compresser les flux XML dans un format binaire bien moins volumineux
  - Cette opération est triviale, puisque le boîtier se contente de détecter le support de gzip sur le client et compresse ensuite le flux XML sortant dans ce format
  - Ce type d'opération s'applique aussi bien aux flux XML que HTML

7.

MTOM et Fast InfoSet

MTOM

- **MTOM** = Message Transmission Optimization Mechanism (W3C)
- Méthode d'envoi de données binaires par services Web (pièces jointes, attachements)
- **MTOM** est habituellement utilisé avec XOP (XML-binary Optimized Packaging)

MTOM

- Efficacité de MTOM = taille des messages envoyés dans le flux

- Puisque SOAP utilise le XML, les données binaires doivent être encodées en texte, généralement du « Base64 », qui a la particularité d'augmenter la taille des données binaires de 33%

- MTOM propose un moyen d'envoyer ces données dans sa forme originale, ce qui évite une augmentation de la taille due à l'encodage du texte

MTOM

- **MTOM** doit être activé sur la classe d'implémentation du service
  - Annotation @MTOM
  - 2 paramètres :
    - **enabled** : activation de MTOM (booléen)
    - **threshold** : seuil pour codage XOP (entier >= 0)

MTOM

- **MTOM** peut être utilisé également côté client pour l'upload
  - Activation de MTOM sur le binding (*setMTOMEnabled(true)*)

```
// Activation de MTOM sur le client
BindingProvider bp = (BindingProvider) server;
SOAPBinding binding = (SOAPBinding)
   bp.getBinding(); binding.setMTOMEnabled(true);
```

MTOM

- **Test avec SoapUI**
  - https://www.soapui.org/docs/soap-and-wsdl/attachments/
  - Création d’un projet pour upload de fichier

Fast Infoset

- **Fast Infoset** = spécification d’un XML binarisé
  - Format de codage binaire pour l'ensemble des informations XML comme alternative au format de document XML
  - Sérialisation plus efficace que le XML basé sur du texte
- Fast Infoset vise à optimiser à la fois la taille du document et les performances de traitement

Fast Infoset

- **Fast Infoset** = spécification d'un XML binarisé
  - Propriété sur le client

```
getRequestContext().put(com.sun.xml.ws.client.ContentNegociatio
    n.PROPERTY, « pessimistic »);

Map<String, Object> ctxt =
    ((BindingProvider)proxy).getRequestContext();
ctxt.put(JAXWSProperties.CONTENT_NEGOTIATION_PROPERTY,
    "pessimistic");
```

MODULE HTML / CSS

HTML - CSS

BLAIVIE Olivier
o.blaivie@gmail.com

Introduction

- Langage Web: Ensemble de technologie permettant d'afficher des pages Web

- Un site web peut être définis par du **code**, du **texte**, des **images**, des **styles**, etc …

- Technologies pour structurer un site web :
  - HTML (HyperText Markup Language) : Code structurel
  - CSS (Cascading StyleSheets) : Feuilles de styles
  - JavaScript: Actions, Interactions avec l'utilisateur …

1.

Le HTML

Introduction HTML

- **HTML1**: Naissance du HTML, crée par Tim Berners-Lee en 1991

- **HTML2**: (1994) => Nouveautés mineures.
- **HTML3**: (1996) => Tableaux, Scripts, positionnements…
- **HTML4**: (1998) => Tableaux complexes, formulaires, dissociation des style (CSS).

- **HTML5**: (2014) Version actuelle => Simplification, nouveautés ++

Les bases du HTML

- HTML est un langage de balise.

- Une balise permet d’indiquer la nature du texte.

- La syntaxe pour utiliser une balise est la suivante:
  - <balise_name> </balise_name> ou <balise_name />

- La syntaxe pour déclarer une page html est divisée en 4 items:
  - <!DOCTYPE html>
  - <html> </html>
  - <head> </head>
  - <body> </body>

- L’ajout de commentaire en HTML se fait de la manière suivante :

Les balises

- Il faut toujours structurer son code à l'aide des balises.

- Quelques exemples :

Paragraphe	
Titres	<h1> </h1> , <h2> </h2> … <h6> </h6>
Saut de ligne	
Liste à puce	<ul> <li> *elmt1* </li> <li> *elmt2* </li> … </ul>
Image	<img src="image.png" alt="monImage"/>
Lien externe	<a href="*url* "> *Text* </a>
Texte important	elmt

Les attributs

- Un attribut est une information additionnelle/spécification liée à une balise.

- Les attributs se déclarent dans les balises ouvrantes et sont cumulables.

- La syntaxe d’un attribut est la suivante :
  - <balise_name atr1=”value1” atr2=”value2” ...> <balise_name/>

- Quelques exemples :
  - <h1 color=”red” align=”center” ...> </h1>
  - <img src=‘’image.png’’ alt=‘’monImage’’ width=”50%”/>
  - <a href=‘’https://www.m2iformation.fr/”></a>

TP 1

Créer une page (home.html) qui aura pour nom ‘MyCRM’ et qui permettra d’afficher à l’écran les informations suivantes :

- Un titre d’onglet “CRM”
- Une icone de CRM
- Un titre (centré): “MyCRM”
- Un sous titre: “Description”
- Une description: “Cette application permet de faire de la gestion de commande et de client.”
- Les mots “commande” et “client” doivent ressortir

TP 2

Reprendre la page et ajouter :

- Une petite image de CRM
- Deux titres plus petits que la description: “Clients” et “Commandes”
- Une liste de client: “Jean Dupont”, “Sarah Abdram” et “Rachel Taeck”
- Une liste de commande: “CMD01”, “CMD02 ”, “CMD03” et “CMD04”
- Un lien de redirection vers le site Google

Les spécificités du tableau

- Principales balises pour afficher un tableau :
  - Structure : <table></table>
  - En-têtes (header): <thead></thead>
  - Contenu (body): <tbody></tbody>
  - Ajout d’une ligne: <tr></tr>
  - Ajout de colonne : <th></th> et <td></td>

- Exemple d’attribut :
  - Bordures : border="1" (border=0 par défaut !)
  - Fusionner (horizontalement) : colspan="2"
  - Fusionner (verticalement) : rowspan="2"
  - Gérer l’alignement: align:"right"
  - Fixer la taille de cellule: width:"50%"

Les spécificités du formulaire

- Principales balises pour afficher un formulaire :
  - Structure : <form></form>
  - Identification unique : name=”form_name”
  - Attribut de méthode: method=”post"

- Balises de formulaires :
  - Textes : <label>
  - Champs de saisie: <input> (text, number, date, checkbox, password ...)
  - Liste déroulante: <select> - <option></option> - </select>
  - Bouton: <button></button>
  - Champs de saisie spécial: <textarea>

TP 3

Reprendre la page en remplaçant les listes à puces des clients et des commandes par 2 tableaux, centré sur la page et affiché l'un en dessous de l'autre.

Les tableaux doivent avoir la même représentation que ci-joint :

Liste des clients

Nom	Prenom
Jean	Dupont
Sarah	Abdram
Rachel	Taëck

Liste des commandes

Numéro de commande	Etat
CMD01	Payé
CMD02	Payé
CMD03	Non Payé
CMD04	En attente de paiement

TP 4

- Créer une nouvelle page en reprenant le même titre d’onglet et de page centré. Ajouter un sous-titre “Ajouter un client:”

- Créer un formulaire de type POST avec :
  - *3 champs de saisie “Nom”,“Prenom” et “Password”*
  - *Un champs de sélection “Civilité” avec les choix “Mr”, “Mme”, “Melle“*
  - *Un champs de type checkbox avec le texte “Actif” coché*

- Ajouter un lien de redirection vers la page d’accueil.

2.

Le CSS

Introduction au CSS

- Naissance du CSS pour accompagner le HTML en 1996
- Le CSS permet de gérer les couleurs, styles, polices, positionnement, …
- Complémentaire avec le HTML
- Comment l’utiliser ?
  - Directement dans le code HTML avec la balise <style> </style>
  - Dans un fichier externe .css
  - Lien classique : <link rel=”stylesheet” type=”text/css” href=”style.css” />

La Syntaxe

- Structure CSS : Sélecteurs + Propriétés + Valeurs
- Attributs spéciaux : class et id

```
h1{
    color: blue;
}
```

```
<h1 class="class_name"></h1>

.class_name{
    color: blue;
}
```

```
<h1 id="id_name"></h1>

#id_name{
    color: blue;
}
```

Quelques exemples de propriété

Propriété	Syntaxe	Exemples
Couleurs	color background-color	color: red; / color: rgb(255, 99, 71) / color:#F10606 background-color: gray; / background-color: #808080
Police	font-family	font-family: Times; / font-family: Impact, Verdana, "Arial Black";
Taille texte	font-size	font-size: 15px; / font-size: small / font-size: 1,5em;
Styles	font-style font-weight …	font-style: italic; font-weight: bold; / text-decoration: underline;
Alignement	text-align	text-align: center; / text-align: right / text-align: justify
Marges	margin padding	margin: 5px 5px 5px 5px; / margin-top: 10px; / margin-bottom: 30px; / padding: 10px; / padding-left: 15px

Notions importantes

- Héritage:
  - Les propriétés enfants héritent des propriétés du parent
- Surcharge de propriété:
  - Une propriété redéfinie écrase l’ancienne

Positionnement

- Le positionnement permet de gérer la mise en page des éléments à l'aide de la propriété CSS position.

- Il faut différencier 4 types principaux de positionnement :
  - **(static)**: Valeur par défaut - Positionnement statique des éléments
  - **relative** : Positionnement ajustée par rapport à la position initiale de l'élément
  - **absolute** : Positionnement ajustée par rapport au parent le plus proche (possibilité </style>des éléments les uns sur les autres)
  - **fixed** : semblable a absolute, mais en s'appuyant sur la fenêtre visible

- Positionnement avec **top**, **left**, **bottom** et **right.** Les valeurs **négatives** sont possibles.

- La propriété z-index permet de prioriser l'élément à afficher (en cas de chevauchement)

TP 5

- Créer une nouvelle page html + un fichier CSS associé. Tous les styles doivent être réalisés dans le fichier CSS.
- La page doit afficher :
  - Un grand titre “MyCRM”, écrit en gris et centré.
  - Un sous-titre “Détails des commandes” souligné.
  - 3 sections nommés “Petite:”, “Moyenne:” et “Grande:” Ajouter respectivement les valeurs “12”, “150” et “13” avec la balise “paragraphe” sous chaque section.
  - Les sections et les paragraphes doivent être décalé de 25 pixels.
  - La valeur “150” doit être écrit en plus gros, et en rouge.
  - La couleur de fond doit correspondre au code couleur **#ECF0F1**.

TP 6

Reprendre la page précédente et effectuer les modifs à l’aide du CSS.

1. Réduire l’espace entre les trois sections “petites”, “moyennes” et “grandes”. Les 3 valeurs doivent être à droite de leur titre respectif, et alignés horizontalement toutes les trois (RELATIVE).

2. Ajouter l’image du CRM de l’onglet dans le coin supérieur gauche de l’écran. Ajuster sa taille (ABSOLUE).

3. Ajouter un footer “Footer” en bas de la page qui ne peut être visible qu’en scrollant. Ajouter une autre image du CRM, avec un fond de couleur jeune, au coin supérieur droit de l’écran. L’image du CRM doit rester visible même en scrollant (FIXED).

Display, div et span

- La propriété display définit le type d’affichage utilisé (inline vs block)
- Les balises div et span sont des conteneurs génériques
- Ils agissent sur la propriété display

Le CSS dynamique

- Le CSS permet de modifier les style dynamiquement (pseudo-class)

```
selecteur:pseudo-class {
    propriété: valeur;
}
```

Action	Syntaxe
Survol	hover
Clic	active
cliqué	focus
Sélectionné	checked
Activé / désactivé	enabled / disabled

TP 7

Reprendre la page du formulaire. Effectuer les modifications :

1. La couleur de fond: **#ECF0F1**
2. Le titre « Application CRM AN21 »: **gris** et **centré**
3. Les labels écrits en **gras**, sauf le mot « Actif ».
4. Le lien de retour est écrit en **vert**, il change en **rouge** quand la souris passe dessus.
5. Les champs de **saisie** et de **sélection** doivent se trouver **sous** leurs input
6. Le **checkbox** et le **texte** « Actif » doivent être sur la **même ligne**, précédé du texte « **Statut** » écrit en gras.
7. Chaque bloc de label/input doit avoir un écart de **15 pixels**.
8. Ajouter un bouton nommé « Valider », aligné à droite, avec un fond de couleur orange qui passe au jaune au clic. Le curseur de la souris devra être de type pointeur.

MODULE JS

JavaScript / JavaScript avancé

BLAIVIE Olivier
o.blaivie@gmail.com

1.

Introduction JS

Qu’est ce que le JavaScript ?

- JavaScript, c’est quoi ?
  - JavaScript n’est pas Java !
  - Langage de script permettant d'interagir avec une page Web
  - JS permet de créer des pages dynamiques / des interactions
  - JS fonctionne avec HTML et CSS
  - Standardisé par EcmaScript

- Implémentations possibles :
  - Balises: <script></script>
  - Fichier externe: <script type="text/javascript" src=”file.js”></script>

Syntaxe

- Une instruction JS se termine toujours par un point virgule ;

- Les déclarations de variables doivent êtres précises - Attention à la casse !
  - myVar et Myvar sont 2 variables différentes.

- Les commentaires s'écrivent de deux façons :
  - // commentaires
  - /* commentaires

    sur plusieurs

    lignes

    */

Les variables (1/2)

- Déclarer / Manipuler une variable
  - Mot clé var / let
  - Déclarer une variable: let varName;
  - Initialiser une variable : varName = 1;
  - Additionner: varName = varName + 4; / varName += 5
  - Soustraire: varName = 6-1; / varName -=4
  - Multiplier / Diviser: varName = 6*5; / varName = 4/2;
  - Incrément / Décrément: varName++; / varName--;

- Les constantes:
  - Variables non modifiables
  - Mot clé const
  - Déclarer une constante: const varName = 0;

Les types de variable

- On dit que JavaScript à un typage faible

- Il existe 7 types primitifs en JavaScript :

String	let ville = "Toulouse";
Number	let prix = 55;
Boolean	let existe = true; / let existe = false;
Null	let varNull = null;
Undefined	let ville;
Object	let ville = { nom: "Toulouse", cp:"31100" }
Symbol	...

Outils et méthodes utiles

- Afficher une alerte
  - alert("mon message"); / alert(maVariable);
- Afficher la console du navigateur :
  - console.log("mon message"); / console.log(maVariable, "1");
- Concaténation :
  - let a = "Je"; / let b = "suis"; / let c = a + b;
- Conversion de type:
  - String en nombre (ou float) : parseInt() - parseFloat()
  - All en string: string()

TP 1

Reprendre la page du formulaire.

1. Créer un fichier JavaScript et le lier au formulaire.

2. Votre fichier JS doit permettre :
    a. Déclarer 2 variables, implémenter avec des valeurs de type number
    b. Déclarer une constante qui est égal à la somme des 2 variables
    c. Afficher un message d’alerte « Bienvenue sur la page de formulaire »
    d. La console du navigateur affiche le contenu des variables.
    e. La console affiche « Le résultat est » + le résultat du calcul.

Conditions de contrôle

- Condition IF - ELSE

```
if (condition1) {
    // Traitement IF
} else if (condition2) {
    // Traitement ELSE IF
} else {
    // Traitement ELSE
}
```

- Condition ternaire :
  - condition ? if true : if false;

```
function getFee(isMember) {
  return (isMember ? '$2.00' : '$10.00');
}
```

Les boucles

- FOR : for (let i=0; i<10; i++) { … };
- FOR … IN : for (index in tab) { tab[index] };
- FOR … OF : for (index of tab) { index };
- WHILE : while (condition) { … };
- DO … WHILE : do {… } while {condition};
- Le mot clé **break** permet de sortir d’une boucle prématurément.

Objets et tableaux

- Déclaration d'une structure d'objet :

```
class Ville {
    constructor(nom, cp) {
        this.nom = nom;
        this.cp = cp;
    }
}
```

- Création d'un objet : `let ville1 = new Ville("Toulouse", "31100");`

- Accès aux propriétés d'un objet: `let nomDeVille = ville1.nom;`

- Déclaration d'un tableau :

```
let tableau = [];
let tableau = ["value1", "value2", "valueX"];
```

- Accès aux données du tableau: `let premierElement = tableau[0];`
  - Ajouter un élément: tableau.push("x");
  - Compter les items: tableau.length;
  - Supprimer élément: tableau.pop();

Opérateurs logiques

- Les conditions ont besoin d'opérateurs logiques, en voici quelques uns :

Egalité	==
Eglité stricte	===
Différent de	!= ou <>
Différent stricte de	!==
Plus grand / petit	> / < / >= / <=
ET / OU / NON	&& / || / !

Les fonctions, Document et évents

- Définition d'une fonction :
  - Retour attendu : mot clé 'return'

```
function afficherMessage(msg) {
    console.log(msg);
}
```

- L'interface Document permet de récupérer:
  - un élément par son id: document.getElementById('id');
  - la valeur d'un formulaire: document.getElementById('id').value;
  - la valeur d'un texte: document.getElementById('id').innerHTML;
  - Affectation: document.getElementById('id').innerHTML = "test";

- Quelques évènements :
  - Récupération du clic : onclick
  - Touche clavier: onkeypress

```
<button onclick="afficherMessage('monMessage')">Envoyer</button>
```

TP 2

Reprendre la page du formulaire.

1. Créer une fonction qui affiche une alerte avec le nom et le prénom saisie dans les champs de texte correspondant. Déclencher la fonction au clic du bouton ‘Valider’.

2. Ajouter une condition qui compte le nombre de caractère du nom, s’il est inférieur à 5 : afficher une alerte « Nom trop court », sinon afficher « Bonjour Monsieur » avec le nom saisi.

3. La condition doit maintenant compter les caractères du nom ET du prenom. De plus, remplacer l’alerte par un texte écrit sous le statut. Si le nom/prénom est trop court, ce texte doit être rouge. Sinon, il doit être vert.

TP 3

Reprendre la page du formulaire.

1. Supprimer le texte saisi dans Nom et/ou prénom s’il est inférieur à 5 caractères.

2. Si le nom et le prénom réponds aux conditions ET si le mot de passe n’est pas vide, alors il faut créer un objet de type « Client » avec toutes les informations renseignées dans le formulaire. Vérifier l’enregistrement avec un console.log ou une alerte.

3. Créer un tableau de client, et ajouter les au fur et à mesure. A chaque ajout, afficher une alerte indiquant le nouveau nom du client, et le nombre de client total.

TP 4

Reprendre la page du formulaire.

1. Ajouter un champs “Age” sous le prénom (label + input).

2. En appuyant sur le bouton ‘Valider’, appeler un fonction qui à pour objectif :
   a. d’enregistrer un objet Client qui reprends toutes les valeurs du formulaire **SI** les contrôles sont respectés :
      i. Nom et prénom avec 5 char min
      ii. Password non vide et max 6 char
      iii. Age renseigné
   b. SINON créer une sous fonction qui retourne un message d’erreur indiquant les champs en erreur.
   c. Un message “Félicitation, client créé” en vert en cas de succès
   d. OU Un message avec le message d’erreur en rouge en cas d’échec

Tableaux : Méthodes

- Voici quelques méthodes d’accès au tableau :
  - Créer un tableau vide: `let tab = [ ];`
  - Ajouter un élément: `tableau.push(elem);`
  - Compter les items: `tableau.length;`
  - Supprimer élément: `tableau.pop();`
  - Parcourir les éléments: `for (let i = 0; i < tab.size; i++) { };`
  - Parcourir les éléments: `for (index in tab) { tab[index] };`
  - Récupérer un élément: `table[0];`

TP 5

Reprendre la page du formulaire.

1. Ajouter une section ‘Informations sur les clients’ sous le formulaire, à exactement 50px du dernier élément.

2. Insérer dans cette section :
   a. un texte ‘Information souhaitée’ suivi d’une liste déroulante avec les choix ‘Recherche Client’ et ‘Moyenne age’
   b. un bloc de saisie de recherche (label + input)
   c. un bouton ‘Valider’.

3. En cliquant sur ‘Valider’, afficher les informations souhaitées sous le bouton:
   a. Si Moyenne : “Moyenne age des clients = “ + moyenne calculée
   b. Si Recherche:
      i. Si client trouvé : Afficher les infos du clients avec le nom en majuscule.
      ii. Sinon : Message d’erreur

2.

AJAX et JQuery

AJAX - Introduction

- AJAX (Asynchronous JavaScript + XML) est un **modèle** regroupant plusieurs technologies (HTML, CSS, JS, XML).

- Plus précisément, il s’agit d’une approche pour **modifier dynamiquement** du contenu Web, **sans recharger une page**.

- AJAX communique via l’objet XMLHttpRequest pour récupérer des données d’un serveur distant.

- Les requêtes effectués par AJAX se font via JavaScript - Il s’agit de requêtes **HTTP** (GET - POST).

AJAX - Fonctionnement

- Les technologies utilisées dans AJAX :
  - XML: Format choisi pour l’échange des données avec le serveur (aujourd’hui remplacé par du JSON)
  - XMLHttpRequest: Object permettant la communication asynchrone
  - JS: Interactions avec l’utilisateur + injection dynamiques des datas
  - HTML/CSS: Présentation / Affichage des datas

- Les 4 étapes à suivre pour effectuer une requêtes asynchrone :
  - Création de l'objet XMLHttpRequest
  - Initialisation de la requête (méthode HTTP, url)
  - Envoie de la requête

AJAX - Méthodes XMLHttpRequest

- Dans AJAX, plusieurs méthodes sont définies :

- open() = initialise une requête en JavaScript ;
- send() = envoie une requête asynchrone par défaut ;
- abort() = abandonne la requête si celle-ci a déjà été envoyée ;
- setRequestHeader() = définit la valeur de l'en-tête HTTP de la requête ;
- getResponseHeader() = retourne la chaine de caractère contenant le texte de l'en-tête de la réponse spécifié ou null si la réponse n'a pas été reçue ;
- getAllResponseHeaders() = retourne la réponse de tous les en-têtes ou null si aucune réponse n'a été reçue ;
- overrideMimeType() = surcharge le type MIME retourné par le serveur.

AJAX - Exemple

```
// Création XMLHttpRequest
let xhr = new XMLHttpRequest();

// Initialisation avec la méthode open() + déclaration format + envoi avec send()
xhr.open("GET", "184.152.10.25/users");
xhr.responseType = "json";
xhr.send();

// Chargement de la réponse avec onload() + Récupération du statut
xhr.onload = function(){
  if (xhr.status != 200){
    alert("Erreur " + xhr.status + " : " + xhr.statusText);
  }else{
    alert(xhr.response.length + " octets  téléchargés\n" + JSON.stringify(xhr.response));
  }
};

// Gestion des erreurs avec onerror()
xhr.onerror = function(){
  alert("La requête a échoué");
};

// Gestion du téléchargement de la réponse en temps réel avec onprogress()
xhr.onprogress = function(event){
    // lengthComputable = booléen; true si la requête a une length calculable
    if (event.lengthComputable){
        //loaded = contient le nombre d'octets téléchargés
        //total = contient le nombre total d'octets à télécharger
        alert(event.loaded + " octets reçus sur un total de " + event.total);
    }
};
```

JQuery- Introduction

- JQuery est une bibliothèque JavaScript.

- JQuery à pour but:
  - de simplifier la manipulation du DOM HTML
  - de gérer les événements
  - de communiquer dynamiquement avec le CSS
  - de simplifier l’utilisation d’AJAX

- JQuery s’ajoute de 2 façons :
  - En téléchargeant la librairie
  - En utilisant le CDN (Content Delivery Network)

```
<script src=”jquery3.4.1”></script>
<script src=”<url_cdn>”></script>
```

jQuery- Exemples

```
$(document).ready(function(){
    $("span").first().css("color", "green");
    $("span").eq(1).css("color", "blue");
    $("span").last().css("color", "orange");
});
```

```
$(document).ready(function(){
    /*Sélectionne tous les ancêtres des
éléments ul et ajoute une bordure
    noire d'1px autour de chacun d'entre
eux*/
    $("ul").parents().css("border", "1px
solid black");

    /*Ne cible que les ancêtres des éléments
ul qui sont des div et change
    leur couleur de fond en bleu*/
    $("ul").parents("div").css("background-
color", "lightBlue");
});
```

```
$("#envoyer").click(function(){

    $.ajax({
        url : 'send_mail.php',
        type : 'POST', // Le type de la requête HTTP, ici devenu POST
        data : 'email=' + email + '&contenu=' + contenu_mail, // On fait passer nos variables,
exactement comme en GET, au script more_com.php
        dataType : 'html'
    });

});
```

TP 6

Reprendre la page du formulaire.

1. Intégrer la librairie jQuery.

2. Dans la liste déroulante des informations, ajouter une option pour lister les utilisateurs existants.

3. Cette option doit déclencher une requête sur le serveur pour récupérer la liste des utilisateurs présents dans la base de données.

4. Afficher alors les username et mail de chaque utilisateur récupéré.

MODULE BOOTSTRAP

Le framework BOOTSTRAP

BLAIVIE Olivier
o.blaivie@gmail.com

Introduction

- Bootstrap, c’est quoi ?
  - Un framework CSS (CSS + JS)

- Pourquoi l’utiliser ?
  - Gain de temps de développement
  - Un framework très largement utilisé
  - Facile à implémenter / utiliser
  - Permet le responsive
  - Documentation très complète

- Comment l’utiliser ? 2 solutions :
  - Ajout de la librairie
  - Import des liens (CDN)

Fonctionnement / Positionnements

- Bootstrap s'organise sur un système de grille (colonnes / lignes)

- Utilisation de l'espace avec les conteneurs:
  - class container: Adaptation de la largeur
  - class container-fluid: Largeur fixe (100%)

- Découpage d'une page, 5 tailles d'écrans par défaut :
  - Extra small: .col- ⇒ Smartphone
  - Small: .col-sm- ⇒ Smartphone / Tablette
  - Medium: .col-md- ⇒ Tablette
  - Large: .col-lg- ⇒ PC
  - Extra large: .col-xl- ⇒ Grand écran / TV / Retro

  - Découpage custom: @media(min-width: ...px) { ... }

Organisation des lignes / colonnes

- Ajout d’une ligne :
- Ajout d’une colonne:
- Découpage en 12 colonnes !
- Exemple:

```
<div class=”row”>
    <div class="col-4">Element 1</div>
    <div class="col-8">Element 2</div>
</div>
```

Element 1	Element 2

Quelques exemples …

- Pour récapituler, Bootstrap propose des composants “clé en main” (à customiser si besoin).

- Voici quelques exemples :
  - Barre de navigation: class=”navbar”
  - Boutons: class="btn btn-primary”
  - Popup d’info: class="toast
  - …

- Toute la doc’ est en ligne !

https://getbootstrap.com/docs/4.5/getting-started/introduction/

TP 1

Reprendre les pages créées.

1. Importer Bootstrap dans les pages (CSS + JS)
2. Refaire la mise en forme avec Bootstrap.
3. Ajouter le composant Navbar

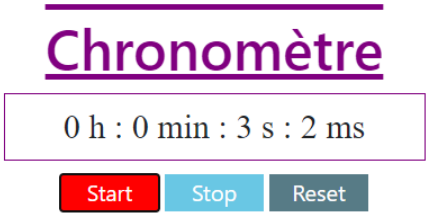
Surcharge de CSS

- En important la librairie BootStrap, le CSS utilisé par défaut est celui proposé par la librairie.
- Pour ajouter des propriétés CSS, il est nécessaire de **surcharger** le CSS existant.
- Pour écraser un CSS existant, il faut utiliser le mot clé **!important** derrière la propriété css à surcharger.
- Exemples avec les composants “Card” et “Alert” de Bootstrap

TP Recap

Un client vous demande de créer une page Web. Ces exigences sont les suivantes :

- Créer une page Web qui affiche un Chronomètre.
- Le Chronomètre peut être démarré, interrompu ou réinitialisé par un utilisateur.
- L’ application doit montrer les heures, minutes, secondes et millisecondes qui défilent ou qui sont stoppées.
- L’application doit s’approcher de la maquette suivante :



Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-132

SE CONNECTER S'INSCRIRE

MODULE Angular

Le framework Angular

BLAIVIE Olivier
o.blaivie@gmail.com

1.

Introduction et outils

Environnement de travail

- **NodeJS:** Toolbox pour dev et réseaux

https://nodejs.org/en/download/ (LTS)

- **VSCode:** IDE multi language gratuit

https://code.visualstudio.com/download

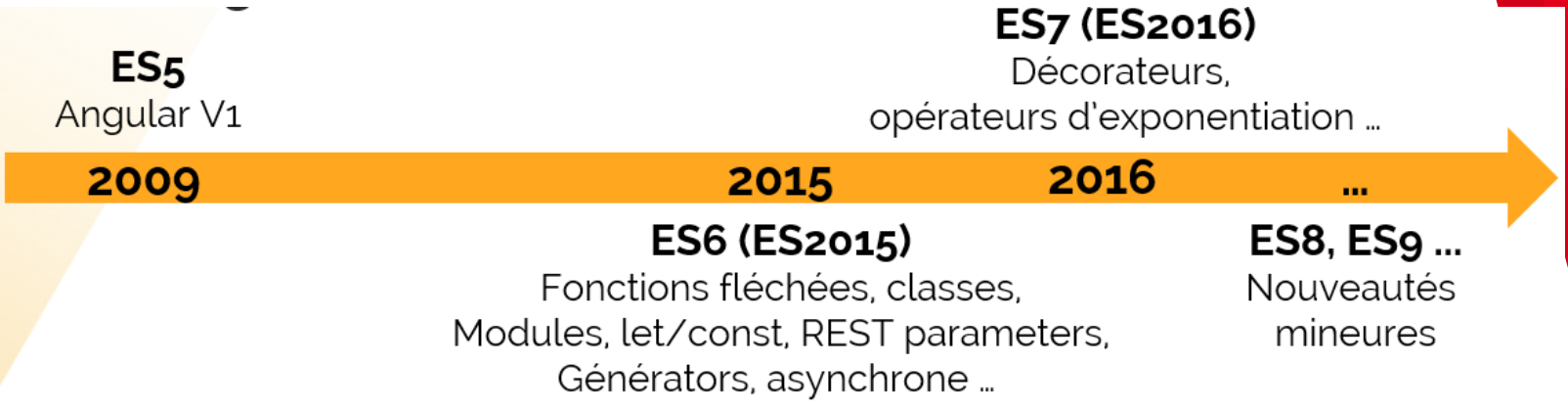
  - Plugin Angular Essential
  - Config settings (Auto Save + onFocusChange
  - Config settings (Icons): material-icon-theme

- **Git:** GitBash + GitLab

Normes ES

- **ECMASCRIPT ?**
  - Standards / Normes JavaScript
  - Permet de définir la **syntaxe**, les **type** de variables …

- **ES et Angular:**



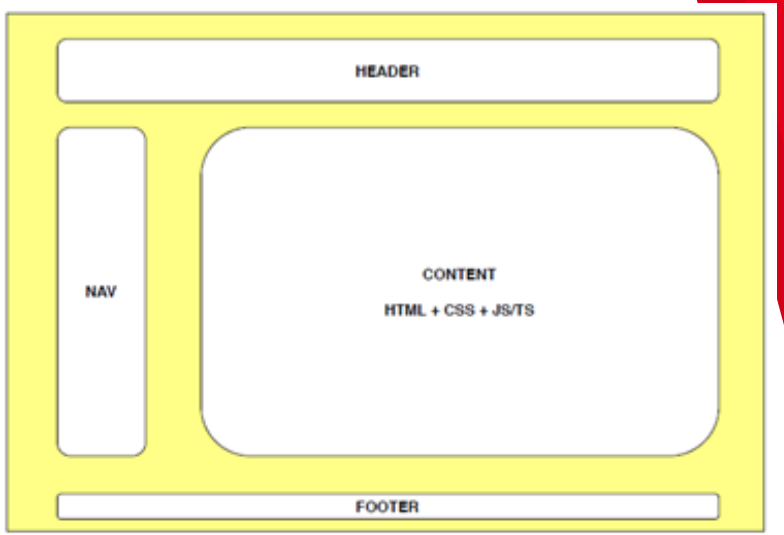
Un peu d'histoire

- **Angular, un framework ?**

- **AngularJS (2010) à Angular 2 (2016)**

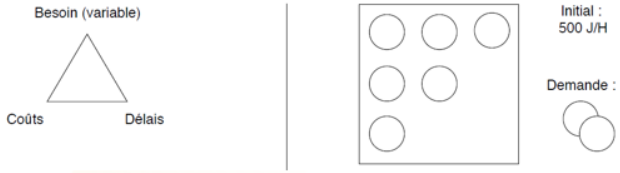
- **Termes Clés :**
  - TypeScript
  - Décorateurs
  - WebComponents

WebComponents



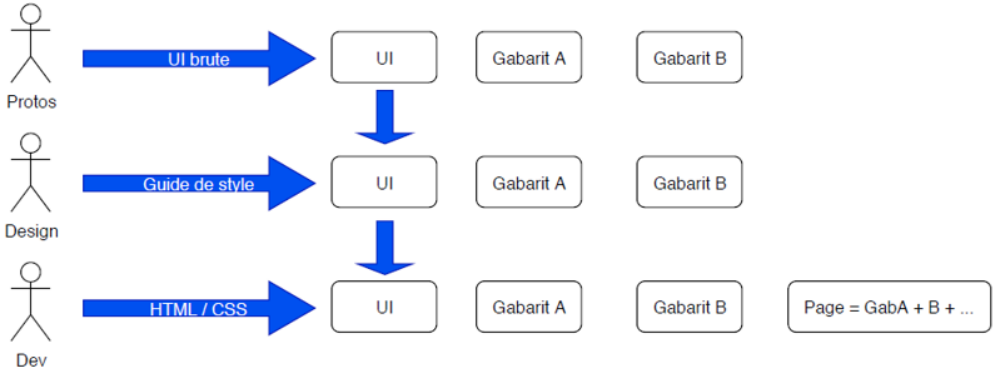
De l'agilité au logiciel

Principes de l'agilité :



Développement traditionnel :

Développement parallèle :



2.

Les Schématics

Démarrage d'un projet

- **Les Schématics**

- **Angular CLI**

- **Premières commandes :**
  - ng new *<app_name>* : Génération d'un projet
  - ng serve : Serveur de développement
  - ng build : Génération des sources
  - npm start : Script de démarrage
  - npm run *<script_name>* : Exécuter un script
  - ng test : Exécuter les tests

Configuration du projet

- **Fichier de configuration / dépendances d'un projet Angular :**
  - angular.json
  - package.json

- **Ajout de librairie :**
  - npm install *<lib_name@lib_version>*
  - npm add *<lib_name@lib_version>*

- **Intégration de la documentation (Compodoc) :**
  - La lib Compodoc permet de générer une doc rapide
  - https://compodoc.app/guides/installation.html

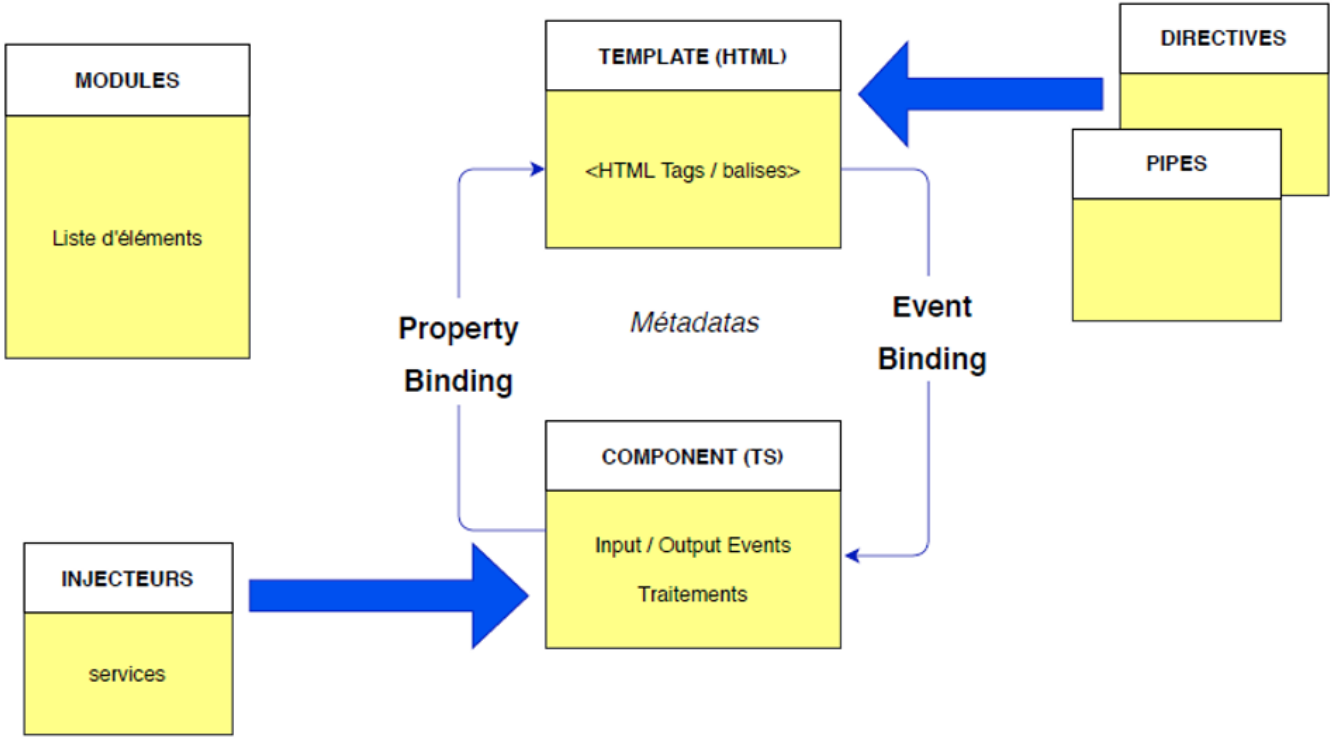
Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-10
  - M2IC-11

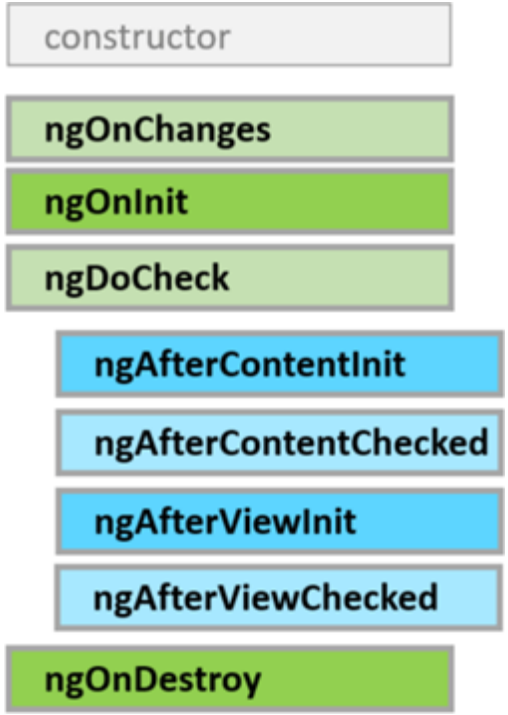
3.

Architecture

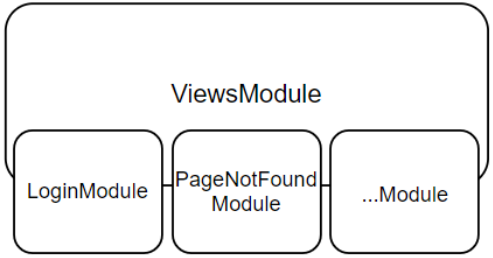
L'architecture Angular

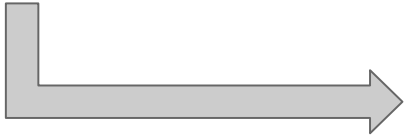


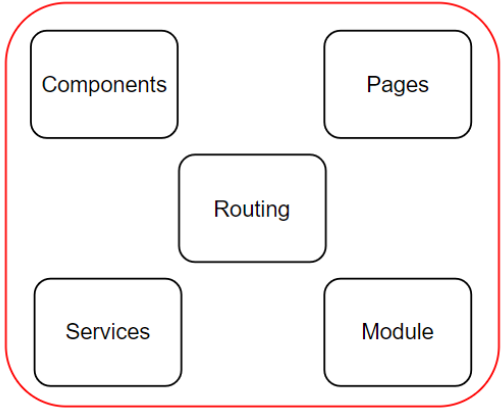
Cycle de vie des composants



Découpage technique







Projection et sélecteurs

- **Sélecteurs :**
  - Identification unique d'un composant (*selector*)

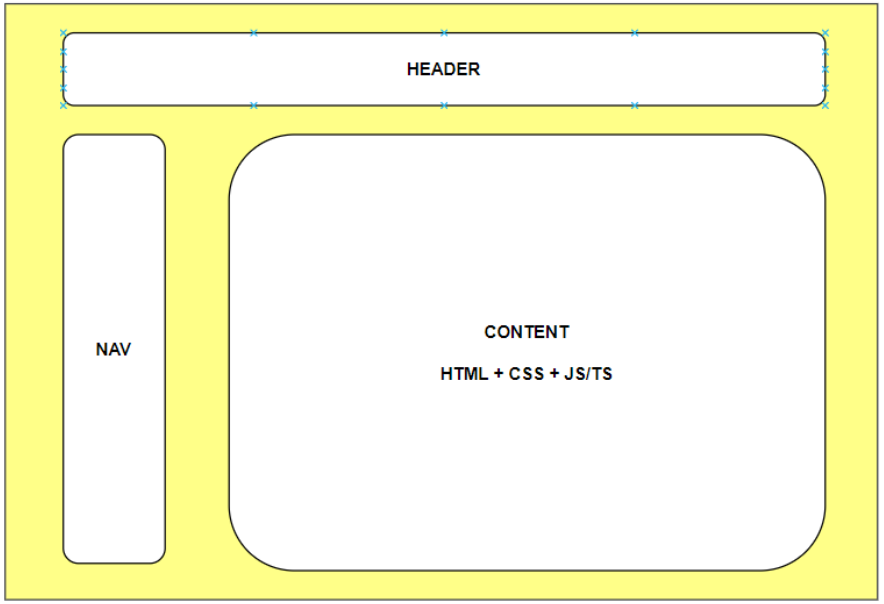
- **Projection :**
  - Mécanisme permettant de projeter du contenu
  - Projection d'un composant: *<balise_name ></balise_name >*
  - Projection de contenu non défini: *<ng-content ></ng-content >*

- **Visibilité des composants**
  - Interne au module
  - Propager la visibilité grace aux exports (composants, pipes ...)
  - Utiliser les composants grace aux imports (module)

Commandes clés

- **Les Schématics permettent de générer :**
  - des composants: ng generate component « *name* »
  - des modules: ng generate module « *name* »
  - des modules (avec routage): ng generate module « *name* » --routing=true
  - des services: ng generate service « *name* »
  - des pipes: ng generate pipe « *name* »
  - des directives: ng generate directive « *name* »
  - des enumerations : ng generate enum « *name* »
  - des interfaces : ng generate interface « *name* »
  - des modèles : ng generate class « *name* » --type=model
  - ...

Maquette du projet



Projet CRM

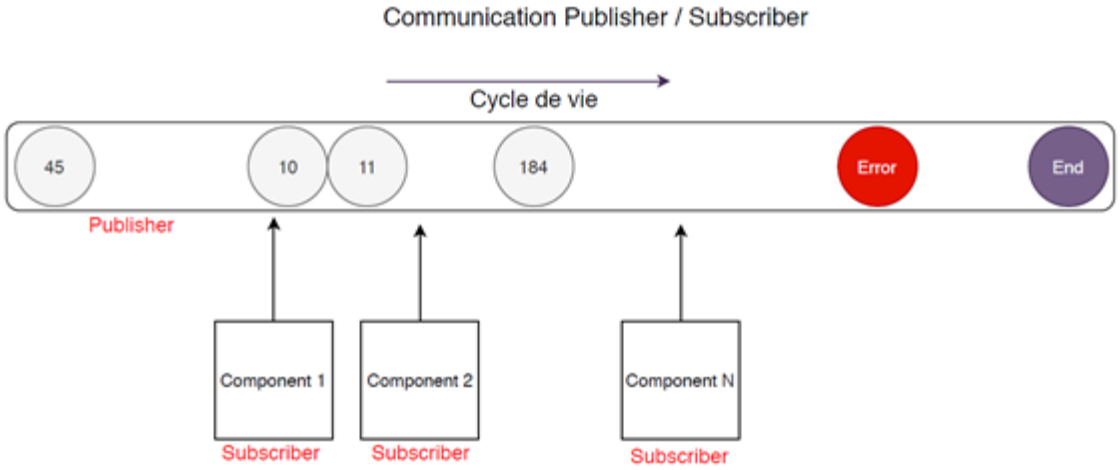
- **Tickets JIRA du backlog :**
  - M2IC-12
  - M2IC-13
  - M2IC-14
  - M2IC-15

4.

Les observables

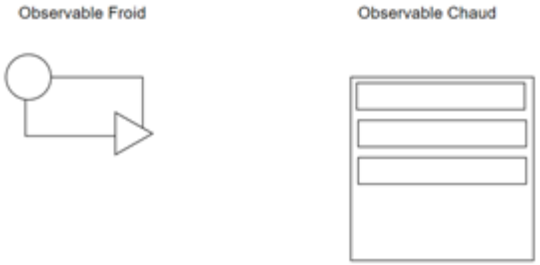
Qu’est ce qu’un observable ?

- **CallBack ? Promise ? Observable ?**



Observables et RxJS

- **Un observable peut être**
  - Synchrone (*Chaud*)
  - Asynchrone (*Froid*)

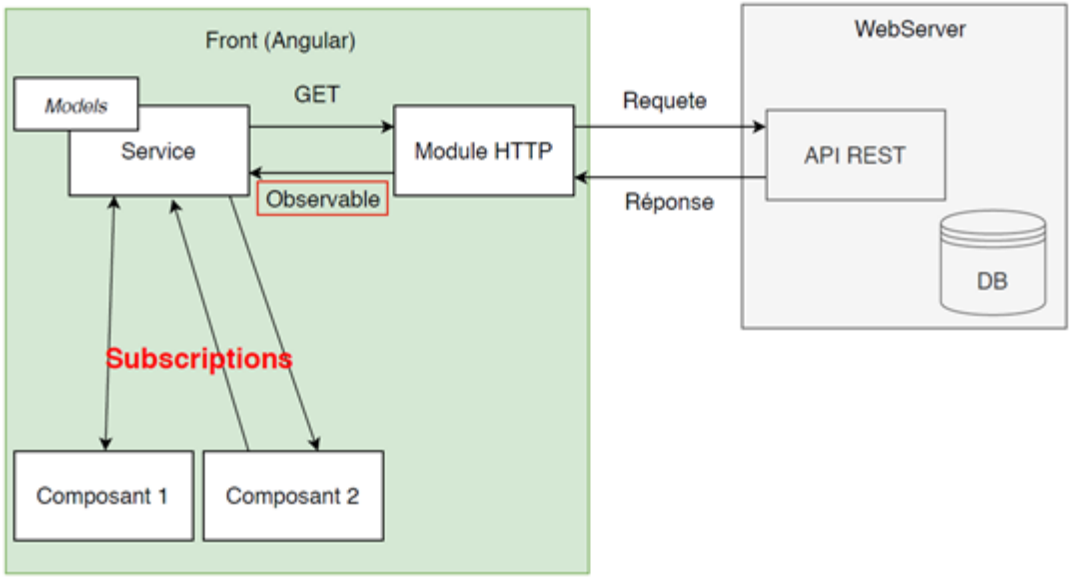


- **Librairie RxJS:**
  - Traitement sur des flux de données asynchrones
  - Traitement sur des événements
  - Utilisation d’opérateurs sur les observables (pipe, map, filter …)

- **Et les Subjects ?**

Architecture Client / Serveur

Archi C/S classique avec observables



Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-38
  - M2IC-40
  - M2IC-39

5.

Le Routage

Routage

- **Le routage permet de naviguer entre page (composants)**

- **Angular à son propre mécanisme de routage et s'implémente à l'aide de la balise** <router-outlet ></router-outlet>

- **2 types de routage : Classique vs Lazyloading**

- **Propriétés additionnelles au routage :**
  - Transmission de données
  - Transmission de paramètres
  - Controles des routes
  -

Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-41
  - M2IC-42
  - M2IC-16
  - M2IC-43
  - M2IC-44

6.

Les directives structurelles

Les directives structurelles ?

- Les directives structurelles d'Angular ont pour but de manipuler dynamiquement les éléments du DOM
- Elles commencent toutes par un astérisque
- Elles permettent d’ajouter des conditions ou itérer sur une liste ou un tableau

*ngIf, hidden et ng-template

- Le ***ngIf** et **hidden** permettent d'ajouter une condition dans l'affichage des éléments du DOM

- ***ngIf** supprime les éléments alors que **hidden** les masquent

- Il est possible d'ajouter un else à l'aide du **ng-template**

- La syntaxe est la suivante :

```
<div *ngIf="condition;else keywordElse">condition remplie</div>

<ng-template #keywordElse>Il s'agit du else</ng-template>
```

*ngFor

- **Le *ngFor permet de parcourir une liste dans le DOM**

- **Plusieurs informations peuvent être récupérés en plus de l'élément itéré de la liste :**
  - **index** : position de l'élément.
  - **even/odd** : indique si l'élément est à une position paire/impaire.
  - **first / last** : indique si l'élément est à la première/dernière position.

- **La syntaxe est la suivante :**

```
<div *ngFor="let book of bookList">{{book.name}}</div>
```

```
<div *ngFor="let book of bookList; let i=index; let a=first">
```

7.

Les décorateurs

Qu’est ce qu’un décorateur ?

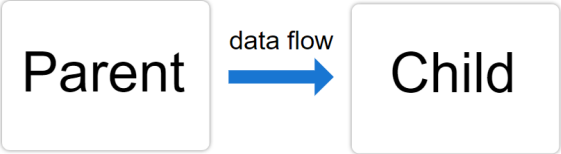
- **Annotation déclarative permettant d'ajouter des métadonnées sur une classe (@class)**
- **Implémenté par TypeScript / Imports depuis @angular/core**
- **Décorateurs générés avec les Schematics**
- **Exemples:** @Component, @Injectable, @Pipe …

Le décorateur @Input()

- **Utilisation du décorateur @Input() pour spécifier qu'une propriété est une entrée pour un composant ciblé.**

- **Le décorateur @Input() permet d'injecter / modifier la valeur d'une variable du composant récupéré.**

- **Un @Input() peut accepter tout type de valeur (number, string, Objet …**

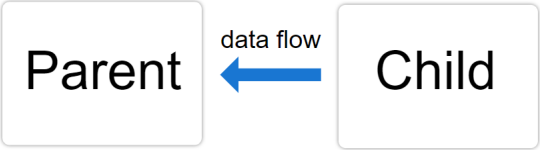


Le décorateur @Output()

- **Utilisation du décorateur @Output() pour spécifier qu'une propriété à un événement en sortie pour un composant ciblé.**

- **Le décorateur @Output() permet de transmettre une évènement.**

- **Spécificité : Ce dé lé à une propriété de type E**



Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-45
  - M2IC-46
  - M2IC-47

Le décorateur @Pipe()

- Un **Pipe** permet d'effectuer des filtres sur un template.

- Le **Pipe** va modifier les valeurs affichés sur le template HTML grâce au binding (ne modifie pas les données !)

- La syntaxe pour utiliser un pipe :
  - {{ var | pipeName }}
  - {{ var | pipeName = 'value' }} (avec param)
  - {{ var | pipeName1 | pipeName2 | ... }} (chainage)

Le décorateur @Directive()

- **Il existe deux type de directives :**
  - **directive structurelle**
  - **directive d'attribut**

- **Les directives structurelles modifient le DOM :**
  - ***ngIf**
  - ***ngFor**
  - **...**

- **Les directives d'attributs modifient un comportement :**
  - **[ngClass]**

Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-49
  - M2IC-50
  - M2IC-51

8.

Manipulation/Accès au DOM

Qu’est ce que le DOM ?

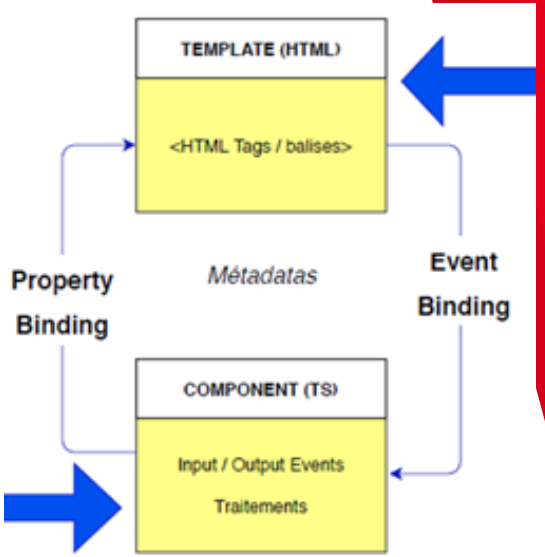
- **Document Object Model (DOM) :**
  - Représentation des éléments sources d’une page HTML

- **Shadow DOM : Provient des normes des composants**
  - Permet d’encapsuler des parties du DOM
  - Permet de surcharger des styles
  - Primordial pour Angular => Web Components

- **ViewEncapsulation propose par Angular :**
  - Native
  - None
  - Emulated

Manipulation des données

- **Type d'échange de données :**
  - Interpolation :
    - {{ varExample }}
  - Property Binding :
    - <button type="bouton" [dsabled]="var">
  - Event Binding :
    - <button type="bouton" (click)="test()">
  - Two-way Data Binding ;
    - <app [(size)]="fontSizePx"></app>



Les Guards

- **Un Guard permet de restreindre et contrôler un accès à une route.**

- **Les Guards sont ajoutés dans les fichiers de configuration des routes.**

- **La méthode CanActivate est implémenté par le Guard et retourne une valeur de type Boolean ou Observable<Boolean>**

Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-52
  - M2IC-53
  - M2IC-54

9.

Les formulaires

Angular et les formulaires

- **Utilité d'un formulaire :**
  - Créer une interaction avec un utilisateur, en s'appuyant sur de la data (création, édition, visu.. etc)

- **La librairie Reactive Forms : Tooblox complète**
  - Class FormControl => Contrôles
  - Class FormGroup => Regroupements de contrôles
  - Class FormArray => Création d'une liste de contrôles
  - Class FormBuilder => Constructeur de formulaire

- **Les Validators permettent d'ajouter des contrôles sur des champs**

Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-55
  - M2IC-56
  - M2IC-57
  - M2IC-58
  - M2IC-59
  - M2IC-60
  - M2IC-61

10.

Allons plus loin ...

Projet CRM

- **Tickets JIRA du backlog :**
  - M2IC-62
  - M2IC-63
  - M2IC-64
  - M2IC-65
  - M2IC-66
  - M2IC-67

THE END.

Avec tous nos remerciements et toutes nos félicitations pour avoir suivi ce cursus.

Christophe Guéroult : c.gueroult@coderbase.io
Pierre Nédélec : p.nedelec@coderbase.io
Nicolas Haziza : email formateur
Olivier Blaivie : o.blaivie@gmail.com
Marc Gilbert : email formateur